Radio Shack

Cat. No. 62-2075

More TRS-80 Assembly Language Programming

By Bill Barden

An Advanced Look at Challenging, Efficient Assembly-Language Programs—A Valuable Tool for A Variety of Applications

U.S.A. $5.95

MORE TRS-80 ASSEMBLY-LANGUAGE PROGRAMMING

by

William Barden, Jr.

Radio Shack®
A DIVISION OF TANDY CORPORATION

FIRST EDITION
FIRST PRINTING

International Standard Book Number:
Library of Congress Catalog Card Number:

Printed in USA

Preface

This book is a follow-up to Radio Shack's book *TRS-80 Assembly-Language Programming*. In that book we described the architecture of the Z-80 microprocessor used in the TRS-80, presented the instruction set and addressing modes, and gave some examples of common assembly-language programming operations.

More TRS-80 Assembly-Language Programming builds on the material found in the previous book by taking an even more practical look at TRS-80 Model I and III assembly-language programming. It answers such questions as "How do I use the Disk Editor/Assembler?," "Where in memory should I put an assembly-language program?," "What about embedded machine-language code?" and "How do I go about writing and implementing a large assembly-language program?"

We have organized the material in this book into four sections. In the first section, "Using Assembly Language," we review some of the general material we covered in the earlier book and then describe the practicalities of assembling programs in the TRS-80. Among the practicalities we discuss are the operation of both the cassette-based Editor/Assembler and Disk Editor/Assembler. Moving along, we get into the general approaches to executing assembly-language programs, both "stand-alone" assembly-language programs and the "embedded in BASIC" machine-language approach.

Section II, "Assembly-Language Techniques," describes four types of processing that can be implemented in assembly language — "number-crunching;" working with character data; table operations, including sorting and searching; and graphics display processing. We present each type of processing first from a general design standpoint — and then we put that theory into practice, with assembly-language code to illustrate the methods of solving the problems of each type of processing.

Because of the unique structure and design of the TRS-80 hardware there are a number of assembly language techniques that are peculiar to this system. Along that line, we take two more chapters in this section to describe cassette input/output, parallel printer operation, and disk file manage operations. Some of you may be especially interested in the material on disk file operations since assembly-language calls may be made directly to TRSDOS disk file manage routines to read and write random files and perform other operations.

The third section, "Larger Assembly-Language Projects," contains complete listings of two large assembly-language programs. The first of these is a Morse Code Program (MORG), which transmits random or defined Morse code messages through the cassette output port at speeds of 5 to 60 words per minute. The second program is an experiment in artificial intelligence — a tic-tac-toe learning program that starts off being not very bright but learns how to play the game until it is virtually unbeatable. Neither program could be implemented in BASIC to execute in "real-time." Also in this section we discuss the general approach in writing large programs in assembly-language along with an actual case study from a TRS-80 software house.

The fourth and final section contains the appendices covering the Z-80 instruction set grouped in functional order and Z-80 operation code listings.

As the author, I hope you find the material in this book a useful supplement to *TRS-80 Assembly-Language Programming,* but I also hope you will be able to use it in its own right. Assembly-language programs are fast, efficient, and challenging, and help to make the TRS-80 even more of a valuable tool for all types of programming applications.

To my wife Janet, for her encouragement

Table Of Contents
More TRS-80 Assembly-Language Programming

Section I: Using Assembly Language

Chapter Five

Section II: Assembly-Language Techniques

Chapter Six

Chapter Seven

Chapter Eight

Chapter Fourteen

Appendices

SECTION I
Using Assembly Language

Chapter One
Assembly-Language Basics

In this chapter we'll review some assembly-language basics. If you want a more comprehensive treatment, refer to *TRS-80 Assembly-Language Programming* (Radio Shack 62-2006). If you've never experimented with assembly language, review some of the material presented in the previous book (especially the "General Concepts" section). An alternative is to read this chapter thoroughly and pay close attention to assembly-language examples we've included later in the book.

> By the way, when we use the term "TRS-80" in this book we'll be referring to both the Model I and III. There are some minimal differences between the two systems (which we'll note) but in general, material in this book applies to either system.

The TRS-80 uses the built-in instruction set of the Z-80A microprocessor. The Z-80A is a third-generation microprocessor chip that is truly a "computer on a chip." Using assembly language is mainly a matter of learning both the instruction set of the Z-80A microprocessor and the skills needed in putting together those instructions to form programs.

There are over 700 separate instructions for the Z-80 with a number of different addressing modes. Each of the instructions, however, performs a simple function and is easy to understand by itself. Furthermore, many of the instructions are similar in nature, and understanding how one specific instruction works might mean that you can easily unravel 23 similar ones!

The instruction set of the Z-80 is provided in Appendix I with the instructions grouped according to logical function. If you want to find an instruction that will load the A register in the Z-80 with a memory operand, for example, you could look under "Loads" and then under "A Load Memory Operand" to find the instructions that perform this function. Many times you'll find a number of different instructions can accomplish the same function.

Appendix II gives the actual format of each instruction. Normally, you will not be concerned with the format, as the Assembler will automatically translate a **mnemonic** for the instruction into the **machine-language** format.

It's possible to translate a string of assembly-language instructions by hand into the equivalent bits shown in the instruction formats of Appendix II. If you did this, the result would be called a **machine-language program.** However, there are a number of excellent automatic **assembler programs** that Radio Shack provides that eliminate such tedious hand work. One of these is the cassette-based Editor/Assembler which takes **symbolic source** lines and translates them into machine-language code. The second is the disk-based Editor/Assembler, a more advanced version of an assembler. We'll look at both assemblers in this book and point out any differences that occur between them.

A Typical Assembly Program

One listing is said to be worth a thousand words . . . Let's take a look at a typical assembly-language program and scrutinize it in some detail to review some of the things we should know before we go on to advanced topics.

The assembly-program **listing** is shown in Figure 1-1. The program itself is designed to convert a character string representing a decimal value of 0-65535 into an equivalent 16-bit binary value. This program could be used, for example, to convert a keyboard input to binary for processing.

```
                08970 ;********DECIMAL TO BINARY CONVERSION SUBROUTINE*********
                08980 ;*   CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING   *
                08990 ;*   DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535.  *
                09000 ;*   ENTRY: (HL)=BUFFER CONTAINING ASCII                *
                09010 ;*          (B)=NUMBER OF CHARACTERS                    *
                09020 ;*   EXIT:  (HL)=BINARY # 0-65535                       *
                09030 ;*          NZ IF INVALID ASCII CHARACTER OTHERWISE Z   *
                09040 ;*   ALL REGISTERS SAVED EXCEPT A,HL                    *
                09050 ;
8590 C5         09060 DECBIN  PUSH   BC             ;SAVE REGISTERS
8591 D5         09070         PUSH   DE
8592 DDE5       09080         PUSH   IX
8594 DD210000   09090         LD     IX,0           ;SET RESULT
8598 DD29       09100 DEC040  ADD    IX,IX            ;INTERMEDIATE*2
859A DDE5       09110         PUSH   IX
859C DD29       09120         ADD    IX,IX            ;*4
859E DD29       09130         ADD    IX,IX            ;*8
85A0 D1         09140         POP    DE               ;*2
85A1 DD19       09150         ADD    IX,DE            ;*10
85A3 7E         09160         LD     A,(HL)           ;GET CHARACTER
85A4 D630       09170         SUB    30H              ;CONVERT
85A6 FAB685     09180         JP     M,DEC070         ;GO IF LT "0"
85A9 FE0A       09190         CP     10               ;TEST FOR GT "9"
85AB F2B685     09200         JP     P,DEC070         ;GO IF GT "9"
85AE 5F         09210         LD     E,A              ;NOW IN E
85AF 1600       09220         LD     D,0              ;NOW IN DE
85B1 DD19       09230         ADD    IX,DE            ;MERGE
85B3 23         09240         INC    HL
85B4 10E2       09250         DJNZ   DEC040           ;GO IF MORE
85B6 78         09260 DEC070  LD     A,B            ;COUNT TO A
85B7 B7         09270         OR     A              ;SET OR RESET Z FLAG
85B8 DDE5       09280         PUSH   IX             ;RESULT TO HL
85BA E1         09290         POP    HL
85BB DDE1       09300         POP    IX             ;RESTORE REGISTERS
85BD D1         09310         POP    DE
85BE C1         09320         POP    BC
85BF C9         09330         RET                   ;RETURN
```

Figure 1-1. Typical Assembly-Language Program

This collection of assembly-language statements makes up a **subroutine**, a structure very similar to a BASIC subroutine. It is located at one place in memory and can be **called** as often as necessary.

There are three main segments of the assembly-language program: the assembly-language **source code**, the edit line numbers and the assembly-language **machine code.**

Figure 1-2 shows the assembly-language source code portion of the listing. The source code consists of symbolic lines similar to BASIC statement lines. In general, each line represents one assembly-language instruction written in **mnemonic** form. The mnemonic of the instruction is merely a shorthand way to express the instruction. It's much simpler to write DEC DE than to write "take the contents of the DE register, subtract one, and put the results of the operation back into DE".

```
                    LABEL   OP-CODE
                    FIELD   FIELD    OPERAND FIELD        COMMENTS FIELD

                    ;********DECIMAL TO BINARY CONVERSION SUBROUTINE*********
                    ;*    CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING    *
                    ;*    DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535.   *
                    ;*    ENTRY: (HL)=BUFFER CONTAINING ASCII                 *
                    ;*           (B)=NUMBER OF CHARACTERS                     *
                    ;*    EXIT:  (HL)=BINARY # 0-65535                        *
                    ;*           NZ IF INVALID ASCII CHARACTER OTHERWISE Z    *
                    ;*    ALL REGISTERS SAVED EXCEPT A,HL                     *
                    ;
8590 C5             DECBIN  PUSH    BC                  ;SAVE REGISTERS
8591 D5                     PUSH    DE
8592 DDE5                   PUSH    IX
8594 DD210000               LD      IX,0                ;SET RESULT
8598 DD29           DEC040  ADD     IX,IX                 ;INTERMEDIATE*2
859A DDE5                   PUSH    IX
859C DD29                   ADD     IX,IX                 ;*4
859E DD29                   ADD     IX,IX                 ;*8
85A0 D1                     POP     DE                    ;*2
85A1 DD19                   ADD     IX,DE                 ;*10
85A3 7E                     LD      A,(HL)                ;GET CHARACTER
85A4 D630                   SUB     30H                   ;CONVERT
85A6 FAB685                 JP      M,DEC070              ;GO IF LT "0"
85A9 FE0A                   CP      10                    ;TEST FOR GT "9"
85AB F2B685                 JP      P,DEC070              ;GO IF GT "9"
85AE 5F                     LD      E,A                   ;NOW IN E
85AF 1600                   LD      D,0                   ;NOW IN DE
85B1 DD19                   ADD     IX,DE                 ;MERGE
85B3 23                     INC     HL
85B4 10E2                   DJNZ    DEC040                ;GO IF MORE
85B6 78             DEC070  LD      A,B                 ;COUNT TO A
85B7 B7                     OR      A                   ;SET OR RESET Z FLAG
85B8 DDE5                   PUSH    IX                  ;RESULT TO HL
85BA E1                     POP     HL
85BB DDE1                   POP     IX                  ;RESTORE REGISTERS
85BD D1                     POP     DE
85BE C1                     POP     BC
85BF C9                     RET                         ;RETURN
```

Figure 1-2. Source Code

There are four **fields** in each assembly-language source line.

The second field is the mnemonic representing the instruction to be used. Each of the 700 or so instructions has a predefined mnemonic that the assembler recognizes as a valid instruction. As the mnemonic defines an **operation code** for the instruction, this field is often referred to as the **op-code** field.

The third field in the source line is the **operand** field. The number of operands for instructions varies from none to three. The RET instruction shown in the figure, for example, requires no operands, while one of the LD instructions requires two — one specifying the register pair that points to a memory address {(HL)}, and a second specifying the register to be loaded with the contents of that memory address (A). The complete op-code and operand form of the instruction is LD A,(HL).

The fourth field of the instruction is an optional **comments** field. This field is used solely for comments associated with the instruction, similar to the "REMarks" statement of BASIC. A source line is a **comment line** when the line starts with a semicolon (;). *The comments field must always start with a semicolon.*

The remaining field of the source line is the optional **label field.** We use this field to define a label for the instruction, which can then be referenced by another instruction in the program. The line DEC040 ADD IX,IX, for example, has the label DEC040 for the ADD instruction. A later instruction, DJNZ DEC040, causes a jump to DEC040 if there is a zero result in the B register. The use of symbolic names for instruction locations makes it unnecessary to keep track of the absolute locations for each instruction in memory. Keeping track of absolute locations is a burdensome chore, although it can be done in **machine-language programming**, which does not use an assembler. The label may or may not be suffixed by a colon, depending upon the assembler.

The second part of the assembly listing, shown in Figure 1-3, is the **editing of line numbers**. The source code lines are entered from the keyboard to an **editor program**, which is usually part of an Editor/Assembler package. The Editor uses line numbers for each of the source lines with the line numbers in ascending sequence. The typical line number increment is 10, so that the line numbers are 100, 110, 120, etc. Using the Editor, lines may be modified, inserted, or deleted. Characters within the lines may also be processed.

```
                08970 ;********DECIMAL TO BINARY CONVERSION SUBROUTINE*********
                08980 ;*   CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING   *
                08990 ;*   DECIMAL NUMBER TO BINARY. MAXIMUN VALUE IS 65535.  *
                09000 ;*   ENTRY: (HL)=BUFFER CONTAINING ASCII                *
                09010 ;*          (B)=NUMBER OF CHARACTERS                    *
                09020 ;*   EXIT:  (HL)=BINARY # 0-65535                       *
                09030 ;*          NZ IF INVALID ASCII CHARACTER OTHERWISE Z   *
                09040 ;*   ALL REGISTERS SAVED EXCEPT A,HL                    *
                09050 ;
8590 C5         09060 DECBIN  PUSH    BC              ;SAVE REGISTERS
8591 D5         09070         PUSH    DE
8592 DDE5       09080         PUSH    IX
8594 DD210000   09090         LD      IX,0            ;SET RESULT
8598 DD29       09100 DEC040  ADD     IX,IX            ;INTERMEDIATE*2
859A DDE5       09110         PUSH    IX
859C DD29       09120         ADD     IX,IX            ;*4
859E DD29       09130         ADD     IX,IX            ;*8
85A0 D1         09140         POP     DE               ;*2
85A1 DD19       09150         ADD     IX,DE            ;*10
85A3 7E         09160         LD      A,(HL)           ;GET CHARACTER
85A4 D630       09170         SUB     30H              ;CONVERT
85A6 FAB685     09180         JP      M,DEC070         ;GO IF LT "0"
85A9 FE0A       09190         CP      10               ;TEST FOR GT "9"
85AB F2B685     09200         JP      P,DEC070         ;GO IF GT "9"
85AE 5F         09210         LD      E,A              ;NOW IN E
85AF 1600       09220         LD      D,0              ;NOW IN DE
85B1 DD19       09230         ADD     IX,DE            ;MERGE
85B3 23         09240         INC     HL
85B4 10E2       09250         DJNZ    DEC040           ;GO IF MORE
85B6 78         09260 DEC070  LD      A,B             ;COUNT TO A
85B7 B7         09270         OR      A               ;SET OR RESET Z FLAG
85B8 DDE5       09280         PUSH    IX              ;RESULT TO HL
85BA E1         09290         POP     HL
85BB DDE1       09300         POP     IX              ;RESTORE REGISTERS
85BD D1         09310         POP     DE
85BE C1         09320         POP     BC
85BF C9         09330         RET                     ;RETURN
```

Figure 1-3. Editing Line Numbers

It's important to note that the edit line numbers are used for editing purposes only and are not used in the program (as they are in BASIC) to refer to other source lines. The output of the Editor is an assembly-language **source file**, a collection of source lines that represent a source program. This file is stored on cassette tape or disk, depending upon the Editor/Assembler.

The third part of the assembly-language listing is shown in Figure 1-4. This is the **machine code** portion of the assembly. When the Assembler portion of the Editor/Assembler processes the source file, the Assembler **translates** the source lines into the corresponding machine code form of the instruction. For example, the Assembler translates the `OR A . . .` instruction into a hexadecimal "B7". The hex B7 corresponds to a binary 10110111, which the Z-80 microprocessor will decode as an OR instruction that ORs the contents of the A register with the A register. The Assembler automatically translates the symbolic source lines into a form that the Z-80 processor can recognize, binary ones and zeroes.

MEMORY LOCATION
FOR INSTRUCTION

DATA REPRINTING
INSTRUCTION

```
                    08970 ;********DECIMAL TO BINARY CONVERSION SUBROUTINE*********
                    08980 ;*   CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING   *
                    08990 ;*   DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535.  *
                    09000 ;*   ENTRY: (HL)=BUFFER CONTAINING ASCII                *
                    09010 ;*          (B)=NUMBER OF CHARACTERS                    *
                    09020 ;*   EXIT:  (HL)=BINARY # 0-65535                       *
                    09030 ;*          NZ IF INVALID ASCII CHARACTER OTHERWISE Z   *
                    09040 ;*   ALL REGISTERS SAVED EXCEPT A,HL                    *
                    09050 ;
8590 C5             09060 DECBIN  PUSH    BC              ;SAVE REGISTERS
8591 D5             09070         PUSH    DE
8592 DDE5           09080         PUSH    IX
8594 DD210000       09090         LD      IX,0            ;SET RESULT
8598 DD29           09100 DEC040  ADD     IX,IX             ;INTERMEDIATE*2
859A DDE5           09110         PUSH    IX
859C DD29           09120         ADD     IX,IX             ;*4
859E DD29           09130         ADD     IX,IX             ;*8
85A0 D1             09140         POP     DE                ;*2
85A1 DD19           09150         ADD     IX,DE             ;*10
85A3 7E             09160         LD      A,(HL)          ;GET CHARACTER
85A4 D630           09170         SUB     30H             ;CONVERT
85A6 FAB685         09180         JP      M,DEC070        ;GO IF LT "0"
85A9 FE0A           09190         CP      10              ;TEST FOR GT "9"
85AB F2B685         09200         JP      P,DEC070        ;GO IF GT "9"
85AE 5F             09210         LD      E,A             ;NOW IN E
85AF 1600           09220         LD      D,0             ;NOW IN DE
85B1 DD19           09230         ADD     IX,DE           ;MERGE
85B3 23             09240         INC     HL
85B4 10E2           09250         DJNZ    DEC040          ;GO IF MORE
85B6 78             09260 DEC070  LD      A,B             ;COUNT TO A
85B7 B7             09270         OR      A               ;SET OR RESET Z FLAG
85B8 DDE5           09280         PUSH    IX              ;RESULT TO HL
85BA E1             09290         POP     HL
85BB DDE1           09300         POP     IX              ;RESTORE REGISTERS
85BD D1             09310         POP     DE
85BE C1             09320         POP     BC
85BF C9             09330         RET                     ;RETURN
```

Figure 1-4. Machine Code

The second column of Figure 1-4 is the hexadecimal representation of all the assembly source lines. As there are two hexadecimal digits per 8 bits (one byte), each pair of hex digits represents one byte of the instruction. Z-80 instructions may be one, two, three, or four bytes long, as you can see from the figure. Normally, the assembly-language programmer should know the rudiments of binary and hexadecimal representation. If you're not familiar with binary or hexadecimal numbers, spend a few hours reviewing a basic text on the subject.

Hints and Kinks 1-1

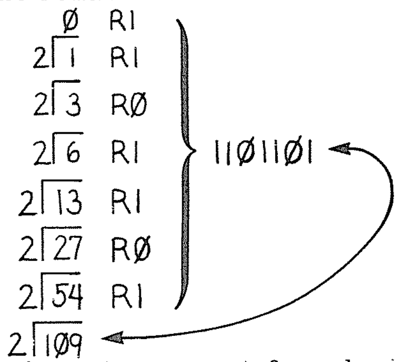
Binary/Hexadecimal Representation

Hexadecimal is just a shorthand way to represent binary numbers. To convert from binary to hex, group the binary number into 4-bit groups and then convert to a hex digit 0–9, A, B, C, D, E, or F.

Binary	0011	1011	1001	1111
Hex	3	B	9	F

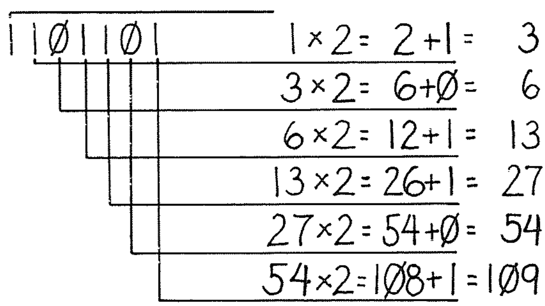
To convert back, perform the operation in reverse.

To convert from decimal to binary, divide by two and arrange the remainders in reverse order.



Use the same scheme to convert from decimal to hexadecimal.

To convert from binary to decimal, use ''double-dabble.''



Use the same scheme to convert from hex to decimal.

. . . . . I know – Right now you're saying, ''If God had wanted man to count in hexadecimal, he would have given him 16 fingers . . . . . '' You'll be surprised, however, at how easy it is to work in binary or hex after some practice. Soon you'll be balancing your checkbook in it! Hmmmmm . . . Maybe that would help mine . . . .

The first column of the machine code portion, which is in hexadecimal notation, represents the **location** of the machine-code data in memory. The instruction `PUSH BC`, for example, has been assembled at location 8590H. If you were to take the machine code from column two and enter it into RAM *(Random Access Memory)* starting at location 8590H (by using T-BUG or DEBUG), the 48 bytes from 8590H to 85BFH would represent the machine-language program for the DECBIN subroutine. The location column often represents the **absolute location** of the instructions, although with the Disk Editor/Assembler it may represent the **relative locations** from the start (we'll see how in a later chapter).

How the DECBIN Subroutine Works

Now let's take a look at the operation of DECBIN so we can review some basic concepts about Z-80 architecture, instructions, and addressing. Before we wrote DECBIN, we knew the functions it had to perform — we wanted to write some assembly-language code that would take a given ASCII string representing a decimal number and convert it to binary form.

Subroutines

It's convenient to write this code as a **subroutine**. A subroutine is nothing more than a collection of instructions that performs a particular function. Subroutines are handy — rather than write the instructions each time we need the function, we define the subroutine once so it occupies one particular area in memory, and then we call it up whenever we need that function in the program. This little convenience saves memory space since the code only occupies one point in memory. It also saves us the development time of writing the source lines over and over. Assembly-language subroutines are functionally identical to BASIC subroutines.

Subroutines also are important for another reason. They break the program up into a number of small modules that perform well-defined functions. This pattern of modules makes the entire programming task much easier than writing a large amount of **in-line** code.

We can call the DECBIN subroutine by a CALL instruction that is very similar to a BASIC GOSUB statement. The CALL instruction **jumps** to the subroutine but saves the return address of the next instruction after the CALL. When the RET, or RETurn instruction in DECBIN is finally executed at completion, the **return address** of the instruction after the CALL is retrieved or **popped** from the **memory stack**, and a return is made.

The appearance of the CALL might be:

```
NAME  CALL  DECBIN ;THIS IS THE CALL TO DECBIN
      LD    A,1    ;RETURN HERE
      .
      .
      .
```

Stack Operations

The memory stack is an area of RAM set aside for certain functions. Whenever a CALL instruction is executed, the current address of the Z-80 **program counter** register is saved in the stack area; the address is "pushed" onto the stack as shown in Figure 1-5. Whenever a RET instruction is executed, the return address is retrieved, or "popped" from the stack and put into the program counter to cause a return to the instruction following the CALL. The stack is also used as temporary storage, as we shall see. The stack area set aside for this function and others is typically 100 bytes.

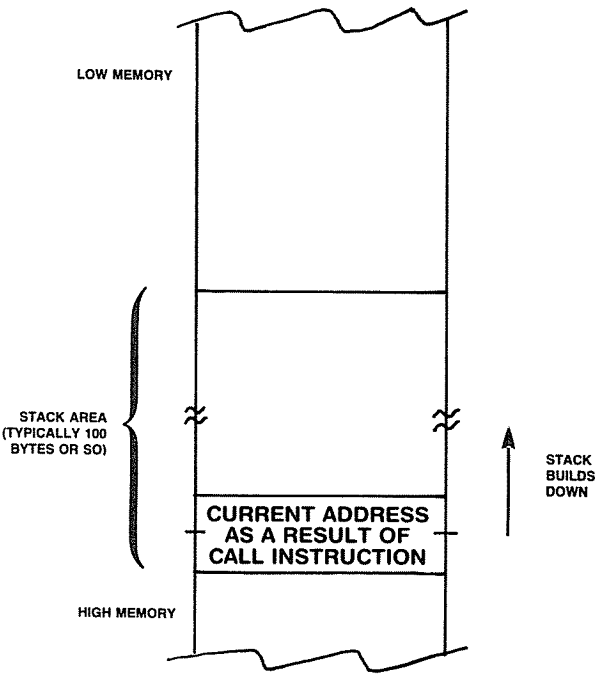
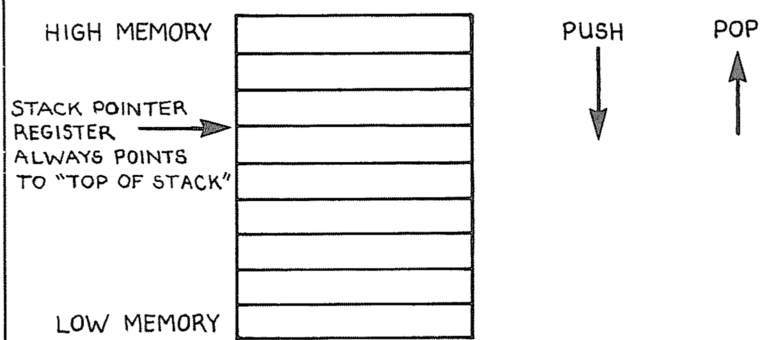


Figure 1-5. Stack Action for a CALL/RET

Hints and Kinks 1–2
Stack use

It's easy to remember how the stack works if you think of a dinner plate stacker in a restaurant. (No, not Ed, the part time counter man – the mechanical one!) As a plate (8 bits of data) is placed on the stack, the stack is pushed down. As a plate (data) is taken off the stack, the stack pops up. Every time a CALL is made, two bytes of data, making up two address bytes, are pushed onto the stack. Every RETurn pops the two bytes. PUSH and POP instructions operate similarly.



To set the stack pointer, perform the "LD SP,XXXX" instruction, where XXXX is some (usually) high memory location, as the stack ''builds down''.

Just as every he must have a she, every PUSH must have a POP and every CALL must have a RET! Otherwise stack will "gobble up" memory below it as it digests byte after byte, pushing data further and further down!

Before we can call the DECBIN subroutine, we must set up the subroutine **arguments** in preparation for the subroutine action. The arguments are what might be popularly known as the "gozintas" and the "gozoutas." In this subroutine, we're **entering** with a pointer to a string of ASCII characters representing decimal digits 0-9 and the number of characters in the string. That's the "goes into." We're **exiting** (the "goes out of") with a binary number from 0 to 65535 representing the converted result. Also, if we detect an invalid ASCII character in the string, such as the one in "123$56" — we want to know about it. In the case of "123$56", the **Z flag** in the Z-80 is reset (an "NZ" condition).

When we started writing the subroutine, we had to do some thinking about how to pass the arguments. One of the arguments is a pointer to a string of ASCII characters. The characters are identical to the ones we would get by typing in characters from the TRS-80 keyboard. Each character takes up one byte, as shown in Figure 1-6. We would like to convert the string of characters into the equivalent binary number. Since we must put some limit on the number to be converted, we chose 65535, a number that can be held in 16 bits, which is the size of a Z-80 register pair. It seems convenient, then, to return the result in a Z-80 register pair of 16 bits.

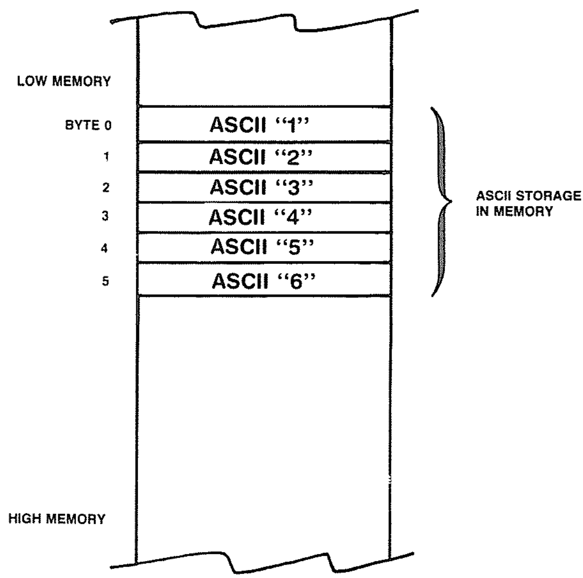


Figure 1-6. ASCII Character Storage

Hints and Kinks 1-3

ASCII

ASCII is simply a standardized 7-bit code used for uniform representation of character data. See the tables in many of the TRS-80 manuals. All BASIC strings are stored in ASCII. Data to be displayed on the screen or line printer must first be converted to ASCII by software.

Fortunately the ASCII codes for 0-9 and A-Z and a-z run in sequence, making it much easier to convert. This is not always true of character codes. Some peripherals have codes based on type-ball position, the phase of the moon, or both!

Passing Arguments

How would we pass the ASCII string to the subroutine? With a limit of 65535, we would have a maximum of 5 bytes. Although we could pass these five bytes in five Z-80 registers (or 2½ register pairs), we'll choose instead to pass a **pointer** to the memory locations that contain the string. This pointer value can be held in 16 bits, as any TRS-80 memory location can be expressed as a value of 0 through 65535.

The number of bytes in the string may vary from 1 to 5; this is also an argument that must be passed to the DECBIN subroutine. As this is a small value, it can be passed in a single Z-80 register of 8 bits.

Z-80 Registers

Any of the general purpose (A, B, C, D, E, H, or L) registers or register pairs (BC, DE, HL) could have been chosen to hold the arguments. We have chosen HL to hold the string pointer and B to hold the count — since HL is convenient because many of the instructions use the HL register pair as a **register indirect pointer** to memory locations. B is conveniently used, as a special DJNZ instruction uses B as a counter. A typical CALL to the DECBIN might now look like this:

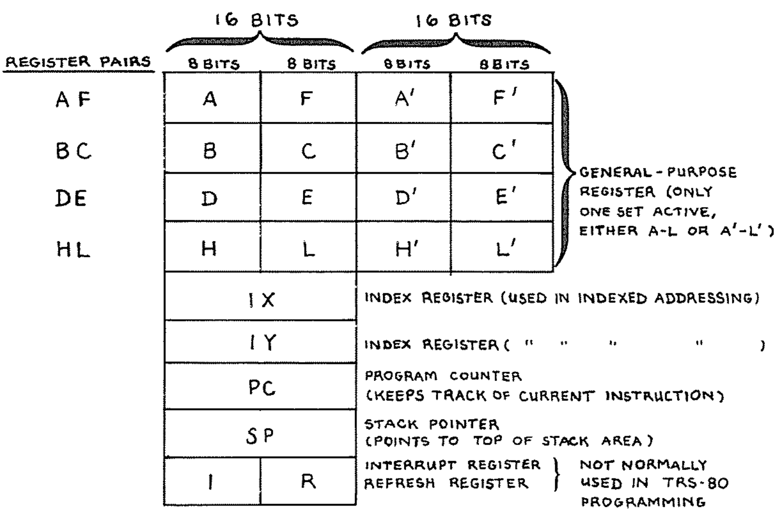
```
NAME1  LD    HL,BUFFER    ;ADDRESS OF
                          STRING BUFFER

       LD    B,5          ;# OF CHARACTERS

       CALL  DECBIN       ;CONVERT ASCII
                          TO BINARY

       LD    A,1          ;RETURN HERE

       .

       .

       .
```

Hints and Kinks 1–4

Z–80 Registers

These are the Z–80 registers accessible to TRS–80 assembly–language programmers. A is used for many arithmetic operations. The HL register pair is used as a ''16–bit accumulator'' for 16–bit arithmetic. Selection of either AF or AF' is done by `EX AF,AF'`. Selection of either B–L or B'–L' is done by `EXX`.

Seem like it is easy to get confused about which set of registers, prime or non–prime you are using? You bet. Much TRS–80 software uses only one set. Typically the prime set would be used for interrupt processing. There is no reason that you should not use both sets, however, as long as you keep track of which set is ''current.''



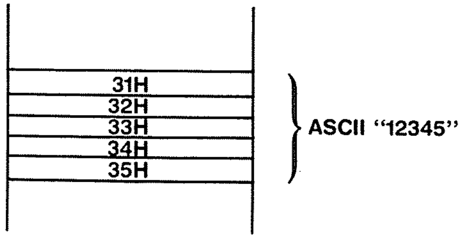
In the above sequence, the HL register pair is loaded with the address of the string **buffer**, the locations at which the ASCII characters are stored. Note that the `LD HL,XX` instruction (see Appendix II) is an **immediate-load** type instruction that loads a 16-bit value into the HL register pair. In this case, the assembler will load a 16-bit address corresponding to the location of label **BUFFER** into HL. We have loaded the B register with 5 as a typical string length. The CALL is then made to the DECBIN subroutine, with a return after the conversion to the instruction following the CALL.

Algorithm for DECBIN

Now let's get into the operation of DECBIN itself. The sequence of operations for DECBIN goes something like this:

1. Clear a result variable.
2. Get an ASCII digit from memory, starting from the leftmost digit of the string.
3. Convert the ASCII digit to binary. (Since an ASCII "O" through "9" is represented by hexadecimal 30 through 39, this involves subtracting 30H from the ASCII digit.)
4. Add the result of the subtract, binary 0-9, to the total.
5. If this is the last ASCII digit, you're done. If it's not the last digit, multiply by 10 and go back to step 2.

We've used a slightly modified version of this **algorithm** in DECBIN. Rather than checking for the last ASCII digit and **then** multiplying by 10, we'll multiply by 10 as an initial action. This process is illustrated in Figure 1-7, for an ASCII string of 5 digits.



STEP	RESULT	RESULT*10	ASCII DIGIT	DIGIT-30H	ADD TO RESULT
1	0	0	31H	1	1
2	1	10	32H	2	12
3	12	120	33H	3	123
4	123	1230	34H	4	1234
5	1234	12340	35H	5	12345

Figure 1-7. Algorithm for DECBIN

DECBIN — Detailed Analysis

Now let's go through the DECBIN subroutine to see how this is implemented in code. The first nine lines of the subroutine are comment lines describing the action of the subroutine, the entry conditions, and the exit conditions. The remainder of the subroutine implements the algorithm.

The name of the subroutine DECBIN, is expressed by the label opposite the first instruction mnemonic. During assembly time, this label will be equated to the location of the subroutine, and any `CALL DECBIN` will assemble as a CALL to the proper absolute location.

Hints and Kinks 1-5
Symbol Table

As the Assembler does its work, it builds a symbol table in memory. Every symbol encountered is put into the table as a six-character name and associated address value. Any references to the symbol cause the Assembler to search the symbol table for the value of the symbol.

SYMBOL	VALUE	
DEC Ø2Ø	8ØØØ	
DEC Ø4Ø	8XXX	
DEC Ø7Ø	8XXX	
EDD ƀƀƀ	FFFF	
EDD Ø7Ø	9XXX	
ENN ƀƀƀ	9AAF	
FOO ƀƀƀ	7XXX	
SEX ƀƀƀ	9ABC	

VALUE — USUALLY CORRESPONDS TO LOCATION OF LABEL IN MEMORY

FFFF — AN "UNRESOLVED SYMBOL" NOT YET ENCOUNTERED IN SOURCE LINE OR COULD ALSO BE "UNDEFINED"

9ABC — A... OH, NEVER MIND...

The first three instructions PUSH the BC and DE register pairs and the IX register onto the stack, which allows the stack to be used as a temporary storage area. The contents of BC, DE, and IX, six bytes in all, will be saved in the stack area as shown in figure 1-8. They will be retrieved with subsequent POP instructions. "Why," you ask, "are the contents of BC, DE, and IX saved?"

The answer is that it's convenient for the programmer to CALL subroutines with data in the Z-80 general purpose or other registers without having to worry about those

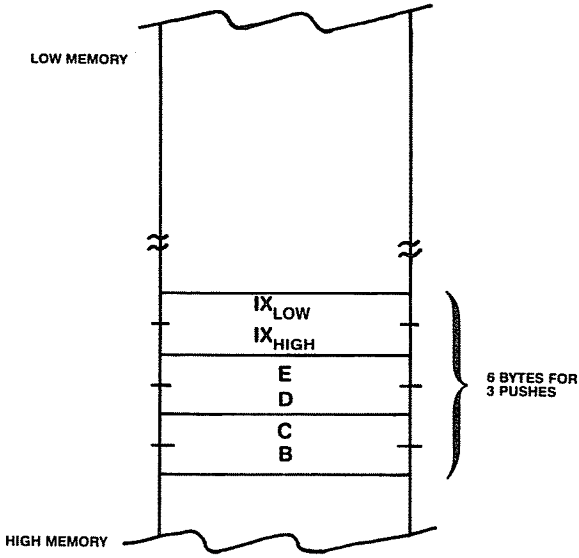


Figure 1-8. Stack Action for PUSH/POP

registers being used and the data being destroyed. As `B, DE,` and `IX` are used in the processing of `DECBIN,` their initial contents are saved in the stack at the beginning of the subroutine. Now the registers can be used for calculations and intermediate results during the processing of the subroutine, and their contents can be restored just prior to the RETurn. Of course any registers that are used to **pass back** the results of the subroutine would have new contents anyway and should not be saved in the stack.

The `LD IX,0` instruction clears the IX index register to 0. IX will hold the intermediate results of the conversion.

The assembly-language code from label DEC040 through `DJNZ DEC040,,` represents the main body of the subroutine. It is a loop (and the remarks are indented to indicate the loop condition). The number of times through the loop is determined by the number of ASCII digits to be processed. Five digits would involve five iterations, four would be four iterations, and so on.

The first order of business is to take the current contents of the IX register and to multiply it by 10. Although we could CALL another subroutine that performs a multiply, we choose to do it an alternative way by a "shift and add" technique. First we perform the ADD IX,IX, which adds the contents of the IX register to itself. Any number added to itself doubles the number, and that's the case here. The number*2 is then is PUSHed into the stack by the PUSH IX instruction. After the PUSH, the number*2 is in both IX and the stack. Two more ADDS change the contents of IX to number*4 and number*8, respectively. The next instruction, POP DE, pops the number*2 value from the stack and puts it into register pair DE. Now an ADD of IX and DE is done with the result going to IX. We are adding (number*8+number*2), which is the same as number*10. In effect, we've multiplied the contents of IX by 10.

The LD A,(HL) instruction is a LoaD instruction that loads the A register with the contents of a memory location. The HL register pair is used as a **register indirect** pointer to the memory location to be loaded. If, for example, HL contained 9000H, the contents of memory location 9000H would be loaded into the A register. The parentheses around the (HL) indicate that a **memory location**, rather than an **immediate** value, is involved in the instruction.

As we entered the subroutine with the HL register pointing to the first (leftmost) byte of the string, the first execution of `LD A,(HL)` would result in the first ASCII character of the string being loaded into the A register.

The ASCII character in A is now converted from hex 30 through 39 (ASCII "0" through "9") to a binary value of 0 through 9 by subtracting 30H. An ASCII "5", for example, would produce a binary value of 35H-30H=5.

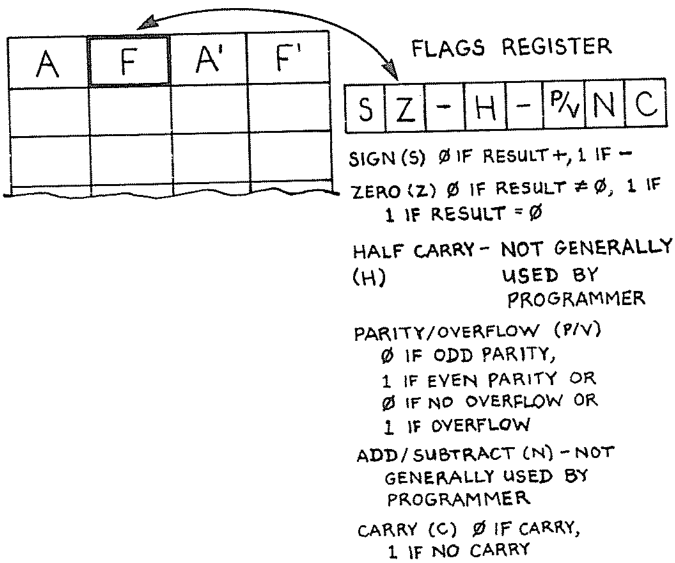
There is one problem here, however. If the ASCII character is not 30H through 39H, the result will be incorrect. We chose to check the validity of the ASCII string in the subroutine, so we'd best do that right now. If the result of (ASCII character-30H) is less than 0, then the ASCII character was less than 30H. If the result of (ASCII character-30H) is greater than 9, the ASCII character is greater than 39H.

When we performed the `SUB 30H`, the set of **flags** in the Z-80 was set according to the result of the subtract. If the result was 0, the Z flag was set; otherwise, it was reset (NZ). If the result was positive the Sign flag was reset (P); otherwise, it was set (M or minus). Similarly, the other flags (half-carry, carry, parity/overflow, add/subtract) were affected. The flags are not always affected by an instruction — LoaDs never affect flags, for instance. However, many instructions, especially adds and subtracts, do affect the flags according to the results of the instruction.

Hints and Kinks 1-6
Flags

Flags are a collection of individual bits grouped as the ''flags register.'' Flags are used to test results of instructions by conditional jumps. Some instructions (LDs, PUSH, POP, RET, others) never affect the flags. Other instructions (ADDs, SUBs, CP) always affect the flags while still other instructions only affect certain flags. See Appendix II.

Don't hesitate to use instructions that don't affect the flags before testing the flags with conditional jumps. Just be certain which instructions affect which flags – there are some surprises here!



We can test the result of the subtract for a minus condition by the `JP M,DEC070` instruction. If the result of (ASCII character-30H) was negative, the Sign flag will be set (M), and the instruction will jump on the M condition to label

DEC070. If the result was positive, the jump will be ignored, and the next instruction in sequence will be executed. If the result was negative, the jump to DEC070 will cause the subroutine to terminate prematurely based on the invalid ASCII character.

After the **conditional jump**, we make another test. The CP 10 instruction compares the result to 10. *A compare is essentially a subtract of an 8-bit value from the contents of the A register.* Like a subtract, the flags are affected and can be used for conditional branching. Unlike a subtract, the contents of A remain unchanged; the result is discarded. In this case the value to be subtracted is an **immediate value** of 10. If the Sign flag is set after the CP, the conditional jump JP P,DEC070 will cause a jump to location DEC070 to terminate the subroutine because of an invalid ASCII character. Otherwise the next instruction in sequence will be executed.

Assuming that the ASCII character was valid, we now have a binary value of 0-9 in the A register. The LD E,A copies that value from A into the E register. The next instruction, LD D,0, loads the D register with a 0 value. In fact, the DE register pair is now loaded with the value 0-9. If the D register were not cleared by loading 0, the DE register pair would contain garbage in the D register, and DE could not be used in further operations involving the binary value of 0-9.

Now the binary value in DE is added to the intermediate result in IX. If this is the first time through the loop, IX now contains the binary value of 0-9. If this is the second time through the loop, IX contains A1*10+A2, where A1 is the first converted ASCII value and A2 is the next. The third time produces A1*100+A2*10+A3, and so on.

The INC HL instruction adds one (INCrements) to the contents of the HL register and puts the results back into HL. (No flags are affected, by the way.) Initially, HL pointed to the leftmost character in the buffer. Each time through the loop, HL is adjusted to point to the next ASCII character so it may be processed in turn.

The `DJNZ DEC040` is a unique instruction. It operates as follows: The contents of the B register is decremented by one. If the result of the decrement is not zero (NZ), the instruction jumps back to the specified jump address, in this case DEC040. If the result of the decrement is zero, the next instruction in sequence is executed. The mnemonic stands for "Decrement and Jump if Not Zero."

We entered the DECBIN subroutine with a count in the B register of 1 to 5, representing the number of ASCII digits to be converted. Each time through the loop, the count in B is adjusted downward by one until zero is reached. As long as the contents of B **after the decrement** is not zero, the loop is reentered at DEC040.

Hints and Kinks 1–7
Loop Trace

Here's a ''trace'' of individual registers through part of the DECBIN routine. This type of trace can be done with paper and pencil to ''play computer'' and verify during ''desk checking'' that the program works as desired.

A	IX	DE	HL	B
31	Ø		BUF	3
1	1	1	BUF+1	2
	2			
	4			
	8	2		
	1Ø			
32				
2	12	2	BUF+2	1
	24			

Right now you're probably asking. ''Must assembly-language programming be this tedious?'' Well, yes... Look at the rewards. though—fast speed, compact code, the challenge..uh..maybe I'll go back to BASIC...

After the loop has been completed, IX contains the result of the conversion of the ASCII string. The `LD A,B` instruction loads the A register with the contents of B. If the count in B is 0, the loop has been completed for all ASCII digits. If B is other than 0, a premature jump was made to DEC070 because of an invalid ASCII digit. After B has been loaded to A, we execute an `OR A` instruction. This instruction is a commonly used instruction to test the contents of A. ORing any number with itself does not change the number. The `OR A` takes the contents of the specified register (A) and ORs it with the A register. When the specified register is A, this ORs A with itself. The important thing here, though, is that the flags are set on the result of the OR. The Z flag is set if the result in A is zero or reset if the result is non-0. Since A contains the previous count in B, the Z flag is set if B was zero (successful termination of loop) or reset if B was non-zero (invalid ASCII character). As the following PUSH, POP, and RET instructions **do not affect the flags**, the Z flag will remain set or reset on the return to the **calling program**.

As the result is in IX, and we specified the result in HL on exit, a transfer must be made. A `PUSH IX` followed by a `POP HL` pushes the contents of IX onto the stack and then immediately pops it back to the HL register pair. This is a common way to transfer the contents of one register pair to another as there is no LD from one register pair to another. The three POPs at the end of the subroutine restore the original contents of IX, DE, and BC. Note that the order of the contents is opposite from the way they were initially pushed onto the stack as the stack is a "last in, first out" operation.

The RET instruction pops the return address from the stack, puts it into the program counter, and causes a return back to the calling program at the next instruction after the CALL DECBIN. The HL register pair now contains the result of the conversion (0-65535), and the Z flag is set if the conversion was correct or reset if an invalid ASCII character was encountered.

The DECBIN subroutine is probably a typical TRS-80 subroutine in terms of complexity and size. If you have trouble with some of the concepts involved with DECBIN, review the appropriate material in *TRS-80 Assembly-Language Programming*, and continue to follow the examples closely in future chapters. We'll attempt to explain any subtleties as they come up.

Chapter Two
Assemblers and Assembling-EDTASM

Radio Shack has two assemblers available for the TRS-80, the cassette-based Editor/Assembler and the disk-based Disk Editor/Assembler. To make things easier, we'll refer to the former as "EDTASM" and the latter as "the Disk Assembler." In this chapter, we'll be primarily discussing EDTASM. Some of our discussion will also apply to the Disk Assembler. The two assemblers are similar in that they both assemble Z-80 assembly-language code, but they are different in the ways they load the code. In this book we'll be talking about using both EDTASM and the Disk Assembler, and to keep things straight, we'll note any differences in format or technique as we go along.

EDTASM is more of a "single-user" assembler for short programs, while the Disk Assembler is normally used for more advanced work and larger programs. Chapter 13 has a large program that has been assembled by EDTASM, and we'll use that program for frequent examples in this and other chapters.

A Look At EDTASM

For openers, let's take a look at EDTASM. Some of this material might be familiar to you from using EDTASM or from reading *TRS-80 Assembly-Language Programming*. The operations in EDTASM are more straightforward than some of the advanced features of the Disk Assembler, so we'll start with the operation of EDTASM and then use that as a base for discussing the Disk Assembler.

As we know from Chapter 1, EDTASM includes both an Editor and an Assembler. The Editor has commands and subcommands similar to the Editor in Level II BASIC. Using these commands, an assembly-language source file can be initialized, modified and written out to cassette tape for subsequent assembly by the Assembler portion of EDTASM. (As an alternate approach, the source lines can be assembled directly from the buffer in memory to check for a valid assembly before writing the file out to cassette.) We won't go into the editor commands and subcommands as they're covered both in the EDTASM *User Instruction Manual* and in our previous book.

Hints and Kinks 2-1
Typical Edit Sequence

Here's a typical edit sequence to create an assembly-language source file with EDTASM:

1. Load EDTASM.
2. *I100,10 starts lines at 100 with increments of 10.
3. Enter source lines. Each line is terminated by an ENTER. Use tabs between labels, op-codes, operands, and comments.
4. Hit BREAK. This brings you back to the Editor command mode.
5. W NAME. Write file to cassette with name NAME.

See the Editor/Assembler User Instruction Manual for directions on Editor commands and subcommands.

What we **are** going to cover in this section are the subtleties of EDTASM source language **syntax** and **pseudo-ops**. Syntax (please, no bad jokes) refers to the rules of structuring assembly-language lines, while pseudo-ops are commands to the assembler in the **op-code** field of the assembly-language line. For clarity, we'll use examples from a program assembled by EDTASM in the last section of this book.

Source Line Syntax

Figure 2-1 shows a typical section of code for the MORG program. The syntax is fairly straightforward. The source lines are **free format** in that there are no specified columns for labels, op-codes, operands, and comments. You can write source code in any convenient format as long as there's a space between each field, as long as an op-code does not start in column 1, and as long as a semicolon precedes a comment.

COMMENT LINE STARTED BY SEMICOLON

SEMICOLON PRECEDES COMMENT FIELD

```
                 01500 ;
                 01510 ;****************DEFINE MESSAGE N ROUTINE****************
                 01520 ;
80EB 2AE985      01530 DEFINE  LD      HL,(CURCUR)      ;GET CURRENT CURSOR
80EE 22EB85      01540         LD      (LSTCUR),HL      ;SAVE
80F1 CDFB83      01550 DEF005  CALL    CLRCOM           ;CLEAR COMMUNICATIONS AREA
80F4 21F586      01560         LD      HL,MSG5          ;DEFINE MESSAGE
80F7 01403F      01570         LD      BC,LINE13        ;LINE 13
80FA CD6D83      01580         CALL    DSPMES           ;DISPLAY DEFINE MESSAGE
80FD 0601        01590         LD      B,1              ;1 CHARACTER
80FF CD0D84      01600         CALL    INPUTS           ;GET CHARACTER
8102 C27F81      01610         JP      NZ,DEF050        ;GO IF GT 1
8105 0601        01620         LD      B,1              ;1 CHARACTER
8107 CD9085      01630         CALL    DECBIN           ;CONVERT TO BINARY
810A C27F81      01640         JP      NZ,DEF050        ;GO IF ERROR
810D 7D          01650         LD      A,L              ;MSG # NOW IN A
810E 21C987      01660         LD      HL,MSG11         ;INPUT MESSAGE
8111 01803F      01670         LD      BC,LINE14        ;LINE 14
8114 CD6D83      01680         CALL    DSPMES           ;DISPLAY MESSAGE
8117 2AEB85      01690         LD      HL,(LSTCUR)      ;GET OLD CURSOR
811A 22E985      01700         LD      (CURCUR),HL      ;RESTORE
811D F5          01710         PUSH    AF               ;SAVE MESSAGE #
811E CDC782      01720         CALL    FNDMSG           ;GET ADDRESS OF MESSAGE
8121 202A        01730         JR      NZ,DEF035        ;GO IF NO CURRENT MSG FOR #
                 01740 ;CURRENT MSG FOR # IN MBUF - MUST DELETE
8123 E5          01750         PUSH    HL               ;SAVE START
```

TAB (RIGHT ARROW) USED TO SPACE TO PRESET FIELDS (OPTIONAL)

Figure 2-1. Assembly Source Code

Because we like neat listings, we've used the **right arrow** tab function in this and other codes to tab to the next tab position. You might also notice that when loops occur, we've indented the loop in the comments column to make them easier to follow.

A small sermon here from one who has learned the hard way: you can't use too many comments in a listing. Invariably I find myself looking at assembly-language code months later and wondering why I did a particular thing. Comments help!

The labels for assembly-language lines are optional. In most of the programs here, we've used the following general rules for labels:

1. The first label of a subroutine or main section of code is a descriptive name for the code, such as SCROLL or CLRCOM.
2. The labels in that particular section of code use the first three letters of the descriptive name plus three digits.
3. The digits in the labels are generally in ascending order. For example, `INP020` follows `INP010`.

Of course, you can use any scheme you wish for labels. An approach like the one above does make it easier to locate code and ultimately makes coding easier.

The op-codes in the source code follow **standard Zilog mnemonics**. These are the mnemonics that Zilog, the designer of the Z-80, defined for each instruction.

The operands also follow the Zilog format for arguments. This format uses parentheses to indicate a memory address and no parentheses to indicate an immediate value; it also fixes the number of arguments for any particular instruction. Let's illustrate that first point about parentheses: If a source line appears such as `LD HL,(3880H)`, the parentheses indicate that the contents of memory location 3880H (and 3801H) will be loaded into the HL register pair. But if a source line, (such as `LD HL,3880H`) appears, the lack of parentheses indicates that the **value** 3880H will be loaded into the HL register pair.

Hints and Kinks 2-2
Flag Mnemonics

The mnemonics used in conditional jumps refer to flag conditions. Typical instructions might be

```
JP NZ,LOCN    ;JUMP IF NOT ZERO
JR C,LOOP     ;JUMP IF CARRY
```

The mnemonics and associated flag settings are:

MNEMONIC	FLAG SETTING
NZ (non-zero)	Z=0
Z (zero)	Z=1
NC (no carry)	C=0
C (carry)	C=1
PO (parity odd)	P/V=0
PE (parity even)	P/V=1
P (sign positive)	S=0
M (sign negative)	S=1

There are no conditional jumps for H (half-carry flag) or N (add/subtract flag). These are used internally by the Z-80 in instruction execution.

The most frequent conditional jump is on the Z flag, next the carry, next sign, next the ''overflow'' condition of P/V, and next the ''Parity'' condition of P/V.

Similarly, LD A,(HL) means that HL is used as a memory indirect pointer register and that the contents of the location pointed to by HL will be loaded into A. LD A,B, of course, means the contents of B will be loaded into A. Note that the format for Z-80 instructions in the operand field is always **destination, source**. The source operand, whether memory or register, is always last, while the result, or destination, is always first.

Constants

Now a word about numeric **constants**. As you've probably noticed, we've been using H right along as a suffix for a hexadecimal constant. Any hex value must have an H as a suffix to indicate the data is hexadecimal. If the first digit of the hexadecimal constant is A through F, legitimate hex digits, then a 0 must be added before the hex digit. It's easy to see why this is so since the Assembler must be able to differentiate between constants and labels.

If a constant has no suffix, it's assumed to be a decimal constant. A suffix of D signifies a decimal constant (so there's no point in using it!), and a suffix of 0 signifies an octal constant (so there's probably no need for this suffix either, since you'll probably never require octal constants).

If a character is bracketed by single quotation marks, it's interpreted as an ASCII character. To load the B register with an ASCII A, for example, we'd have `LD B,'A';LOAD A INTO B REGISTER`.

We'll discuss **expressions** involving constants and symbols after we look into the use of **pseudo-operations**.

Pseudo-Operations

Most assembly-language source lines are **generative** types of lines. A mnemonic representing a Z-80 op-code generates the corresponding machine code for that instruction at that point in the assembly. However, there are a number of commands to the assembler, called pseudo-operations or "pseudo-ops," that don't generate instructions. Instead, they inform the Assembler of certain actions to be taken, or they create data values.

The first of these is the **Origin**, or **ORG**, pseudo-op. The Origin informs the Assembler that the following code is to be assembled for a particular memory location. For an example of this, look at Figure 2-2, where `MORG` has been assembled to run at 8000H. When the Assembler encounters the ORG statement, it will set an internal

assembly location counter to the value of the operand in the ORG statement. This assembly location counter will be adjusted by the length of each instruction as an instruction is assembled. Unless the code is **relocatable**, code produced for one Origin will not run anywhere else in memory, since there are addresses in many instructions which are **absolute** addresses. (We'll discuss **relocatability** shortly.)

SETS ASSEMBLY LOCATION COUNTER TO 8000H

```
8000          00100          ORG     8000H
              00110 ;MORG-0820
              00120 ;*************MORSE CODE GENERATOR PROGRAM*************
              00130 ;
              00140 ;
              00150 ;*******************SYSTEM EQUATES********************
              00160 ;
3C00          00170 SCREEN   EQU     3C00H          ;START OF VIDEO DISPLAY
3C40          00180 LINE1    EQU     SCREEN+64      ;SECOND LINE
3EC0          00190 LINE11   EQU     SCREEN+704     ;TWELFTH LINE
3F00          00200 LINE12   EQU     SCREEN+768     ;THIRTEENTH LINE
3F40          00210 LINE13   EQU     SCREEN+832     ;FOUTEENTH LINE
3F80          00220 LINE14   EQU     SCREEN+896     ;FIFTEENTH LINE
3FC0          00230 LINE15   EQU     SCREEN+960     ;SIXTEENTH LINE
0002          00240 ENTER    EQU     2              ;ENTER CHARACTER
0001          00250 CLEAR    EQU     1              ;CLEAR CHARACTER
0064          00260 DBDEL    EQU     100            ;DEBOUNCE DELAY IN MS
0065          00270 DBDELP   EQU     DBDEL+1        ;DEBOUNCE DELAY+1 MS
000A          00280 MLDEL    EQU     10             ;MAIN LOOP DELAY IN 1/10 MS
03C0          00290 SPEEDF   EQU     960            ;FINAGLE FACTOR FOR SPEED
              00300 ;
              00310 ;******************MORSE EXECUTIVE*******************
              00320 ;
8000 F3       00330 START    DI                     ;DISABLE INTERRUPTS
8001 315994   00340          LD      SP,TOPS        ;SET STACK POINTER
8004 11EA88   00350          LD      DE,MBUF        ;MESSAGE BUFFER ADDRESS
8007 010B0A   00360          LD      BC,2571        ;2571 BYTES
```

FIRST INSTRUCTION ASSEMBLED AT LOCATION 8000H

Figure 2-2. ORG Use

The END pseudo-op marks the end of the source file and is self-descriptive. If the END has an operand, use it as the starting point in the program when the program is loaded as in `END START`.

There are three pseudo-ops that generate data in EDTASM. The first of these is **DEFB**, or DEFine Byte, which generates a single 8-bit value. The next is **DEFW**, or DEFine Word, which generates a 16-bit value. The last is **DEFM**, or

DEFine Message, which generates a string of ASCII characters that usually represents a message used in the program. Each of these three pseudo-ops can have a label, if desired. Examples of each are shown in Figure 2-3.

```
                ;*******************WORKING STORAGE**********************
                ;
85DC 0000       TMP1    DEFW    0               ;TEMPORARY STORAGE
85DE 00         PRINTF  DEFB    0               ;PRINTER FLAG:0=OFF,1=ON
85DF 00         LPFTF   DEFB    0               ;LP 1ST TIME FLAG:0=1ST TIME
85E0 00         CHARCT  DEFB    0               ;LP CHARACTER COUNTER
85E1 D204       SEED    DEFW    1234            ;DEFAULT SEED
85E3 2E16               DEFW    5678
85E5 9001       DOTO    DEFW    400             ;DOT ON TIME (3 WPM DEFAULT)
85E7 B004       DASHO   DEFW    1200            ;DASH ON TIME (3 WPM DEFLT)
85E9 003C       CURCUR  DEFW    3C00H           ;CURRENT CURSOR POSITION
85EB 003C       LSTCUR  DEFW    3C00H           ;LAST CURSOR POSITION
85ED 0000       TSLC    DEFW    0               ;TIME IN MS SINCE LAST CHAR
85EF 20         LASTR   DEFB    ' '             ;LAST RANDOM CHARACTER SENT
85F0 F592       IBUFL   DEFW    IBUF            ;POINTER TO LAST IBUF SLOT
85F2 F592       IBUFN   DEFW    IBUF            ;POINTER TO NEXT IBUF SLOT
                ;
                ;*******************SYSTEM MESSAGES********************
                ;
85F4 20         MSG1    DEFM    '                               ***MORG***
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 2A 2A 2A 4D 4F
     52 47 2A 2A 2A 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20
8634 43                 DEFM    'CHAR=SEND CHARACTER  SHIFT 0-9=SEND MSG N'
     48 41 52 3D 53 45 4E 44
     20 43 48 41 52 41 43 54
     45 52 20 20 53 48 49 46
     54 20 30 2D 39 3D 53 45
     4E 44 20 4D 53 47 20 4E

865D 20                 DEFM    '  SHIFT R=SEND RANDOM  SHIFT D=DEFINE MS'
     20 53 48 49 46 54 20 52
     3D 53 45 4E 44 20 52 41
     4E 44 4F 4D 20 20 53 48
     49 46 54 20 44 3D 44 45
     46 49 4E 45 20 4D 53
8685 47                 DEFM    'G   SHIFT S=DEFINE SPEED  SHIFT P,N=PRINT'
     20 20 20 53 48 49 46 54
     20 53 3D 44 45 46 49 4E
     45 20 53 50 45 45 44 20
     20 53 48 49 46 54 20 50
     2C 4E 3D 50 52 49 4E 54

86AE 20                 DEFM    ' OR NO'
     4F 52 20 4E 4F
```

Figure 2-3. DEFB, DEFW, DEFM Use

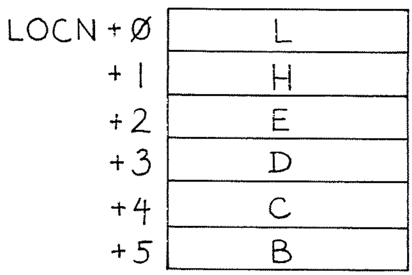
The DEFW generates 16-bits of data in standard Z-80 **word** format. In this format, the **least significant byte** of the 16 bits is stored in the first byte, and the **most significant byte** of the 16 bits is stored in the second byte. If you look at the machine code produced in Figure 2-3, you'll see the hexadecimal data generated in `CURCUR DEFW 3C00H`, for example, is arranged as `003C`.

Hints and Kinks 2-3
Sixteen-Bit Data Format

Many programmers get confused over 16-bit data format. When a 16-bit piece of data is stored in memory, it is <u>always</u> stored least significant byte followed by most significant byte. The code

```
LD          (LOCN),HL       ;STORE HL
LD          (LOCN+2),DE     ;STORE DE
LD          (LOCN+4),BC     ;STORE BC
```

would store data in the LOCN area as follows:



Data in the stack (say, a PUSH HL) is stored the same way.

When data is retrieved (LD DE,(LOCN+2) or POP BC) it's put back into the registers in identical fashion. (Whew! What if the designers had chosen to retrieve it in opposite fashion. . . . Thank goodness for cool heads in Silicon Valley . . .).

The DEFM simply generates a one-byte ASCII character for every character in the string. The string is started and terminated by a single quotation mark (').

The **DEFS** (DEFine Storage) pseudo-op reserves a number of bytes at the current assembly location. If, for example, you want to define a table that would be filled with values during program execution, you might have the following code:

```
TABLE DEFS 100      ;100-BYTE TABLE
```

The assembler would then increase the assembly location counter by 100, bypassing 100 bytes. Note that the machine code bytes produced for a DEFS consist of garbage, or unknown, data. A DEFS does not fill the vacant space with zeroes, all ones, or anything; it just leaves whatever is there to begin with.

The last pseudo-op is EQU, or EQUate. To understand this pseudo-op, you need to understand the processing that the Assembler goes through. The Assembler makes two (or more) **passes** through the source file. After the first pass, it has assembled a **symbol table** of all labels with a corresponding numeric value for each symbol. (This is the information displayed at the end of the program listing.)

In most cases, each symbol has a value that represents the assembler location counter value, which is essentially the location at which the instruction or data for that symbol will reside. The EQU pseudo-op, however, can force the Assembler to equate a label either with a numeric value or another label. This is the case for `CDTAB` as shown in Figure 2-4. `CDTAB` is equated to `CTAB`. In other words, the value associated with CTAB — the location of 8812H — will also be used for symbol `CDTAB`. In this case, we made the association because the CDTAB contained the same data as the first 44 locations of CTAB.

```
                10080 ;*******************CTAB CHARACTER TABLE******************
                10090 ;*   TABLE OF CHARACTERS TO BE SENT IN RANDOM MODE.      *
                10100 ;*   DISTRIBUTION DOES NOT CORRESPOND TO THAT IN NOR-    *
                10110 ;*   MAL TEXT. SPACE CHARACTER NOMINALLY EVERY 5TH       *
                10120 ;*   CHARACTER.                                          *
                10130 ;
8812 41         10140 CTAB     DEFM    'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.,?/- =;'
     42 43 44 45 46 47 48 49
     4A 4B 4C 4D 4E 4F 50 51
     52 53 54 55 56 57 58 59  <---------- THESE BYTES ARE CDTAB!
     5A 30 31 32 33 34 35 36
     37 38 39 2E 2C 3F 2F 2D
     20 3D 3B
883E 30         10150          DEFM    '0123456789.,?/- =;ABCDEFGHIJKLMNOPQRSTUVWXYZ'
     31 32 33 34 35 36 37 38
     39 2E 2C 3F 2F 2D 20 3D
     3B 41 42 43 44 45 46 47
     48 49 4A 4B 4C 4D 4E 4F
     50 51 52 53 54 55 56 57
     58 59 5A
886A 20         10160          DEFM    '                 '
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20
887C 41         10170          DEFM    'ABCDEFGHIJKLMNOPRSTUVY'
     42 43 44 45 46 47 48 49
     4A 4B 4C 4D 4E 4F 50 52
     53 54 55 56 59
```

```
          ;
          ;*******************CDTAB CODE TABLE*********************
          ;*    TABLE OF VALID ASCII CHARACTERS TO BE TRANSMITTED. *
          ;*    INDEX TO CHARACTER USED TO OBTAIN TIMING CODES     *
          ;*    FROM TTAB.                                         *
          ;
8812      CDTAB    EQU     CTAB          ;SAME DATA
002C      CDTABS   EQU     44            ;SIZE OF DATA
          ;
```

Figure 2-4. EQU Use Example 1

As figure 2-4 also shows, you can use the EQU pseudo-op for CDTABS. CDTABS is equated to 44. Every time CDTABS is referenced, as in `LD B,CDTABS`, the value loaded for CDTABS will be 44. `LD B,CDTABS` would therefore become equivalent to `LD B,44`. Equates are used in this fashion so an easily-remembered symbolic name can be given to constants such as size of tables, addresses of I/O devices, etc.

Another use of EQU is shown in Figure 2-5. Here `MBUF` (Message Buffer), has been equated to `$`. `$` is a special symbol that represents the current value of the assembler location counter. After the MBUF equate, `MBUF` would be stored in the symbol table as the value 88EAH. The next line, `ENDM EQU $+2571`, equates the label `ENDM` to `$+2571`. Since the assembler location counter did not change from the previous equate (no instructions or data were generated), `ENDM` is equated to `88EAH+2571` or `92F5H`. This little trick is synonymous to:

```
MBUF EQU  $
     DEFS 2571
ENDM EQU  $
```

and is quite commonly used.

```
          ;********************MESSAGE BUFFER**********************
          ;*    ARBITRARILY SET AT 2560 BYTES (256 BYTES PER MSG   *
          ;*    PLUS MSG# PLUS 1 TERMINATOR).                      *
          ;
88EA      MBUF     EQU     $
92F5      ENDM     EQU     $+2571
          ;
```

Figure 2-5. EQU Use Example 2

Operators and Expressions

EDTASM allows a limited number of operations involving addition, subtraction, ANDing, and shifting. These operations are performed by the **operators** +, −, &, and <. Addition of constants are shown in Figure 2-6, along with one subtraction example for the size of FTAB. Expressions may also include a mixture of numeric data and labels in any combination.

```
               01130 ;
               01140 ; FUNCTION TABLE
               01150 ;
80BE C4        01160 FTAB    DEFB    'D'+80H        ;DEFINE MESSAGE
80BF D3        01170         DEFB    'S'+80H        ;DEFINE SPEED
80C0 D2        01180         DEFB    'R'+80H        ;TRANSMIT RANDOM
80C1 B0        01190         DEFB    '0'+80H        ;TRANSMIT MESSAGE 0
80C2 B1        01200         DEFB    '1'+80H        ;                 1
80C3 B2        01210         DEFB    '2'+80H        ;                 2
80C4 B3        01220         DEFB    '3'+80H        ;                 3
80C5 B4        01230         DEFB    '4'+80H        ;                 4
80C6 B5        01240         DEFB    '5'+80H        ;                 5
80C7 B6        01250         DEFB    '6'+80H        ;                 6
80C8 B7        01260         DEFB    '7'+80H        ;                 7
80C9 B8        01270         DEFB    '8'+80H        ;                 8
80CA B9        01280         DEFB    '9'+80H        ;                 9
80CB D0        01290         DEFB    'P'+80H        ;SET PRINT
80CC CE        01300         DEFB    'N'+80H        ;RESET PRINT
000F           01310 FTABS   EQU     $-FTAB         ;SIZE OF FUNCTION TABLE
0000           01320         END
00000 TOTAL ERRORS
```

Figure 2-6. Operators and Expressions

Using EDTASM to Edit and Assemble Programs

EDTASM is geared to editing and assembling one **module** of assembly-language code. After you have specified, flowcharted or analyzed, and coded a program on paper, use the Editor in EDTASM to enter the complete program code. Although you have probably divided the program into a number of different functional modules, include the modules as one large, whole assembly-language source file.

Hints and Kinks 2–4

Typical Assembly Sequence

With the source file in memory (either from cassette or from keyboard entry), a typical assembly sequence might be:

1. *A/NO/WE Assemble to screen with no object; wait on errors.
2. Go back to Editor to correct any errors. When there are no errors, go to 3.
3. *A/NO/LP Assemble to line printer with no object.
4. Take the listing and do a comprehensive desk check (Coffee or other stimulants allowed.)
5. Repeat steps 1 through 4 as often as required.
6. *A NAME/LP Assemble final assembly to line printer, object to cassette with name "NAME".
7. Debug.

The resulting source file on cassette is then assembled as one large assembly. MORG, the Morse Code Program of Chapter 13, is a large program that approaches the limits of memory capacity. Because all of the source lines must be held in memory at one time, in addition to the symbol table, there's a practical limit to the size of the program that can be assembled under EDTASM. This limit depends upon the memory size of the system, number of characters in the source file, and number of labels used. We'll see in the next chapter how you can use the Disk Assembler to overcome some of the limitations by using a different approach to assembling and loading programs, but let's first get a clear understanding of EDTASM operation.

As we can see from looking at MORG, the entire program is assembled in one swell foop. The ORG statement specifies the starting address of the program, and all instructions are referenced to the assembler location counter. The assembler location counter is initially set by ORG and incremented as each instruction or pseudo-op is generated. EDTASM produces **object code** for the program as a cassette file. The object code is very similar to the machine code shown on the listing, but contains some additional data to hold the file name, origin, number of bytes per record, checksum, etc.

You can load the resulting object code by a SYSTEM command while in Level II BASIC monitor mode. The SYSTEM command enables you to load the object file created under EDTASM and then to transfer control to the starting address of the program (specified by the operand used with the END pseudo-op).

The object code produced by EDTASM generally can be loaded and executed only at one point in memory, the area at which it was ORiGined. If you load the object code at another area (by some devious means), it won't execute properly. Let's see why this is so.

Relocatability

Instructions in the Z-80 instruction set are generally **relocatable** or **non-relocatable**. Relocatable instructions will execute properly anywhere in memory; non-relocatable instructions contain absolute references and will execute only in the area for which they were assembled.

Hints and Kinks 2-5

Why All the Interest in Relocatability?

Much Z-80 literature talks about the relocatable instructions or code and touts the advantages. Are there many advantages in the relocatable Z-80 instructions?

Not really. Any large piece of code is probably not relocatable because it will have to contain JPs and CALLs – unless some pretty clever coding is done. About the only reason for relocatability on the TRS-80 is to allow short code segments to be embedded in BASIC programs. This merges BASIC and assembly-language code and allows the programmer to use assembly-language code to speed up his time <u>critical</u> processing.

Occasionally someone will try to take an existing program that is available in machine language only (no listing) and attempt to relocate it to run elsewhere. This is possible, but <u>very</u> tedious. Try it only on a favorable bio-rhythm day!

Figure 2-7 shows typical code for the MORG program produced by EDTASM. Let's take a look at the instructions to see which are relocatable, which are not, and why.

NOT RELOCATABLE

```
8197 CDFB83    02340 SPE005  CALL    CLRCOM        ;CLEAR COMMUNICATION AREA
819A 21CB86    02350         LD      HL,MSG4       ;SPEED MESSAGE
819D 01403F    02360         LD      BC,LINE13     ;LINE 13
81A0 CD6D83    02370         CALL    DSPMES        ;DISPLAY SPEED MESSAGE
81A3 0602      02380         LD      B,2           ;2 CHARS
81A5 CD0D84    02390         CALL    INPUTS        ;GET CHARACTER STRING
81A8 2035      02400         JR      NZ,SPE020     ;GO IF GT 2 CHARACTERS
81AA CD9085    02410         CALL    DECBIN        ;CONVERT TO BINARY
81AD 2030      02420         JR      NZ,SPE020     ;GO IF ERROR
81AF 7D        02430         LD      A,L           ;GET SPEED 0-99
81B0 FE03      02440         CP      3             ;TEST FOR 3 WPM
81B2 FADF81    02450         JP      M,SPE020      ;GO IF LT 3 WPM
81B5 FE3D      02460         CP      61            ;TEST FOR 60 WPM
81B7 F2DF81    02470         JP      P,SPE020      ;GO IF GT 60 WPM
81BA 21C003    02480         LD      HL,SPEEDF     ;1200/WPM=DOTO TIME
81BD 4F        02490         LD      C,A           ;WPM LS BYTE
81BE 0600      02500         LD      B,0           ;NOW IN BC
81C0 11FFFF    02510         LD      DE,-1         ;QUOTIENT
```

RELOCATABLE

Figure 2-7. Relocatable Code

The CP 61 found 12 lines after SPE005 has machine-language code of FE3D. By reference to Appendix II, we can see that the first byte of this instruction is the op-code and that the second byte is the data value of 61 (3DH). This instruction would assemble exactly the same way at any spot in memory, because the op-code is fixed and the data value would have to be the same.

The instruction CALL CLRCOM, however, is a different beastie. CLRCOM is a subroutine that appears somewhat later in the program. If you'll look again at the formats in Appendix II, you'll see that a CALL consists of three bytes. The first of these is an op-code of CDH. The second and third bytes are an **absolute memory address** that hold the call address with the bytes in reverse order. The address in this case is that of CLRCOM, which is at 83FBH. Would this instruction execute properly if MORG were moved to another area of memory? Obviously not, as the CLRCOM subroutine would also be relocated, and its address would be other than 83FBH. A similar situation would exist for instructions such as JP SPE020, LD HL,MSG4, and others that contain absolute addresses.

You might be wondering if it's possible to write programs in pure **relocatable code**. As a matter of fact, you can write **relocatable code** in relatively short routines that contain no absolute addresses and use JR type jumps for conditional and unconditional jumps. We'll explore some of the techniques in Chapter 5. Longer programs require you to do something such as reassembling your program by EDTASM with a new Origin. Another way is to use the facilities of the Disk Assembler. We'll look into this last method in the next chapter.

Chapter Three
Assemblers and Assembling — Disk Assembler

In this chapter we'll talk about the Radio Shack Disk Assembler. You should use this material only as a supplement to the *Disk Assembler User Instruction Manual,* since an adequate discussion of the Assembler would require more than just a single chapter. A good example for some of the techniques associated with the Assembler is included in Chapter 14 where we illustrate an entire program made up of modules assembled and loaded by the Disk Assembler and Loader. We'll be using further examples in the text of Disk Assembler code and EDTASM code in the remaining chapters of this book.

Comparison Between EDTASM and the Disk Assembler

Although EDTASM is cassette-based and the Disk Assembler must use files from disk, the two systems share more similarities than differences. Both edit assembly-language source files and have approximately the same line-oriented type of Editor with very similar commands and subcommands. Both assemble the source files and look for identical formats as far as Z-80 mnemonics and instruction syntax. Both use pseudo-ops for origin; pseudo-ops for definition of bytes, words, text, and storage; and a pseudo-op for equates. Many EDTASM source files can be converted to Disk Assembler use quite easily with a minimum of editing changes, primarily by adding a colon after each label and changing some of the data definitions.

Hints and Kinks 3-1
Format Differences between
EDTASM and Disk Assembler

There are several format and pseudo-op differences between the two assemblers. The most obvious is the use of a colon after labels on most source lines (not on an EQU).

The pseudo-ops for data definition are also somewhat different. Although DEFB, DEFW, DEFS and DEFM may be used, the Disk Assembler also uses DB in place of DEFB, DW in place of DEFW, DS in place of DEFS, and DC in place of DEFM. DB 'string' can also be used as a DEFS.

Multiple arguments can be used for the data definition pseudo-ops as in "DEFB 2,3,5".

The Disk Assembler also requires at least a 95 character wide line for assembly listing.

Can you convert EDTASM files on cassette to Disk Assembler source files? Conceivably, a short assembly-language conversion program might be used that would convert the format differences above (and others). It would make a nice exercise . . .

The simple uses of the Disk Assembler look very similar to EDTASM. However, when all the capabilities of the Disk Assembler are utilized, there are some major advantages to the Disk Assembler not found in EDTASM. In approximate order of importance, these are:

1. The Disk Assembler uses disk files for source, object, and listing.
2. The Disk Assembler produces object-file output in the form of **relocatable object modules** which are loaded by a special Loader.
3. The Disk Assembler has a macro capability to generate in-line macro code.
4. The Disk Assembler has a number of pseudo-ops relating to program **sections** and conditional assembly.

5. The Disk Assembler has a number of pseudo-ops relating to listing format.

We'll discuss each of these points in the following text, and give a number of examples to illustrate each concept.

Disk Files for the Disk Assembler

The Disk Assembler operates on assembly-language source files on disk created by the Editor portion of the software package. These source files are processed by the Assembler and the resulting object files are output to disk. The object files can be **loaded** into memory by the Loader portion of the package, and a **CMD** (CoMmanD) file can then be dumped to disk.

The CMD file, is a **core-image** file of the machine-language code that makes up the complete assembly-language program. It can be loaded and executed by simply typing in the name of the CMD file after the TRSDOS "DOS READY" prompt.

In addition to the source, object, and CMD files created by the Editor and Assembler, a **listing file** can be created for a hardcopy listing. The listing file can then be "PRINTed" on the TRS-80 system line printer.

Using the Editor for Source Files

The assembly-language source file is similar to other files, but there are significant differences. The assembly-language source file created by the Editor is almost an ASCII file. However, there is some non-ASCII coding for the line numbers at the start of each line. The Editor commands are similar to the Editor commands in Level II BASIC or EDTASM, with some differences. The symbol "." is used to refer to the current line and the symbol "*" refers to the last line of the **edit buffer**, just as in the other editors. **The first line of the buffer, however, is signified by the symbol,** "↑" (up arrow), rather than "#".

Ranges of lines may be specified, just as in the other editors. For example, to delete lines 200 through 300, give the Editor command D200:300. A new type of range specification, however, deletes a specified line through the next *n* lines. D200!3, for instance, would delete line 200 and the next 3 lines following line 200.

The Editor commands — I (Insert), D (Delete), R (Replace), P (Print), and N (Number) — all operate the same as in the other editors. To edit a specific line, though, the command A (Alter), rather than E is given. See Table 3-1.

Table 3-1
Disk Editor/Assembler Edit Commands

Command	Action	Typical Example*
A	Alter	A100 starts edit (Alter) of line 100
B	Begin	B/2 moves to beginning of page 2
D	Delete line	D1ØØ:2ØØ deletes lines 100-200
E	Exit Editor	E NAME/MAC exits Editor, writes file NAME
F	Find	F1ØØ:2ØØ$TWINE$ finds string "TWINE"
I	Insert	I1ØØ,1Ø inserts lines from 100, increment 10
K	Kill	K/2 kills page mark at page 2
L	List	L1ØØ:* prints lines from 100 through end
M	Mark	M1ØØ inserts page mark after line 100
N	Number	N1ØØ,1Ø renumbers lines from 100, increment of 10
P	Print	P1ØØ:2ØØ displays lines 100-200
Q	Quit	Q quits Editor without disk write
R	Replace	R1ØØ replaces line 100
S	Substitute	S1ØØ:2ØØ$MOD II$MOD III$ substitutes "MOD III"
W	Write to disk	W NAME/MAC writes text as disk file "NAME/MAC"
X	Extend	X1ØØ:2ØØ enters extend mode for lines 100-200

*Consult Disk Editor/Assembler manual for the many variations.
$ = BREAK

Editor **subcommands**, commands that operate within a specified line, are very similar to the other editors. You'll discover some new subcommands in this mode, though. See Table 3-2 for a complete list.

Table 3-2
Disk Editor/Assembler Edit Subcommands

Subcommand	Action	"Usual" Form *
A	Prints remainder, enters changes, concludes editing	A
B(lanks)	Inserts spaces	B or iB
C(hange)	Replaces characters	C *<ch>* or iC *<text>*
D(elete)	Deletes characters	D or iD
E(nter)	Enters changes, concludes editing	E
F(ind)	Finds text	F$*<text>*$
G	Inserts characters	G*<ch>*
H(acks)	Delete remainder and enter Insert mode	H or H*<text>*$
I(nsert)	Inserts text	I or I*<text>*$
K(ill)	Kills characters up to *<ch>*	K*<ch>*
L(ine)	Prints line, positions cursor to beginning	L
N(ot)	Restores line, moves to next line	N
O(bliterate)	Deletes all characters up to *<text>*	O*<text>*$
P(osition)	Prints remainder of line, cursor static	P
Q(uit)	Exits edit mode and restores original line	Q
R(eplace)	Replaces characters with *<text>*	iR*<text>*$
S(earch)	Finds character *<ch>*	S*<ch>*
T(runcate)	Deletes remainder of line and concludes edit	T
W(ord)	Moves the cursor to beginning of next word	W
(e)X(tend)	Print remainder of line, go into Insert mode	X
Z(aps)	Deletes word	Z
←	Delete character	←
ENTER	Prints remainder, enters changes, concludes edit	ENTER
SHIFT ←	Restores original line, repositions cursor to beginning	SHIFT ←
SPACE	Spaces over character	SPACE
→	Moves cursor to line end	→

i = number
$ = BREAK or ENTER
<ch> = character
<text> = text string
* Consult Disk Editor/Assembler manual for the many variations

Here's the procedure for creating a new source file:

1. Load the Editor by entering EDIT after the TRSDOS READY prompt. The Editor will be loaded as a CMD file and will start execution.
2. The Editor will ask for the file name by displaying FILE:.
3. Enter a TRSDOS file name with optional extension, password, and drive number; **follow that with the BREAK key**. Since the Assembler works with the extension /MAC, this would be the best choice for the extension. You can also disregard passwords, unless you're really paranoid about your source files! Some examples of source file names might be MTALP/MAC:1 or HTPM/MAC.
4. You'll now be at the command level, as indicated by an asterisk (*), and can start creation of source lines by insert commands such as I100.

To modify an existing source file, follow a similar sequence to the one above, only type an ENTER rather than a BREAK after the file name.

During the source file editing, you can P(rint) the assembly source lines on the screen, or get a hardcopy on your system line printer by the L(ist) command. During the editing session, you will notice some **disk activity** taking place. The Editor will be reading in **pages** for the source file as required.

When you have the source file the way you'd like it, you can write a new source file to the disk by giving the E(nd) command and the name initially used. If this is modification of an existing file, the form of the command is E NAME/MAC, where the NAME of the source file must be different than the one originally read in.

This last point means that when you edit an existing file, you must first read in the old file, perform editing on the old file, create a new file at the end of the edit session, KILL the old file name under TRSDOS, and then RENAME the new file name to the old name to get back to square one. Though somewhat tedious, this goes quite rapidly and does offer some automatic protection against **clobbering** an old file before its time.

```
Hints and Kinks 3-2
Typical Edit, Rename Sequence

Here's a typical Edit and rename sequence:

DOS READY
EDIT                      Load Editor
FILE: EQUATE/MAC          Read in old file
  (title, copyright)
*A100                     Edit line 100
00100     TITLE EQUATE
*E EQUATE1/MAC            End edit, write out
DOS READY
KILL EQUATE/MAC           Kill old file
DOS READY
RENAME EQUATE 1/MAC TO    Rename new to old
  EQUATE/MAC                name
DOS READY
```

Relocatable Object File Assembly

The Assembler will read in the source file created by the Editor, assemble it, and produce a **relocatable object file**, a **listing file**, or both on disk. We need to talk about the relocatable object file, since the philosophy here is quite different from the EDTASM approach.

Figure 3-1 shows a typical assembly listing for the Disk Assembler. (This is a module of the Tic-Tac-Toe Artificial Intelligence Program of Chapter 14.) The appearance of this assembly is very similar to what EDTASM would produce. The **source line image** and the edit line number portion are almost identical.

Figure 3-1. Typical Disk Assembler Listing

```
                   00100          TITLE   MAIN1
                   00110          ENTRY   ARTIP
                   00120          EXT     SCREEN,SCARRY,LINE13
                   00130          EXT     NUMBER,ANALAR,ROTATE,ARRXLA,RAND,INPUT
                   00140          EXT     FILLCH,DSPMES,DRAWL,MSGOUT,DELAY,BINBAS
                   00150          EXT     SCRNDS,HISTUP,MEMORY,BASBIN
                   00160          EXT     GRIDTB,MOVETB,HISTRY,ARRAY1,ARRAY2,ROTTAB
                   00170          EXT     ANALTB,RINDT,RINDTR
                   00180          EXT     MOVENO,FRSTF,NXTHIS,PTABLE,RINDW,NOPTE
                   00190          EXT     NOX,NOO,NOSP,ROTPTR
                   00200          EXT     MSG1,MSG2,MSG3,MSG4,MSG5,MSG6,MSG7,MSG8,MSG9
                   00210          EXT     LASTL,MSG10,MSG11,STACK
                   00220  ;*******************************************************
                   00230  ;*        TIC-TAC-TOE LEARNING PROGRAM                *
                   00240  ;*******************************************************
                   00250  ;
                   00260  ;************START OR RESTART ENTRY POINT**************
                   00270  ;
0000'   31 0000*   00280  ARTIP:  LD      SP,STACK        ;SET STACK
0003'   3E 80      00290          LD      A,80H           ;GRAPHICS ALL OFF
0005'   11 0000*   00300          LD      DE,SCREEN       ;START OF SCREEN
0008'   01 0340    00310          LD      BC,832          ;13 LINES
000B'   CD 0000*   00320          CALL    FILLCH          ;FILL DISPLAY AREA
000E'   21 0000*   00330          LD      HL,MSG1         ;HISTORY MESSAGE
0011'   01 0000*   00340          LD      BC,LINE13       ;START OF HISTORY AREA
0014'   CD 0000*   00350          CALL    DSPMES          ;DISPLAY HISTORY
                   00360  ; DRAW GRID HERE
0017'   DD 21 0000* 00370         LD      IX,GRIDTB       ;TABLE FOR GRID
001B'   DD 7E 00   00380  ART005: LD      A,(IX)            ;GET CHARACTER
001E'   FE FF      00390          CP      0FFH               ;TEST FOR TERMINATOR
0020'   28 16      00400          JR      Z,ART008          ;GO IF DONE
0022'   DD 4E 01   00410          LD      C,(IX+1)          ;LOAD HORIZ/VERT
0025'   DD 46 02   00420          LD      B,(IX+2)          ;LOAD # OF CHAR POSNS
0028'   DD 6E 03   00430          LD      L,(IX+3)          ;START OF LINE, LSB
002B'   DD 66 04   00440          LD      H,(IX+4)          ;START OF LINE, MSB
002E'   CD 0000*   00450          CALL    DRAWL             ;DRAW LINE
0031'   01 0005    00460          LD      BC,5              ;5 BYTES PER LINE
0034'   DD 09      00470          ADD     IX,BC             ;POINT TO NEXT LINE
0036'   18 E3      00480          JR      ART005            ;GO FOR NEXT LINE
                   00490  ; ZERO MOVE TABLE
0038'   AF         00500  ART008: XOR     A               ;0
0039'   11 0000*   00510          LD      DE,MOVETB       ;MOVE TABLE ADDRESS
003C'   01 0008    00520          LD      BC,8            ;8 BYTES
003F'   CD 0000*   00530          CALL    FILLCH          ;ZERO MOVE TABLE
0042'   32 0000*   00540          LD      (MOVENO),A      ;ZERO MOVE NUMBER
                   00550  ; TEST FOR FIRST TIME ACTIONS
```

Looking at the machine language code produced by the Assembler, we see that many of the instructions look familiar. `ADD IX,BC`, for example, generates two bytes of `DD 09`, just as it would in EDTASM. If you look at some of the CALLs, however, you'll notice that the address of the CALL is zeroes. Also, the address for `JP` instructions (not shown) appear to be low-valued, such as `0029`. Note also that 16-bit values are printed in "normal" order on the listing; in memory they are still least significant byte followed by most significant byte. What's the logic behind these differences?

In a relocatable object module containing the machine-language code for this assembly, the machine-language code is referenced to the **start** of the module. The location column on the listing starts at location 0' (the prime indicates that the code is relocatable). Jumps within the module generate instructions that contain addresses that are "displacement" addresses from the start of the module.

The **Loader** for the Disk Assembler handles the task of loading in a number of these relocatable object modules and of filling in the proper addresses for **non-relocatable** instructions such as `JPs` and `CALLs`.

```
Hints and Kinks 3-3
Mode Indicators

The character at the end of the location value
is the mode indicator as follows:
    '           Code relative
    "           Data relative
    !           Common relative
    (space)     Absolute
    *           External

These characters are also used after two-byte
values.
```

Another thing that the Loader does is to **link** together locations from module to module. Obviously there has to be some way for modules to communicate with one another. For example, a portion of **main** code, such as is shown in Figure 3-1, might have to call a subroutine represented by another module.

The loader does this by linking together **EXT(ernals)** and **ENTRYs** from module to module. The ENTRY operand is a location within one module that might be referenced by code in another module, as for a CALL or JP. The EXT operand is a label that is external to the module but is referenced inside the module.

Look again at Figure 3-1. At location 14' a CALL is made to a subroutine DSPMES. This subroutine is external to this module and declared as an EXT in the pseudo-ops at the start of the program. Location ARTIP, however, is the main entry point and is declared as an ENTRY. Note that the CALL to DSPMES results in machine code of CD 0000*. The address will be filled in by the loader at **load time**.

A complete assembly-language program is created by **link loading** a group of relocatable modules, such as the one related to the listing in Figure 3-1. The loader compiles a table of all the EXT and ENTRY points, somewhat analogous to the Assembler symbol table, and eventually goes through a "threaded list" to fill in the addresses for every externally referenced (or **global**) label. Labels within the module (**local** labels) are, of course, no problem.

There are several major advantages to this type of assemble and load scheme: (1) we are no longer limited by memory size in constructing large assembly-language packages; (2) we don't have to worry about relocation as the loader takes care of that automatically; and (3) the assembly-language design job can be more structured by dividing the design into a number of separate modules (an especially valuable feature when more than one programmer is working on a project).

There are a few minor disadvantages. For one thing, the entire edit, assemble, load process is less interactive and more time consuming. It does take somewhat more time to generate a new assembly-language package as the entire load process has to be repeated, even if only one line has been changed. However, the extra time spent is a small price to pay for the increased flexibility of the Assembler.

Macro Capability

Another feature the Disk Assembler has over EDTASM is the ability to use **macros**. Macros are essentially **in-line** subroutines generated at **assembly-time**. For example, suppose that we have six instructions that appear a dozen times in an assembly-language program:

```
ADD  HL,HL      ;VALUE*2
PUSH HL         ;SAVE VALUE*2
ADD  HL,HL      ;VALUE*4
ADD  HL,HL      ;VALUE*8
POP  DE         ;VALUE*2
ADD  HL,DE      ;VALUE*10
```

Every time we wanted to perform these six instructions, we'd have to key them in to the edit. Of course, one alternative would be to include them as a subroutine, but another way would be as a macro.

Figure 3-2 shows the six instructions **defined** as a macro by a **macro definition**, and later **invoked** as a macro by a **macro call**. The macro definition involved the pseudo-op MACRO, which defined the label MUL10 as the macro name, and ENDM as the pseudo-op which ended the definition. After the definition, any use of the macro name in the op-code column would automatically generate a **macro expansion** for the six instructions. You can use the macro as many times as needed.

```
                   ; SAMPLE MACRO USE
                   ; FIRST, DEFINE THE MACRO
                   MUL10    MACRO
                            ADD      HL,HL
                            PUSH     HL
                            ADD      HL,HL
                            ADD      HL,HL
                            POP      DE
                            ADD      HL,DE
                            ENDM
                   ; NOW THE MACRO CAN BE INVOKED AS REQUIRED
0000'   3E 23      START:   LD       A,23H              ;BLAH, BLAH
                            MUL10                       ;INVOKE
0002'   29         ADD      HL,HL
0003'   E5         PUSH     HL
0004'   29         ADD      HL,HL
0005'   29         ADD      HL,HL
0006'   D1         POP      DE
0007'   19         ADD      HL,DE
                            END
```

Figure 3-2. Simple Macro Use

Such macro use does have certain advantages as well as disadvantages. By defining the six instructions as a macro, we've made it much easier to generate the code simply by a macro call in the op-code. The code generated is also somewhat faster than the equivalent subroutine would be, since the overhead of the CALL and RET is gone. On the other hand, we've used up quite a bit more memory than the corresponding subroutine.

If this were the extent of macro capability, you might be tempted to forget the whole thing. However, you can use macros with **arguments** to generate **tailored** code to fit the generalized case.

An example of such a macro is shown in Figure 3-3. This macro will take a given character and fill a screen line with it. The macro definition consists of the macro name and MACRO pseudo-op as before, but it also has two **dummy arguments**, CHAR and LINENO, representing the character and the line number, 0-15, to be used.

```
; THIS IS A SAMPLE MACRO WITH ARGUMENTS
; FIRST DEFINE THE MACRO WITH DUMMY ARGUMENTS
FILL    MACRO   CHAR,LINENO
        LOCAL   LOOP   <----- THIS PREVENTS DUPLICATE
        LD      A,CHAR        LABELS WHEN MACRO
        LD      HL,3C00H+LINENO*64    IS EXPANDED
        LD      B,64
LOOP:   LD      (HL),A
        INC     HL
        DJNZ    LOOP
        ENDM
; NEXT, INVOKE THE MACRO WITH REAL ARGUMENTS
        FILL    ' ',0
```

```
0000'   3E 20                      LD      A,' '          }
0002'   21 3C00                    LD      HL,3C00H+0*64  }
0005'   06 40                      LD      B,64           }
0007'   77               ..0000:   LD      (HL),A         } EXPANSION 1
0008'   23                         INC     HL             }
0009'   10 FC                      DJNZ ..0000            }
                 00230                     FILL    '*',1  }
000B'   3E 2A                      LD      A,'*'          }
000D'   21 3C40                    LD      HL,3C00H+1*64  }
0010'   06 40                      LD      B,64           } EXPANSION 2
0012'   77               ..0001:   LD      (HL),A         }
0013'   23                         INC     HL             }
0014'   10 FC                      DJNZ ..0001            }
                 00240                     FILL    88H,2  }
0016'   3E 88                      LD      A,88H          }
0018'   21 3C80                    LD      HL,3C00H+2*64  }
001B'   06 40                      LD      B,64           } EXPANSION 3
001D'   77               ..0002:   LD      (HL),A         }
001E'   23                         INC     HL             }
001F'   10 FC                      DJNZ ..0002            }
                 00250                     END
```

Figure 3-3. Macro Use With Arguments

The arguments are called **dummies** because they only serve to denote when you are to use "real" arguments when the macro is called. When the macro is invoked, as shown in the listing, "real" arguments replace the dummy arguments, and specialized code is generated with the real arguments put in the proper places in the code. Any number of dummy arguments can be used in a macro, subject only to line length.

It is not hard to see how to create macros to define an entire applications language. The resulting code for the application could consist primarily of macro calls, allowing quick development time but at the expense of memory.

> **Hints and Kinks 3-4**
> **More Complicated Macros**
>
> The macros used as examples in the text are quite straightforward. The Disk Assembler, though, has a number of macro operators, conditional assembly pseudo-ops, and rules for macro definition and use that may be used to construct very powerful macros. Such things as a NUL operator, to test for a null argument, EXITM, to terminate a macro before a complete expansion, and SET, to permit redefinition of a variable name are but a few of the functions that can be used to create truly horrendous code!

Pseudo-Ops For Program Sections and Conditional Assembly

The Disk Assembler adds a number of new pseudo-ops other than those for EXT(ernals), ENTRYs, and macros. Some of these are related to **program sections** and some to **conditional assembly**.

Program Sections

The pseudo-ops ASEG, DSEG, and CSEG are related to **program sections.** The Disk Assembler is initialized in the "relocatable" mode; any code produced in this mode is relocatable and can be modified by the loader at load time to run wherever it is being loaded.

If the ASEG pseudo-op is used, however, all code following will be in **absolute mode**. It will be assembled to run at one specific area in memory. The ASEG is normally followed by an ORG (Origin) to define the area of memory at which the code is to be assembled.

```
ASEG                    ;SET ABSOLUTE
ORG     8000H           ;START OF ABSOLUTE AREA
```

The CSEG pseudo-op defines a **code relative** section of the program. This is the relocatable code section. CSEG is not required unless an ASEG has been used, in which case you should use a CSEG to force the Assembler back into its default mode. You can intermix ASEGs and CSEGs as needed.

The third pseudo-op of this type is the DSEG, or **data relative** segment definition. Here again, DSEG should be followed by an ORG to define where the data is to go. The DSEG could be used, for example, to include all variables and working storage in each program module in a separate data-segment area of memory for convenience in debugging.

Essentially, the Disk Assembler maintains three separate location counters — one for the absolute, one for the code relative, and one for the data relative sections of memory. An example is shown in Figure 3-4, where the three types of program sections are used with relative impunity.

```
0000'                 ;THIS IS ABSOLUTE SECTION. USED TO SPECIFY ABSOLUTE CODE.
                              ASEG
                              ORG     8000H
8000      C3 8000     LOOP1:  JP      LOOP1
                      ;THIS IS DATA SECTION. USED TO SPECIFY A DATA SECTION FOR
                      ;VARIABLES AND DATA.
8003                          DSEG
                              ORG     0C000H
C000"     01          DATA1:  DEFB    1
                      ;THIS IS CODE RELATIVE. USED TO SPECIFY "NORMAL" RELOCATABLE
                      ;CODE.
C001"                         CSEG
0000'     C3 0000'    LOOP2:  JP      LOOP2
                      ;ANOTHER DSEG. NOTE ADDRESS IS ONE MORE THAN LAST DSEG.
0003'                         DSEG
C001"     02          DATA2:  DEFB    2
                      ;ANOTHER ASEG. NOTE ADDRESS IS AT NEXT ABSOLUTE ADDRESS.
C002"                         ASEG
8003      C3 8003     LOOP3:  JP      LOOP3
                      ;ANOTHER CSEG. NOTE ADDRESS IS AT NEXT CODE RELATIVE
                      ;ADDRESS.
8006                          CSEG
0003'     C3 0003'    LOOP4:  JP      LOOP4
                              END
```

Figure 3-4. ASEGs, CSEGs, and DSEGs

Conditional Assembly

Another set of pseudo-ops are concerned with **conditional assembly.** Conditional assembly refers to a segment of code that is either assembled or not assembled according to some defined condition. Conditional assembly might be used, for example, by a company that has many versions of a software program. One segment of code might be necessary if a user has a disk system, while another might be necessary if the user has cassette. The section for a disk system might appear as follows:

```
;ASSEMBLE THIS CODE FOR DISK
        IF      DISK    ;DISK FLAG
DISKC   LD      A,5     ;LOAD FLAG
        .
        .
        .
        ENDIF
```

If variable DISK is not zero, this code will be assembled. If DISK=0, the code will not be assembled. The dots indicate additional code in the segment. There are a number of conditional assembly pseudo-ops including the REPT ("repeat n times") and IRP pseudo-ops. For information on the rest, consult the *Disk Assembler Programming Manual.*

Hints and Kinks 3–5
Elegant Pseudo-Ops

There are a number of pseudo-ops we haven't talked about here, such as SET, which allows an assembly time variable to be redefined, REPT, which allows a block of statements to be repeated n times; and LOCAL, which defines a local label within a macro.

In my opinion, most assembly-language programmers probably won't exercise the full capabilities of the Disk Assembler by using such ''elegant'' operations. The last time I used an elaborate scheme to have the assembler generate a large complicated table of data, I was awakened at 2:00 a.m. by a call from the computer operator. The operator informed me that my (200-line) assembly had been assembling for 4 hours and asked if it was ''normal''!

Pseudo-Ops for Listing Format

There are a number of pseudo-ops in the Disk Assembler that make listings "prettier." TITLE specifies a title to be used on each page of the listing. SUBTTL allows the user to specify a subtitle. PAGE causes a "page eject" to start the listing on a new page; this is handy for isolating logically different sections of code. .COMMENT may be used in place of comment lines starting with a semicolon. The first character of the following text is used as a **delimiter** and the following text is used as a comment block until the same character is encountered.

Several other pseudo-ops control listing output, macro listing output, and **cross-reference** listing output. See the Disk Assembler manual for details.

Using the Disk Assembler

After you have used the Editor to create assembly-language source modules, the Assembler reads in a source module from disk, assembles the code, and optionally produces a relocatable object module and **listing file**.

The format for the assembly command is —

```
*NAME1,NAME2=NAME3
*,NAME2=NAME3
*NAME1,=NAME3 or
*=NAME3
```

In case this format's confusing (I found it so), we'll interpret for you. The first command assembles source file, NAME3, from disk and produces a relocatable object file, NAME1, and a listing file, NAME2, on disk. The listing file may be listed by a PRINT command.

The second command does *not produce a relocatable object-file.* The third command does *not produce a listing file.* The fourth command produces *neither* a relocatable object file *nor* a listing file (handy for assembly error checks).

The names may actually be the same for all three. If a source file is on disk called NAME/MAC, the command

```
*NAME,NAME=NAME
```

produces a relocatable object file called NAME/REL and a listing file called NAME/LST. The file extensions are automatically generated by the assembler. Of course, you can also use passwords (ugh!) or disk drive numbers.

Hints and Kinks 3–6

CREF

We haven't mentioned the cross reference facility of the Editor/Assembler at all. It's not that it isn't useful, but it does create another step in the assembly process. You must set the CREF switch during assembly by an assembler command such as *TEST,TEST/C=TEST. The resulting listing file has ''hooks'' in it that enable processing by CREF. The result is an alphabetized listing of the variable names along with line numbers of lines in which each is referenced or defined. Use at your option.

Loading the Object Modules to Produce a Command File

Let's assume we've gone through the assembly process for the three modules shown in Figure 3-5.They make up a horrendously complicated program to clear the screen, print a title, read a key of 0-7, and output the key to the center of the screen. Three modules were used here so we could illustrate the linkages between the modules at load time.

```
                       00040          TITLE   READ
                       00050          ENTRY   READ
                       00100  ; READ ROUTINE. READS KEYS 0-7 AND DISPLAYS.
0000'   3A 3810        00110  READ:   LD      A,(3810H)       ;GET 0-7
0003'   B7             00115          OR      A               ;TEST A
0004'   28 FA          00120          JR      Z,READ          ;GO IF NONE
0006'   06 FF          00130          LD      B,-1            ;COUNTER
0008'   04             00140  REA010: INC     B               ;BUMP COUNTER
0009'   0F             00150          RRCA                    ;SHIFT OUT BIT
000A'   30 FC          00160          JR      NC,REA010       ;GO IF NO 1 BIT
000C'   78             00165          LD      A,B             ;GET DIGIT
000D'   C6 30          00170          ADD     A,30H           ;CONVERT TO ASCII
000F'   32 3E20        00180          LD      (3E20H),A       ;STORE IN SCREEN
0012'   C9             00190          RET                     ;RETURN
                       00200          END
```

```
                       00100          TITLE   PRINT
                       00110          ENTRY   PRINT
                       00120  ; PRINT ROUTINE. PRINTS TITLE
0000'   21 3C00        00130  PRINT:  LD      HL,3C00H        ;SCREEN START
0003'   11 000E'       00140          LD      DE,MSG1         ;MESSAGE ADDRESS
0006'   1A             00150  PRI010: LD      A,(DE)          ;GET CHARACTER
0007'   B7             00160          OR      A               ;TEST FOR ZERO
0008'   C8             00170          RET     Z               ;RETURN IF END
0009'   77             00180          LD      (HL),A          ;STORE CHARACTER
000A'   23             00190          INC     HL              ;BUMP SCREEN PNTR
000B'   13             00200          INC     DE              ;BUMP CHAR PNTR
000C'   18 F8          00210          JR      PRI010          ;CONTINUE
000E'   54 49 54 4C    00220  MSG1:   DB      'TITLE',0       ;TITLE MESSAGE
0012'   45 00
                       00230          END
```

```
                       00100          EXT     PRINT,READ
                       00110  ; MAIN DRIVER ROUTINE. CLEARS SCREEN, PRINTS MESSAGE
                       00120  ; AND READS AND DISPLAYS KEY BY CALLING PRINT,READ.
0000'   21 3C00        00130  START:  LD      HL,3C00H        ;SCREEN START
0003'   11 4000        00140          LD      DE,4000H        ;SCREEN END+1
0006'   3E 20          00150          LD      A,' '           ;BLANK
0008'   77             00160  LOOP1:  LD      (HL),A          ;STORE BLANK
0009'   23             00170          INC     HL              ;BUMP PNTR
000A'   B7             00180          OR      A               ;CLEAR CARRY
000B'   ED 52          00190          SBC     HL,DE           ;TEST DONE
000D'   19             00200          ADD     HL,DE           ;RESTORE
000E'   20 F8          00210          JR      NZ,LOOP1        ;GO IF NOT DONE
0010'   CD 0000*       00220          CALL    PRINT           ;PRINT TITLE
0013'   CD 0000*       00230  LOOP2:  CALL    READ            ;READ 0-7
0016'   18 FB          00240          JR      LOOP2           ;CONTINUE READING
                       00250          END
```

Figure 3-5. Loading Object Modules

The modules are out on disk as MAIN, PRINT, and READ and emulate the usual arrangement of modules produced while using the Disk Assembler. The main portion of code calls the other modules as subroutines. There could also be ENTRY points in MAIN or EXTernals in PRINT and READ if this were more complicated code.

The Loader is entered by typing L80 after the TRSDOS prompt, TRSDOS READY. The chief commands to consider in the Loader are the -P command, the -N command, and the -E command. The -P command has the form -P:XXXX. It sets the loader location counter to location XXXX. The -N command has the form NAME-N. It establishes a name for the command file. The -E command ends the edit, outputs the linked relocatable modules to the disk as command (/CMD) file NAME, and ends the edit.

The load sequence of commands is shown in Figure 3-6. The load location counter is first set to 8000H by a -P command. The three modules are then loaded by typing in each name. Finally, the linked modules are output to disk as TEST/CMD, and the load is ended. TEST can now be loaded and executed from TRSDOS by simply typing TEST.

```
*-P:8000                                    SET LOAD ADDRESS
*MAIN                                       LOAD MAIN MODULE
DATA        8000        8018                SIZE OF CURRENT DATA
PRINT*      8011        READ*       8014    UNRESOLVED EXTERNS
*PRINT                                      LOAD PRINT MODULE
DATA        8000        802C                SIZE OF CURRENT DATA
READ*       8014                            UNRESOLVED EXTERNS
*READ                                       LOAD READ MODULE
DATA        8000        803F                SIZE OF CURRENT DATA
*TEST-N, -E                                 FILE NAME "TEST", END
(0000       803F)
DOS READY
```

Figure 3-6. Typical Load Commands

Figure 3-6 also shows the loader symbol table displayed after each load. This lists all unresolved symbols in addition to the memory area loaded. At the end of the load, all externals should have been resolved, or a load error will result. If any external is unsatisfied, it means that a module referenced an external name by an EXTern, and there was no corresponding ENTRY point. This may mean that you did not load the module containing the name, or that you didn't declare the name in an ENTRY. Either condition is an error.

The procedure above is the loading process in microcosm for any size assembly-language program. The Tic-Tac-Toe program in Chapter 14 uses 25 separate modules with a large number of ENTRY points, yet the procedure is virtually identical.

The "core image" of the three modules above is shown in Figure 3-7. All addresses have been satisfied during the load process, and the program appears as one contiguous program block.

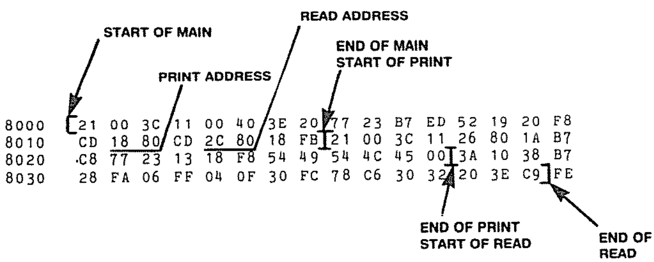


Figure 3-7. Core Image Module

Chapter Four

Loading, Executing, and Debugging Assembly-Language Programs

Up to this point we've talked about general considerations in producing programs using EDTASM and the Disk Assembler — how to edit, assemble, and list source files. We'll move on to discuss the next step, loading and executing the resulting object code. You can run assembly-language programs as "stand-alone" programs or interface them to BASIC programs. This chapter describes both methods. We also cover "debugger" programs and some tricks in debugging techniques. In short, we're going to talk about the subject, "Now that I have the assembled program, what do I do with it?"

EDTASM System Tapes

The result of an edit and assembly using EDTASM is a SYSTEM file on cassette. A SYSTEM file resembles the machine-language code seen on the EDTASM listing. It contains other data, however, to enable the Level II BASIC interpreter to load the machine-language code into the proper place in memory and verify the data as correct. The format for SYSTEM tapes is shown in Figure 4-1.

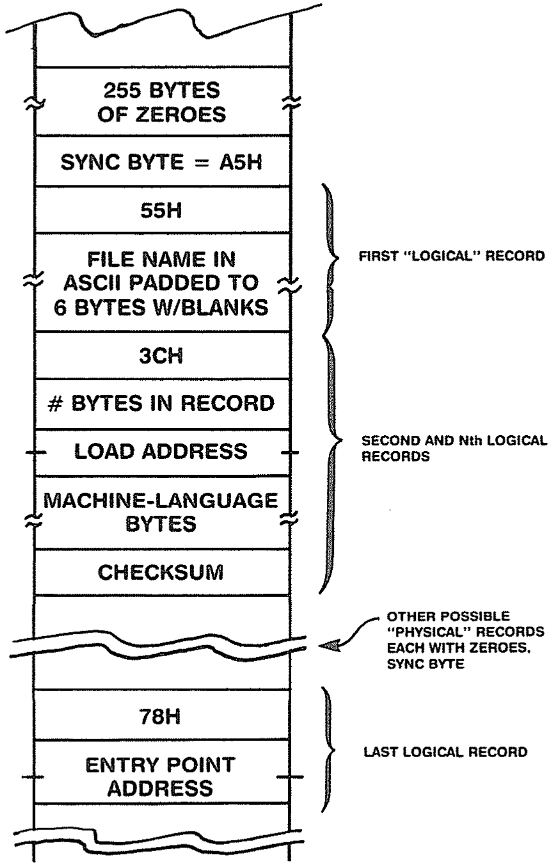


Figure 4-1. SYSTEM Tape Format

A string of 255 zeroes are at the beginning of the cassette file, followed by a **sync byte** of A5H. The sync byte synchronizes the cassette tape driver of the BASIC interpreter — informing it that the next byte will be valid data. The zeroes and sync byte precede each **physical record** on the SYSTEM tape. A SYSTEM tape may have one or more physical records, depending upon the size of the object file. Each physical record is divided into a number of **logical records**, which are essentially **load item** blocks defining file name, data, or entry point. After the sync byte of the first logical record, a file name code of 55H occurs. This informs the interpreter that a file name will appear in the next 6 bytes. The file name is 6 bytes long, padded out with blanks. This is the first logical record of the SYSTEM tape.

The next logical record is headed by a data code of 3CH. This informs the interpreter that machine language data follows in the record. The next byte in the data record is a count of the number of data bytes in the record. If this byte is a zero, the number of data bytes is 256, otherwise the count is the actual number. The next two bytes are the load address for the data to follow. They're set in standard address format, least significant byte followed by most significant byte. Next are the machine-language bytes. The last byte of each data record is a **checksum** byte. A checksum is nothing more than a "check byte" formed by adding all of the individual data bytes. It is used for comparison to another add of the data bytes on load to verify that the data has been loaded properly.

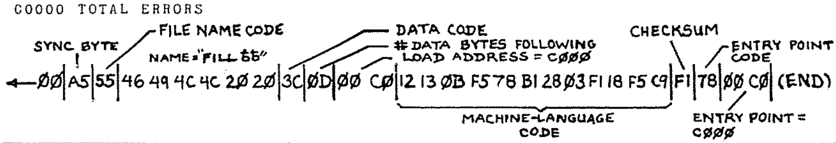
The last logical record on the SYSTEM tape is an entry point code of 78H and two bytes that represent the entry point for program execution.

Hints and Kinks 4-1

An Actual Cassette Object File

The data below represents an actual cassette SYSTEM file produced by assembling the program below onto cassette.

```
C000           08760        ORG     0C000H
               08770 ;
               08780 ;*****************FILL CHARACTER SUBROUTINE***************
               08790 ;*   FILLS DESIGNATED AREA WITH GIVEN CHARACTER          *
               08800 ;*   ENTRY:  (A)=CHARACTER                               *
               08810 ;*           (DE)=AREA                                   *
               08820 ;*           (BC)=NUMBER OF BYTES, 1-65525; 0 IS 65536   *
               08830 ;*   ALL REGISTERS SAVED EXCEPT BC,DE                    *
               08840 ;
C000 12        08850 FILLCH LD      (DE),A              ;FILL CHARACTER
C001 13        08860        INC     DE                  ;BUMP POINTER
C002 0B        08870        DEC     BC                  ;DECREMENT COUNT
C003 F5        08880        PUSH    AF                  ;SAVE FILL CHAR
C004 78        08890        LD      A,B                 ;TEST FOR ZERO
C005 B1        08900        OR      C
C006 2803      08910        JR      Z,FIL010            ;GO IF DONE
C008 F1        08920        POP     AF                  ;RESTORE FILL CHAR
C009 18F5      08930        JR      FILLCH              ;CONTINUE
C00B F1        08940 FIL010 POP     AF                 ;RESTORE A
C00C C9        08950        RET                        ;RETURN
C000           08960        END     FILLCH
00000 TOTAL ERRORS
```



The SYSTEM tape format is identical to that produced by T-BUG, the **debugger** program for cassette-based systems. T-BUG has the capability of producing such a file by the P (punch) command. Conveniently enough, T-BUG can also load in the cassette file by the L command. This means that the object tape produced by EDTASM can either be read by using the BASIC SYSTEM command or by T-BUG!

System Considerations for EDTASM Object Files

EDTASM object files represent one huge program. The start of the program is defined by the ORG (Origin) pseudo-op at the beginning of the program. The size of the program can be determined by the location column on the listing. You can put buffer areas and **working storage** for variables and tables anywhere in the program that you want. There may be a large "open-ended" buffer or buffers in the program, generally at the end of the program to build "up" into higher and higher memory. You can see an example of this structure in MORG, the EDTASM program of Chapter 13.

What should the value be for Origin? This depends upon the environment in which you are going to use the assembly-language program. If your system does not have disk, and the assembly-language program does not interface to a BASIC program, then the program may be ORGed at 4980H. T-BUG occupies the area from about 4380H through 497FH, with an internal stack area building downwards from 497FH. When the program is debugged using the T-BUG stack, the program may use all of memory for storage from the end of the program up to **top of memory**.

You might be curious about establishing your own stack area. You can do it. In this case, set aside approximately 100 bytes at some convenient point either **in** your program, at top of memory, or in some area of memory that you know won't be used. The stack should be initialized immediately with a

```
START LD SP,PEND+100
```

or similar instruction (here the stack has been set to the end of the program plus 100 bytes). Remember, the location loaded into the stack pointer represents the first location to be used by the stack **plus one**.

Hints and Kinks 4–2
Size of Stack

Many programmers ask "How big should the stack area be?" There's no definitive answer. Each time a CALL is made, the two bytes of the return address are pushed onto the stack. Each time a PUSH instruction is executed to temporarily store data, two bytes of a register pair, IX, or IY are pushed onto the stack. A third function that pushes data onto the stack is interrupt action. This can occur in a disk system with the real-time-clock enabled. The "worst case" number of bytes used, therefore is:

2*(number of calls active at any time + maximum number of pushes active at any time + possible interrupt)

If 100 bytes are allocated, this allows for 50 CALL/PUSH/interrupt levels, which are probably more than adequate.

The configuration for assembly-language programs in this type of environment is shown in Figure 4-2.

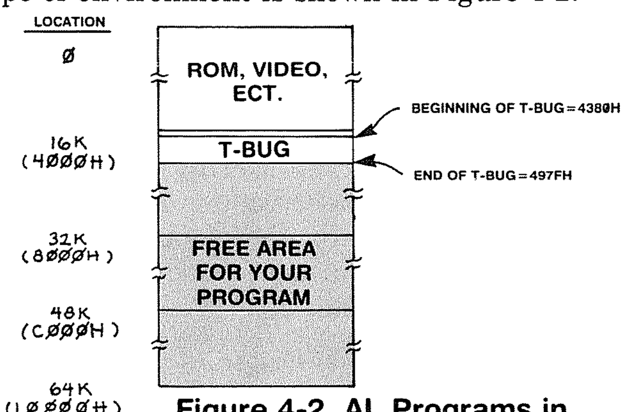


Figure 4-2. AL Programs in Minimum Configuration Systems

If the program is to be used together with a BASIC program in a non-disk system, then there are two considerations. First of all, the ORG can't overlap the BASIC program area. The BASIC program builds up in memory from about 4200H on. The program statements aren't the only things occupying memory, however. Simple variables, arrays, strings, and a BASIC interpreter stack are also stored, as shown in Figure 4-3.

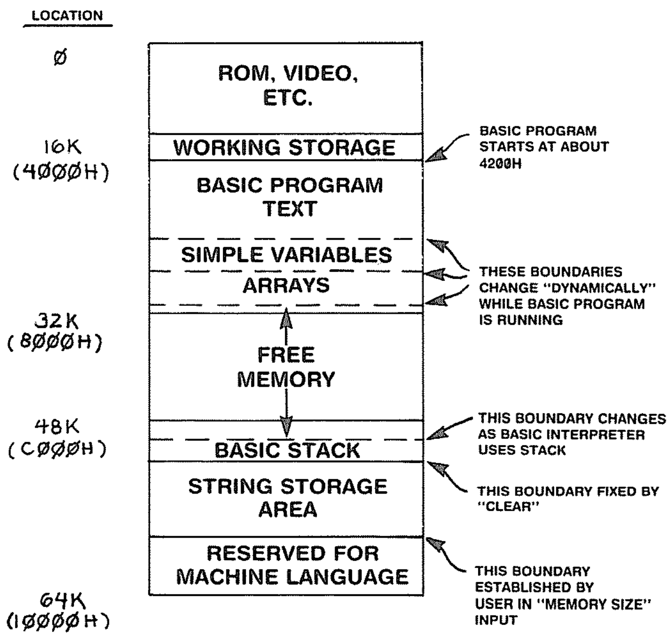


Figure 4-3. BASIC Memory Allocation

Compute the ORG for the assembly by taking the size of the assembled program plus any buffer areas or tables outside of the program and subtracting it from your system's top of memory plus one (8000H for 16K, C000H for 32K, or 10000H for 48K). You may use the BASIC stack with no problem; if you establish your own stack area, add this into the total size to be subtracted.

Assemble the program with this Origin, and protect this memory area by entering the Origin value for MEMORY SIZE when BASIC is first entered. The stack area for BASIC will now build downwards from the Origin minus one location.

A special case arises when the assembly-language program uses machine code **embedded** in the BASIC program. In this case, a short assembly-language program becomes part of the BASIC program itself and can be loaded easily. We'll talk about these techniques in the next chapter.

Debugging With T-BUG

After you edit, assemble, and thoroughly **desk-check** your assembly-language program, you should load it along with Radio Shack's cassette based debugging program, T-BUG. The procedure is simple:

1. Enter Level II BASIC.
2. Enter the area of memory to be protected (MEMORY SIZE), computed by the procedure above.
3. Type SYSTEM for the ">" prompt. This causes the BASIC interpreter to enter the **monitor** mode.
4. Rewind your object tape and prepare for a load.
5. Enter NAME after the *? prompt. NAME is the name you've given your object file for assembly. If you haven't named your file, the cassette file will be called NONAME.
6. Your object tape should now load, as indicated by the blinking asterisk.
7. After a successful load, ready the T-BUG cassette.
8. Enter TBUG after the *? prompt.
9. The T-Bug file should now load, followed by the *? prompt.
10. Enter / after the prompt.
11. T-BUG should now be entered, as evidence by a # prompt.

With both T-BUG and your object program in memory, you're ready to do some serious debugging. (This procedure can't be used when a BASIC program is also resident, as the BASIC program will "overlay" T-BUG.) First of all, you should be familiar with T-BUG commands. These are described in the T-BUG manual, and we won't study them here, but we'll tell you how to make efficient use of them.

Hints and Kinks 4-3
Recap of T-BUG Commands

#M aaaa	Display location aaaa
ENTER (after M)	Display next location
X (after M,J,B,P)	Exit operation
#R	Display registers
#P aaaa bbbb cccc NAME	Write cassette from aaaa through bbbb with starting address cccc and file name NAME
#L	Load a T-BUG or SYSTEM tape
#B aaaa	Set breakpoint
#F	Restore instruction after breakpoint
#G	Continue from breakpoint
#J aaaa	Jump to location aaaa

The first rule of debugging is to be sure you've thoroughly **desk checked** the program. This means you've taken a listing of the program and gone over it minutely, instruction by instruction, to see that it works as planned. This may mean some reference to flow charts and design specs with larger programs.

After desk checking as much as possible, the procedure is basically, "Get it working (even though it probably will run badly) then go back and clean up the flaws." This procedure works well as long as the program design is substantially correct, and there isn't an overabundance of errors.

To get the program working, use a type of "binary search" for flaws. First of all, go ahead and give it a try. Chances are that the program will blow up, but it takes an extremely strong-willed programmer not to try that first execution in hopes that everything will work right off the bat. (It never does!) Reload the program if necessary.

Having gotten that out of your system, set a breakpoint about half way through by using the T-BUG B(reakpoint) command. If, for example, the program is about 100 locations long, set a breakpoint at a location 50 bytes from the start of the program. Now use the J(ump) command to start the program. One of two things will happen. The program may "bomb" again, necessitating a reload, or the breakpoint will be reached.

If the breakpoint is reached, F(ix) the breakpoint and start looking at variables and "tracks" of the program up to that point. See that variables and actions appear to be correct. If you find strange results, write them down and start correcting them one by one by setting new breakpoints at an earlier condition. Don't be too picky about testing every possible condition.

> Hints and Kinks 4-4
>
> How Does T-BUG Breakpoint?
>
> When you put a breakpoint at a specific location, T-BUG puts a CD 80 43 at that location and the next two bytes. This is a ''CALL 4380H'' that calls the T-BUG breakpoint routine. F(ixing) the breakpoint restores the original three bytes.
>
> Be careful in breakpointing that the three bytes temporarily destroyed in breakpointing are not used as variables or instructions by the code to be checked.

If results seem to be correct on cursory looks up to this point, set a breakpoint about half-way into the remaining area, and repeat the procedure. **Zero-in** on the errors in the same fashion.

Patching

You have two alternatives in correcting errors as they're found in the program. You can reassemble and reload, or you can **patch**. If you have never used a patching technique; stay tuned and we'll explain with an example.

Figure 4-4 shows an EDTASM program from MORG in Chapter 13 with two errors. The subroutine is missing an OR C, and the JR Z,FIL010 is erroneously JR NZ,FIL010. (Obviously I had a thorough desk check here!)

```
8583           08770           ORG     8583H
               08780 ;****************FILL CHARACTER SUBROUTINE***************
               08790 ;*   FILLS DESIGNATED AREA WITH GIVEN CHARACTER          *
               08800 ;*   ENTRY:  (A)=CHARACTER                               *
               08810 ;*           (DE)=AREA                                   *
               08820 ;*           (BC)=NUMBER OF BYTES, 1-65525; 0 IS 65536   *
               08830 ;*   ALL REGISTERS SAVED EXCEPT BC,DE                    *
               08840 ;
8583 12        08850 FILLCH    LD      (DE),A              ;FILL CHARACTER
8584 13        08860           INC     DE                  ;BUMP POINTER
8585 0B        08870           DEC     BC                  ;DECREMENT COUNT
8586 F5        08880           PUSH    AF                  ;SAVE FILL CHAR
8587 78        08890           LD      A,B                 ;TEST FOR ZERO
               08900 ;         OR      C                   ;***MISSING***
8588 2003      08910           JR      NZ,FIL010           ;GO IF DONE***S/B Z***
858A F1        08920           POP     AF                  ;RESTORE FILL CHAR
858B 18F6      08930           JR      FILLCH              ;CONTINUE
858D F1        08940 FIL010    POP     AF                ;RESTORE A
858E C9        08950           RET                       ;RETURN
               08960 ;
0000           08970           END
00000 TOTAL ERRORS
```

Figure 4-4. Flawed Program

In testing the program, we loaded the A register with 23H, the DE register pair with 8000H, and the BC register pair with 64H to fill locations 8000H-8063H with 23H. We breakpointed on the RET by B858E. When the breakpoint was reached, it turned out that only the first byte at location 8000H had been filled.

In checking through the code, we found that the JR NZ,FIL010 should have been a Z. Rather than reassembling, we decided to patch the code. The JR NZ,FIL010 assembles as 2003H. By looking in Appendix II (or the Assembler manual), we found that a JR Z would be 28XXH. This is simple to fix using T-BUG — we simply perform a M 8588 20 28 to change the location to a JR NZ,FIL010 and record the patch in a list of patches.

Having patched, we tried again. This time, with an identical procedure, we found 23H stored from 8000H through

8583H! It seems 23H was stored until the program was destroyed! After further head scratching, we found that we had left out an `OR C` instruction!

This time we found we couldn't easily patch because we couldn't fit in another one-byte instruction between the last **bit** of location 8587H and the first **bit** of location 8588H! How do we patch in this case?

In cases like this, we can jump out to a **patch area**, "gin up" some code to make the change, and jump back into the routine. A patch area is any unused area that can be used to store temporary fixes. In this case, since we weren't using locations C000H and up, we designated that as a patch area.

To get to the patch area, we had to put the three bytes of a `JP` somewhere in the code. We put them into the locations that would have normally held `DEC BC` as shown in Figure 4-5. We deleted the three instructions of `DEC BC`, `PUSH AF`, and `LD A,B` and substituted a `JP C000H` instead.

Location	Old		Patched	
8583	12		12	
8584	13		13	
8585	ØB	DEC BC	C3	JP CØØØH
8586	F5	PUSH AF	ØØ	
8587	78	LD A,B	CØ	
8588	$\boxed{28}$03		28Ø3	
858A	F1		F1	
858B	18F6		18F6	
858D	F1		F1	
858E	C9		C9	

PREVIOUSLY PATCHED JR2, FILØ1Ø

Figure 4-5. Patching to Flawed Program

How did we know what code to put in for the JP? By consulting the Appendix, by referring to the Assembler manual, or by **looking through the listing until a similar type instruction was found**!

At the C000H patch area, we put in machine code representing the three deleted instructions, machine code for OR C, and a JP back to location 8588H, as shown in Figure 4-6. You can patch virtually any program this way.

Hints and Kinks 4–5
Hand Assembling

There are still some programmers that I know of that persist in ''hand assembling.'' These are the same types of people that collect balls of string eight feet in diameter, cultivate their own wheat in their city plot, and try to interface Baudot teletypes to their TRS-80s!

What we are doing in patching is a small exercise in ''hand assembling.'' Instead of letting the assembler look up opcodes and resolve addresses, we are doing it ourselves. Hand assembling is not too much of a chore in this case, since we're working with only a few instructions. We can simplify it by looking at our listings for identical or similar instructions and use their formats to make the patches. However, when there are many instructions to be assembled, it's probably best to let the assembler do the job – even to the extent of running a short assembly to get the patches! I've spent too many nights at some customer's site hand assembling patches into a computerized system to advise you differently. Help stamp out hand assembling!

LOCATION	CONTENTS	CODE	NOTES
CØØØ	ØB	DEC BC	RESTORE ORIGINAL INSTRUCTIONS
1	F5	PUSH AF	
2	78	LD A,B	
3	B1	OR C	NEW INSTRUCTION
4	C3	JP 8588H	JUMP BACK
5	88		
6	85		

Figure 4-6. Patch Area Contents

With the patch in place, we tried the subroutine again, and it worked fine for the one case.

This patching process requires some effort, and you may not want to use it. However, you can become very proficient at it and use it to advantage to debug large programs that take a great deal of time to reassemble because of printing time. Hard thinking vs. time: it's a tradeoff you'll have to assess.

A patched program can be written out to cassette at any time. The advantage is that the patches don't have to be reentered when the program **blows up** the next time. Also, as long as you're writing out to cassette, why not include the T-BUG area along with the program area? That way the entire program area, T-BUG and all, can be read in by a SYSTEM command to simplify reloading. The entry point for T-BUG is 17312 decimal.

Disk Assembler Files

The final output of a set of edits and assemblies and a load operation with the Disk Editor/Assembler package is a **command** file (/CMD) on disk. You can load and execute this command file by simply typing the name of the file after the DOS READY prompt. In most cases, however, you must do some debugging before the program is ready to run in this fashion.

The memory area specified to the Loader is, like EDTASM programs, dependent upon the environment in which you are going to run the program. Since the program will normally be loaded by TRSDOS, it must not overlap the TRSDOS area, which usually ends at 6FFFH. This will enable DUMP commands to be used to save **patched** versions of the program.

If the program is to run "stand-alone," without interface to Disk BASIC, then the area from 7000H through top of memory is available for program use. This won't conflict with use of the Disk DEBUG package, as it loads into an **overlay** area below 7000H. If DEBUG were used for debugging, its stack area would be internal to the DEBUG program and wouldn't conflict with program use of the 7000H and up area. Of course, a separate stack area could be maintained by the program if you want. This configuration is shown in Figure 4-7.

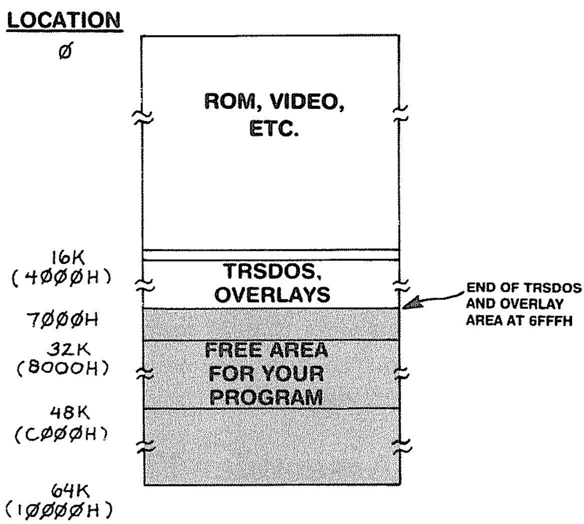


Figure 4-7. AL Programs in Disk Systems Without BASIC Interface

If you're going to use the program together with Disk BASIC, then the area from 7000H on is used for storage of the BASIC program, simple variables, arrays, strings, and the stack as shown in Figure 4-8. This allocation scheme is identical to Level II BASIC, except that the memory allocation area starts higher in memory.

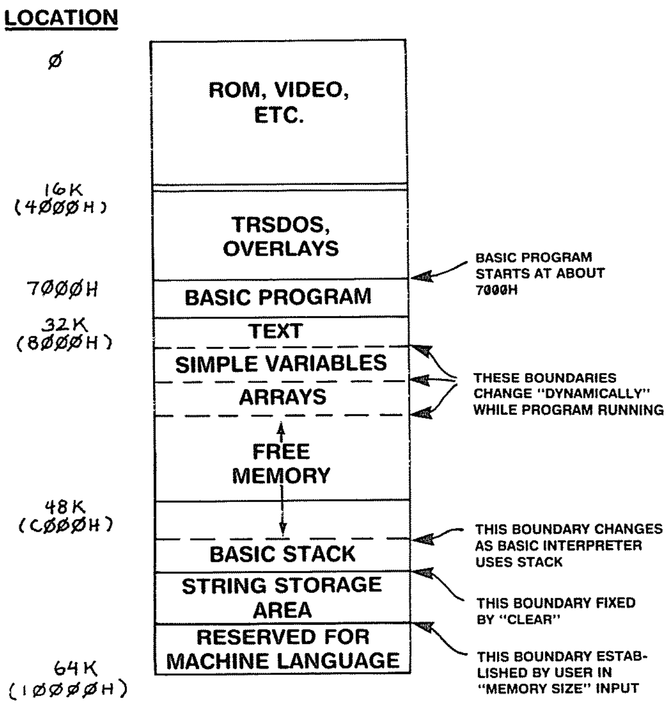


Figure 4-8. AL Programs in Disk Systems With BASIC Interface

The plan here is for you to specify the load address high enough in memory for both BASIC and the assembly-language program to coexist peacefully. Compute the loading address by making a test load and observing the size of the final configuration. Add to this size any external buffer or table areas and approximately 100 bytes for an external stack, if one is to be used. Subtract this total from the top of memory value plus one. The result is the load address you'll specify to create the command module for the program. This process is shown in Figure 4-9. In many cases, of course, you'll have more than enough memory to be fairly "sloppy" in specifying the load address.

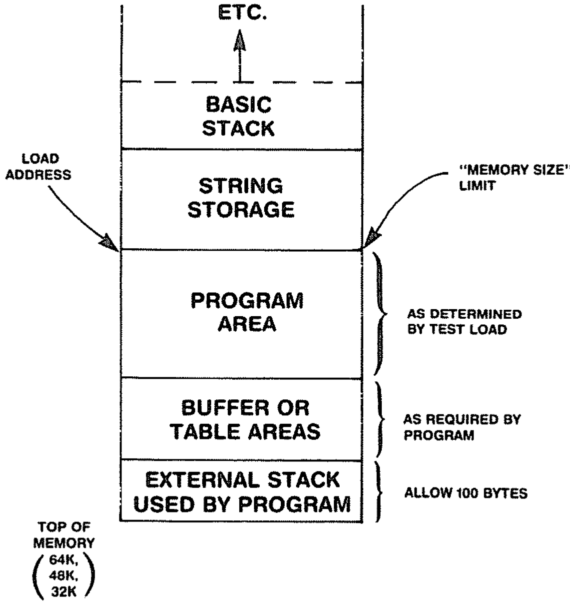


Figure 4-9. Computing the Load Address

Debugging with DEBUG

Once you've created a command module, you're ready to begin debugging. Actually, you should have already done the bulk of the debug! Prior to using DEBUG, you should have done a thorough and detailed check of your assembly-language code. This **desk check** will usually reveal errors in **logic** and implementation that will be much easier to correct by reassembly than by **patching** during debugging.

Hints and Kinks 4–6
Desk Checking

A typical desk check would involve reviewing the specs and flow charts, together with the program listing(s) to see that no <u>logical</u> errors have been made.

Next, you might start with the simplest subroutines and ''walk through'' them, instruction by instruction, using sample data. Write the contents of each register down on paper as you follow the instructions. Work your way up through the program levels until you've gone over all the code once. Reassemble if necessary.

Next, go through the same process again. Look for errors! They're there. (It might help to pretend that it's someone else's code.) This process should be repeated until you've gone through all code once without finding new errors. Yes, this is a lot of work. But it's much easier to correct errors now than during program execution.

When you're satisfied, start the actual ''on-line'' debugging. You'll probably say ''What a stupid error – I should have seen that!'' (It always happens to me . . . .)

To use DEBUG, first load DEBUG by typing in DEBUG after the TRSDOS DOS READY prompt. Then type in the name of the load module you created during the loading process to load it into memory in the area you specified. Now hit BREAK, and you will be in DEBUG.

The Disk DEBUG is quite a bit more powerful than T-BUG. Probably one of its most useful features is the ability to set **multiple breakpoints**. T-BUG allowed only one breakpoint, making it fairly easy for the program to take a path to oblivion. In addition to breakpointing, DEBUG allows for modifying memory or registers, displaying areas of memory in ASCII or hexadecimal, and **single-stepping** the program one instruction at a time. These functions are described in the TRSDOS and Disk BASIC manual, and we'll be giving you some more examples of their use here.

```
                 Hints and Kinks 4-7
               Recap of DEBUG Commands

A                Show display in ASCII
C                Single step instruction,
                   CALLs in full
Daaaa<space>     Display from location aaaa
Gaaaa(,bbbb      Execute at location aaaa with
  (,cccc))         optional breakpoints
                   bbbb,cccc,. . .
H                Display in hexdecimal
I                Single step instruction
M(aaaa)<space>   Set modification address to aaaa
                   and display data
Rrp dddd         Load register pair rp with dddd
  <space>
S                Set display to full screen
U                Set dynamic display update mode
X                Cancel command, set display to
                   register mode
;                Increment memory display to next
                   block
-                Decrement memory display to last
                   block
```

The debugging process with DEBUG is very similar to the one described under T-BUG use. The approach is again to focus in on failing areas of the program by a type of **"binary search."**

Start execution of your program by entering a G(o) command to DEBUG, with one or more breakpoints specified. If the program gets to the breakpoint, go back and look for "tracks" of proper program operation — flags set properly, variables and buffers with correct values, and the like. If errors are found, concentrate on the failing area by setting a breakpoint prior to the error and observing results. If the program doesn't get to the breakpoint, hit BREAK to get back to DEBUG and then try another execution with an earlier breakpoint.

The DUMP command can be used to advantage in **patching** programs. (If you skipped over the discussion of patching under T-BUG, you may want to read it to get familiar with the technique.) When your program has been patched, you can save it on disk by doing something similar to:

```
DUMP NAME (START=X'8000',END=X'9A45').
```

This creates a new file on disk with name, NAME, and extension, /CIM (Core Image Module). You can now load this patched file eliminating a tedious patching operation each time the program "blows up."

Hints and Kinks 4-8

How Do You Know Where A Relocatable Program Is?

One of the problems in link loading is knowing where a location is. Subroutines can be found by looking at the load map listing of all globals (loader M command). Record this information after the load.

It's also easy to put in global symbols, not only for subroutines but also for other selected points in the program. These will then be displayed on the load map, and you'll avoid some hexadecimal arithmetic.

A second trick is to use the -P command to load each module at convenient locations such as 8000H, 8200H, 8400H, etc. This makes the hexadecimal arithmetic a little easier.

To use DUMP, G(o) to location 402DH. This reboots TRSDOS but does not affect any of the user memory area. Now enter the DUMP command with the proper memory limits specified. By the way, if you've patched **outside** your program area, be sure to include that area by altering the START or END address accordingly.

The DUMP command is best used with a "clean" program. When you have found an error and established the proper patch, reload the program; **do not execute** and make the patches and then DUMP the patched, unexecuted program. This will eliminate strange results caused by patching an altered program.

However, if you want, you can also use the DUMP to save the state of any program at any given time during execution. This enables you to load in the program and easily reconstruct the conditions that caused the blowup.

Interfacing Assembly-Language and BASIC Programs

Level II and Disk BASIC both have the capability of calling assembly-language programs while in the middle of BASIC program execution. This means that **time-critical** parts of the BASIC program can be coded in assembly language to speed up program execution, or that BASIC can be used for operations which are harder to code in assembly language, such as string input, report display, and "number-crunching."

Level II USR Calls

The Level II procedure for calling an assembly-language program is as follows:

1. POKE the address of the assembly-language routine to be called into locations 16526 and 16527. The two bytes represent the least significant and most significant bytes of the address in standard Z-80 address format.
2. Any time a call to the assembly-language routine is required, call the routine by a statement such as `A=USR(M)`.

The USR function uses two **arguments**, A and M. These can be "dummies." For example, a call to the assembly-language routine can be made by `A=USR(0)` with variable A being ignored. The USR function simply causes the BASIC interpreter to pick up the address from locations 16526 and 16527 and execute a CALL to that address. The assembly-language routine must have a RET at the end to cause a return back to the interpreter at a point after the CALL. Figure 4-10 shows a typical Level II USR sequence for a short assembly-language routine located at location C000H.

```
100 REM CALL WRITE BOOK ROUTINE AT C000H
200 REM FIRST POKE LEAST SIGNIFICANT BYTE OF ADDRESS
300 POKE 16526,0
400 REM NEXT POKE MOST SIGNIFICANT BYTE OF ADDRESS
500 POKE 16527,192
600 REM NOW CALL ROUTINE BY USR FUNCTION
700 A=USR(M)
800 REM RETURN AFTER USR ROUTINE HERE
```

Figure 4-10. Level II USR Sequence

You can make calls to more than one assembly-language routine easily by setting up 16526,7 with the proper address just prior to the USR call. You can use as many assembly-language routines as you need by preceding each call by a POKE of the address into 16526,7. Of course, if you use only one assembly-language routine, you only have to set up 16526,7 once at the beginning of the BASIC program.

The M argument allows the BASIC program to pass one 16-bit argument to the assembly-language routine. The calling sequence is the same except that M must be a variable (or expression) that is equated to -32768 to 32767. When the assembly-language routine is entered, it must pick up the argument by performing a CALL to ROM at 0A7FH. This CALL passes the 16-bit argument back in the HL register pair.

If you wanted to pass the argument back the other way — in this case, pass a 16-bit value in HL back to the BASIC program — you would make a JP 0A9AH at the end of the assembly-language program rather than a RET. Figure 4-11 illustrates the passing of a single argument back and forth; here, a variable passed to a shift routine. The shift routine shifts the value three bits left and passes back the result. The BASIC program then displays the result on the screen.

```
C000          00100           ORG     0C000H
              00110 ;SHIFT ROUTINE. SHIFTS CONTENTS OF HL LEFT 3 BITS AND
              00120 ;PASSES BACK THE RESULT
C000 CD7F0A   00130 SHIFT     CALL    0A7FH     ;PICK UP ARGUMENT
C003 29       00140           ADD     HL,HL     ;SHIFT ONE BIT POSITION
C004 29       00150           ADD     HL,HL     ;SHIFT TWO BIT POSITIONS
C005 29       00160           ADD     HL,HL     ;SHIFT THREE BIT POSITIONS
C006 C39A0A   00170           JP      0A9AH     ;RETURN ARGUMENT
0000          00180           END
00000 TOTAL ERRORS
```

```
100 REM TEST PROGRAM FOR SHIFT ASSEMBLY LANGUAGE PROGRAM
200 POKE 16526,0
300 POKE 16527,192
400 INPUT "VALUE=";M
500 A=USR(M)
600 PRINT "ORIGINAL=";M,"SHIFTED RESULT=";A
700 GOTO 400
```

Figure 4-11. Assembly-Language/BASIC Argument Passing

Disk BASIC USR Calls

The procedure for interfacing assembly-language programs to Disk BASIC is much the same as that for Level II BASIC. The chief difference is that more than one USR call is permitted. The format of the Disk BASIC USR call is `X=USRn(M)` where *n* is 0 through 9, permitting ten separate USR calls. Each USR call is first defined by a DEFUSR function of the form `DEFUSR=&HXXXX` where XXXX is the address of the assembly-language subroutine in hexadecimal format.

The CALL to 0A7FH to pass an argument from BASIC to the assembly-language routine and the JP to 0A9AH to pass an argument back to BASIC is the same as in Level II BASIC. The sequence for the same assembly-language program as above is shown for Disk BASIC in Figure 4-12.

```
100 REM TEST PROGRAM FOR SHIFT ASSEMBLY-LANGUAGE PROGRAM
200 DEFUSR0=&HC000
300 INPUT "VALUE=";M
400 A=USR0(M)
500 PRINT "ORIGINAL=";M,"SHIFTED RESULT=";A
600 GOTO 300
```

Figure 4-12. Disk BASIC USR Sequence

Handling Multiple Arguments

Many times it is necessary to pass more than one argument between Level II or Disk BASIC and an assembly-language routine. Since the USR call permits you to pass only one 16-bit value, how can you pass multiple arguments? There are a number of different ways to accomplish this.

Packing Arguments

First of all, arguments may be **packed** into 16 bits. If, for example, an x and y screen position must be specified, then two 8-bit fields, one in H and one in L may be used. If you need four arguments and they can be held in four-bit values, then you can use 4 four-bit fields in HL.

Figure 4-13 shows the technique of storing data in a common memory area in an example that SETs a series of points in the assembly-language routine. The points are defined in a table starting at location 32700 and terminated with a -1.

```
C000           00100          ORG     0C000H
               00110 ;**********************************************************
               00120 ; ROUTINE TO PUT AN "O" IN DISPLAY.
               00130 ; ENTRY:  BASIC HL LINKAGE POINTS TO TABLE OF ADDRESSES
               00140 ;         EACH TABLE ENTRY IS SCREEN CHARACTER POSITION
               00150 ;         TO BE SET. TABLE TERMINATED BY -1.
               00160 ;**********************************************************
               00170 ;
C000 CD7F0A    00180 SETO     CALL    0A7FH           ;GET ADDRESS OF TABLE
C003 E5        00190          PUSH    HL              ;NOW IN STACK
C004 DDE1      00200          POP     IX              ;NOW IN IX
C006 DD6601    00210 SETO10   LD      H,(IX+1)          ;MS BYTE
C009 DD6E00    00220          LD      L,(IX+0)          ;LS BYTE
C00C 7C        00230          LD      A,H               ;GET MS BYTE
C00D FEFF      00240          CP      0FFH              ;TEST FOR END
C00F C8        00250          RET     Z                 ;GO IF DONE
C010 3E4F      00260          LD      A,'O'             ;O FOR STORE
C012 77        00270          LD      (HL),A            ;STORE O
C013 DD23      00280          INC     IX                ;BUMP PNTR
C015 DD23      00290          INC     IX                ;TWICE
C017 18ED      00300          JR      SETO10            ;CONTINUE
0000           00310          END
00000 TOTAL ERRORS
```

```
100 'BASIC DRIVER TO STORE OS IN DISPLAY
150 CLS
200 POKE 32700,32:'POKE 3E20H
300 POKE 32701,62
400 POKE 32702,34:'POKE 3E22H
500 POKE 32703,62
600 POKE 32704,36:'POKE 3E24H
700 POKE 32705,62
800 POKE 32706,255:'POKE -1
900 POKE 32707,255
1000 DEFUSR0=&HC000
1100 B=32700:'START OF POKED DATA
1200 A=USR0(B):'MAKE CALL TO AL ROUTINE
1300 GOTO 1300:'LOOP HERE TO RETAIN DISPLAY
```

32700	32	3E20H
1	62	
2	34	3E22H
3	62	
4	36	3E24H
5	62	
6	255	FFFFH
7	255	

Figure 4-13. Passing Multiple Arguments

Passing The Arguments

A second method is to pass the arguments between BASIC and assembly language by using a common memory area. Arguments can be POKEd into the memory area in BASIC or obtained from the area by PEEKs. The assembly-language code, of course, can easily load or store the values. You can define this common memory area as part of the assembly-language program area itself or in a convenient location outside of the assembly-language program.

Another way to pass the arguments is to use the contents of HL to define the address of a **parameter list**. Arguments can then be picked up or stored in the list by the assembly-language program. Of course, the address of the parameter list can be passed back from the assembly-language routine to BASIC. The parameter list may be established in BASIC by using a **dummy string** or **array**. You can then determine the address of the array or string by a VARPTR function in BASIC and pass it to the assembly-language routine.

In the next chapter we'll look at a special case of interfacing assembly-language routines: **embedded** machine-language code in BASIC programs.

Chapter 5
Embedded Machine Code in BASIC

In this chapter, we'll discuss special techniques for **embedding** assembly-language code in BASIC programs. By this technique, short assembly-language routines can be included as DATA statements or strings in the BASIC program. One advantage to this is the ability to store, load, and execute the BASIC and assembly-language code as one file. Additionally, you don't have to compute the addresses of the assembly-language routines manually and include them in the BASIC code — they are automatically found by the BASIC program itself.

The drawbacks to this method are: (1) it works best with short code segments that are **relocatable**; (2) it requires some conversion of the hexadecimal machine code values into decimal values to include in the BASIC code.

Relocatable Code

Before we show several methods for embedding machine code, let's first talk about **relocatability**. As it applies to embedding, "relocatability" does not have the same meaning here as when we used the term with the Disk Assembler and Loader. The Loader relocates code by adding a **relocation bias** to addresses in instructions in order to obtain the correct address for the instruction no matter where in memory the instructions are moved. However, here we're talking about instructions whose machine code form is constant no matter where in memory the instruction is. These instructions would **not** have the relocation bias added to them during the loading process.

As a rule of thumb, any instruction that does not contain a memory address is relocatable. For example,

```
LD      A,B
AND     07FH
LDIR
LD      A,(HL)
```

are all instruction types which have the same machine code form no matter where in memory they're located.

Instructions that contain addresses, however, may or may not be relocatable. If the address reference is to somewhere inside the program and if the program is moved around in memory, the address in the instruction **should** be constantly changing depending upon the area of memory in which the program is loaded, as shown in Figure 5-1.

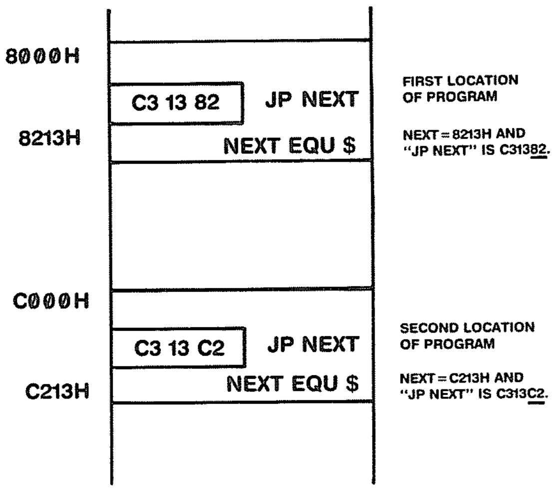


Figure 5-1. Non-Relocatable Direct Address Instruction

If the address in the instruction refers to a memory location that is not inside the program, such as a ROM subroutine or a fixed buffer in high memory, then even though the program may move, the instruction containing the address is relocatable, as shown in Figure 5-2.

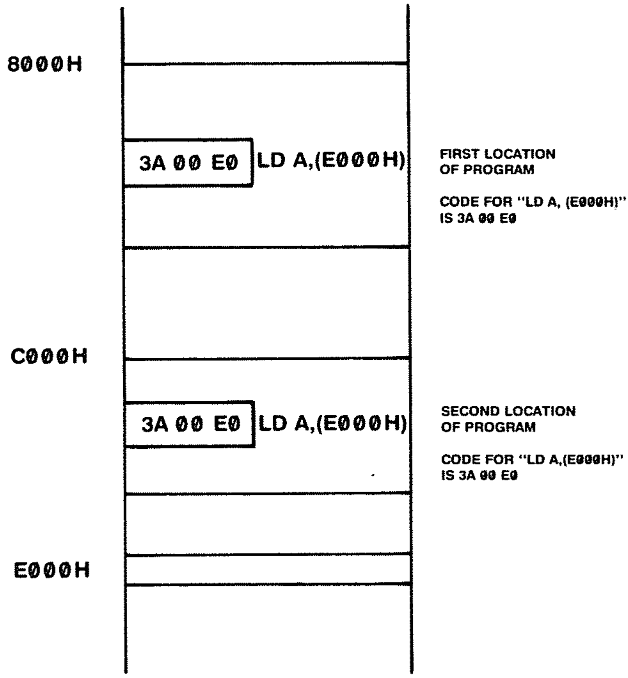


Figure 5-2. Relocatable Direct Address Instruction

Some examples of instructions that are **not** relocatable are

```
LD  HL, TABLE
```
(where TABLE is inside program)

```
LD  A, (FLAGS)
```
(where FLAGS is in program)

```
JP  Z,NEXT
```
(where NEXT is inside program)

Jump instructions that jump to locations inside a program are always non-relocatable, unless they are "Jump Relative" instructions or `DJNZ` instructions. The reason for this (check out the instruction formats in Appendix II) is that the standard `JP` type instructions always have an address in bytes 2 and 3 of the instruction. This address, of course, should change as the program is moved around in memory.

Hints and Kinks 5-1
JR and DJNZ Instructions

The format for the JR and DJNZ instructions is

BYTE Ø	BYTE 1	
OF CODE	E-2	DISPLACEMENT

the opcode is fixed:
JR=18H, JR C=1CH, JR NC=30H,
JR Z=28H, JR NZ=20H, DJNZ=10H

The second byte is computed by counting forward or back from the instruction following the JR or DJNZ. Since the JR or DJNZ is two bytes long, a jump to the following instruction would have a displacement in the second byte of 0; a jump to the JR or DJNZ itself would have a displacement of FEH or −2!

If you're weak on hex math, count back one byte at a time from the relative instruction like this: FDH, FC, FB, FA, F9, etc. to get the second byte. Best not to try this with an instruction 102 bytes back!

The JR and DJNZ instructions, however, use a **displacement** field of one byte in the instruction. This displacement field is used to compute the jump address by addition to the **current contents of the program counter**. Therefore, the jump address (or **effective address**) is always relative to the current location of the instruction. If the jump is to an instruction 6 bytes forward, the format of a JR NEXT is always 18 04; if the jump is to an instruction 6 bytes back, the format of a JR NEXT is always 18 F8. (Remember, the program counter points to the next instruction after the JR by the time it is added to the displacement, making the displacement field a value two bytes less than the displacement from the start of the JR!)

Because the JR and DJNZ are fixed values wherever the program is relocated in memory, they're relocatable instructions you can use in place of JPs. Of course, the jump address must be within 129 bytes forward or 126 bytes back of the JR or DJNZ.

If you're using the Disk Assembler, it automatically flags in the machine code section of the listing which instructions are relocatable and which instructions are non-relocatable. A byte that is relocatable is followed by a blank, while a single quote is used after an address. If the machine code for an instruction doesn't have a quote, the instruction is relocatable.

```
                    Hints and Kinks 5-2
               Non-Relocatable Instructions

The following instructions are not relocatable
unless they reference memory locations that
are fixed and outside the program area in
question:

  Load A
  from memory       LD A,(nn)

  Store A           LD (nn),A

  Load register
  pair
  immediate         LD dd,nn
                    dd=BC,DE,HL,SP,IX,IY
  Load register
  pair memory       LD dd,(nn)
                    dd=BC,DE,HL,SP,IX,IY
  Store register
  pair              LD (nn),dd
                    dd=BC,DE,HL,SP,IX,IY
  Uncondi-
  tional jump       JP nn

  Conditional
  jump              JP cc,nn
                    cc=NZ,Z,NC,C,PO,PE,P,M
  Uncondi-
  tional CALL       CALL nn

  Conditional
  CALL              CALL cc,nn
                    cc=NZ,Z,NC,C,PO,PE,P,M
```

Embedded Machine Code
By DATA to Memory

The first method of embedding machine code is to include the code in DATA statements and then move it to a fixed location in memory. Let's see how this works on a simple program. Figure 5-3 shows the listing for a short program to scroll up the screen one line.

```
8000            00100         ORG      8000H
                00110 ;SCROLL PROGRAM. SCROLLS UP SCREEN ONE LINE. FILLS
                00120 ;LAST LINE WITH BLANKS
8000 21403C     00130 SCROLL  LD       HL,3C00H+64      ;START
8003 11003C     00140         LD       DE,3C00H         ;DESTINATION
8006 01C003     00150         LD       BC,1024-64       ;# BYTES
8009 EDB0       00160         LDIR                      ;SCROLL
800B 21C03F     00170         LD       HL,3FC0H         ;ADDRESS OF LAST LINE
800E 3E20       00180         LD       A,' '            ;BLANK CHARACTER
8010 0640       00190         LD       B,64             ;# BYTES PER LINE
8012 77         00200 LOOP    LD       (HL),A           ;STORE BLANK
8013 23         00210         INC      HL               ;BUMP POINTER
8014 10FC       00220         DJNZ     LOOP             ;GO IF NOT 54
8016 C9         00230         RET                       ;RETURN
0000            00240         END
00000 TOTAL ERRORS
```

Figure 5-3. SCROLL Program Example

The program contains all relocatable code and is meant to be called from BASIC by the techniques discussed in Chapter 4. To embed the code in a BASIC program by that method, take the following steps:

1. Assemble and debug the program completely.
2. Convert the hexadecimal values for the machine code into their decimal equivalents.
3. Put these decimal equivalents into a DATA statement in the BASIC program.
4. Move the DATA values to a predetermined area of memory at the beginning of the BASIC program.
5. Set up the USR address and call the machine code as required.

These steps have been carried out in Figure 5-4. The resulting BASIC program is a simple test of the program; two versions are shown, one for Level II and one for Disk BASIC. (Set MEMORY SIZE to 32767.)

Hints and Kinks 5-3

BASIC Address Computation

In using machine code in higher memory with BASIC—

Certain BASIC functions that operate only with 16-bit integer values don't permit "absolute" values from 32768 through 65535. An "overflow" error results. When addressing memory locations with these functions, the interpreter must be fooled into thinking the memory location value is a legitimate integer value from −32768 to +32767.

This problem only occurs for memory locations in the upper 32K of memory, 32768 through 65535. When you reference these locations, they must be in the form of the expression —

```
LOCATION-65536
```

This problem occurs for the FOR . . . TO command, DEFUSR command, PEEK command, POKE command, and others.

Example:

```
100 PRINT PEEK(60000) does not work,
but
100 PRINT PEEK(60000-65536) does.
```

```
100 REM EMBEDDED MACHINE CODE BY DATA TO MEMORY
200 FOR I=32768 TO 32790
300 READ A
400 POKE (I-65536),A
500 NEXT I
600 DEFUSR0=&H8000:'POKE 16526,0:'FOR LEVEL II
700 'POKE 16527,128:'FOR LEVEL II
800 X=USR0(0):'X=USR(0):'FOR LEVEL II
900 GOTO 800
1000 REM THESE DATA VALUES ARE THE 23 BYTES OF SCROLL IN DECIMAL
1100 DATA 33,64,60,17,0,60,1,192,3,237,176,33,192,63,62,32,6,64
1200 DATA 119,35,16,252,201
```

_ _ DISK BASIC

___ LEVEL II BASIC

Figure 5-4. DATA-to-Memory Embedded Machine Code

In this case, relocatable code was used in the SCROLL program. However, since the predetermined area of 8000H was used, the code **could** have included non-relocatable instructions, as long as the assembly was performed with an ORG of 8000H.

This method can be used for any size of assembly-language program. It does get quite tedious, however, to convert the hexadecimal values into decimal for large amounts of code. A few other disadvantages include the necessity of fixing the memory area for the assembly-language code and the possibility of having to "merge" the DATA table with other DATA values in the BASIC program.

Embedded Machine Code By CHR$ Strings

This method of embedding machine-language code into a BASIC program uses the CHR$ function of Level II or Disk BASIC to construct a string of machine-language values. You can then find the location of the string by the VARPTR function and make a call by the USR function.

You must carry out the same preliminary steps as in the first method: the program must be assembled, debugged, and converted to decimal values. The decimal values must then be used in a BASIC character string such as ZZ$=CHR$(xx)+CHR$(xx)+CHR$(xx)+ . . . where "xx" represents the machine code decimal values.

A BASIC program that calls SCROLL and uses this method is shown in Figure 5-5. The VARPTR function is used to find the location of string ZZ$. Remember from the Level II manual that VARPTR points to a parameter block for the string. The parameter block has the form shown in Figure 5-6. The second and third bytes represent the actual location of the string.

```
50 REM CHR$ EMBEDDED MACHINE CODE
100 A$=CHR$(33)+CHR$(64)+CHR$(60)+CHR$(17)+CHR$(0)+CHR$(60)
+CHR$(1)+CHR$(192)+CHR$(3)+CHR$(237)+CHR$(176)+CHR$(33)
+CHR$(192)+CHR$(63)+CHR$(62)+CHR$(32)+CHR$(6)+CHR$(64)+CHR$(119)
+CHR$(35)+CHR$(16)+CHR$(252)+CHR$(201)
200 B=VARPTR(A$)
300 C=PEEK(B+2)*256+PEEK(B+1):'POKE 16526,PEEK(B+1):'FOR LEVEL II
400 IF C>32767 THEN C=C-65536:'POKE 16527,PEEK(B+2):'FOR LEVEL II
500 DEFUSR0=C:'REM FOR LEVEL II
600 X=USR0(0):'X=USR(0):'FOR LEVEL II
700 GOTO 600
```

--- DISK BASIC

——— LEVEL II BASIC

Figure 5-5. CHR$ Embedded Machine Code

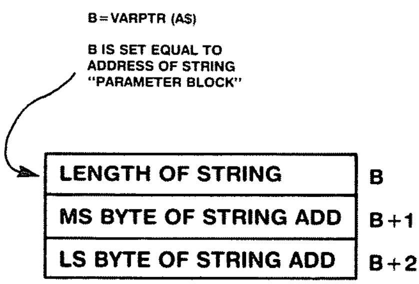


Figure 5-6. String Parameter Block

Because the string is located in the string storage area high in memory, there is a major problem with this method! Both Level II and Disk BASIC go through a "garbage collection" process when they run out of string space. As you recall, a string storage area may be CLEARed initially in the BASIC program.

As operations are performed on strings in the BASIC program, the interpreter uses more and more of the string storage area. The interpreter is **sloppy** and does not clean up old forms of the strings as new string storage space is allocated. However, after many string operations, the interpreter may run out of string storage space and have to go back to "clean up" the string storage area. When this is done, the memory for the strings are reallocated and the string addresses change.

This garbage collection may never happen in the BASIC program you interface to the assembly-language code. It is a function of memory size, string storage area, and string operations. However, because it **can**, it is always best to find the address of the CHR$ string **immediately before** the machine code in the string is called.

This method works quite well for short assembly-language programs, but one of the limitations is in the length of the BASIC line defining the string. To avoid the problem, several strings may be concatenated to produce a larger string, however, a more stringent limitation is the size of the string itself, which cannot be greater than 255 characters. You must use all relocatable code in this method.

DATA Values and Dummy Strings

A third way to embed assembly-language code in BASIC programs is to use a table of DATA values representing machine code as in the first method but to move those values to a "dummy" string. The advantage of this method over the first is that the BASIC program will perform the address calculation automatically by the VARPTR function. The disadvantages are the string size limitation and the requirement of all relocatable code.

The "dummy string" here is a BASIC string statement of the form `ZZ$="THIS IS A DUMMY STRING. . . ."` The number of bytes in the string should be equal to or greater than the number of bytes in the assembly-language program. The text is irrelevant. The location of the string is found by VARPTR, and the bytes from the DATA table replace the characters of the dummy string.

In this case, reshuffling strings is no problem during the garbage collection process since the string location is in the BASIC statement itself and the interpreter records the address of the text as the string address. Figure 5-7 illustrates use of this method with the SCROLL program.

Hints and Kinks 5–4
Using Dummy Strings

There is a slight problem in using a dummy string as a storage area for machine code. BASIC uses a zero (null) byte to mark the end of a text line. If a zero is stored for a byte of an instruction, the BASIC interpreter will truncate the line during edits, lists, and other operations.

You can store zeroes and use the embedded machine language normally. However, don't attempt to edit such a program <u>after execution</u>. Edit the BASIC program before it's run and save it on cassette or disk before execution.

If a program is listed after execution, strange things will occur as the interpreter encounters the modified dummy string. Valid machine codes may not be valid display characters!

```
100 REM DATA TO DUMMY STRING EMBEDDED MACHINE CODE
200 A$="THIS IS A DUMMY STRING!"
300 B=VARPTR(A$)
400 C=PEEK(B+2)*256+PEEK(B+1)
500 FOR I=C TO C+22
600 READ A
700 IF I>32767 THEN POKE I-65536,A ELSE POKE I,A
800 NEXT I
900 IF C>32767 THEN DEFUSR0=C-65536 ELSE DEFUSR0=C
1000 'POKE 16526,PEEK(B+1):'FOR LEVEL II
1100 'POKE 16527,PEEK(B+2):'FOR LEVEL II
1200 X=USR0(0):'X=USR(0):'FOR LEVEL II
1300 GOTO 1200
1400 DATA 33,64,60,17,0,60,1,192,3,237,176,33,192,63,62,32,6,64
1500 DATA 119,35,16,252,201
```

_ _ _ DISK BASIC

_____ LEVEL II BASIC

Figure 5-7. DATA to Dummy String Embedded Machine Code

DATA Values and Array Storage

By taking advantage of the contiguous nature of arrays, you'll have a fourth method of storing machine code within the BASIC program. Arrays in BASIC are allocated by DIM statements. Once you've established the array, the array is treated as a block of data. Any number of bytes can be used in the array. The array location remains fixed as long as new variables are not referenced; its location can always be found by VARPTR.

Hints and Kinks 5-5
Array Storage of Machine Code

If an integer array (such as A%) is used for storage of machine code data, remember that the elements of the array are two bytes long and that the data is stored in normal array operations in standard Z-80 address format. Storing 0 through 4 in array A%(0)-A%(4), for example, produces

LOC+0	Ø	A%(Ø) = Ø
+1	Ø	
+2	1	A%(1) = 1
+3	Ø	
+4	2	A%(2) = 2
+5	Ø	
+6	3	A%(3) = 3
+7	Ø	
+8	4	A%(4) = 4
+9	Ø	

Take this into account when storing machine code data by setting an array element to two machine code bytes —

A%(4) = (machine code byte *n*)*256 + (machine code byte *n*+1)

In this method, the DATA values representing the machine code are moved to a dummy array, the VARPTR function is used to find the address of the array, and the DATA values are moved initially to the array. To make storage into the array more straightforward, the array should preferably be an array of integer values, since they occupy two bytes per element. The elements of an array are **contiguous**, that is, they occupy successive locations in memory, and the DATA values can be stored with no problem. Figure 5-8 shows this method.

```
100 REM DATA TO ARRAY EMBEDDED MACHINE CODE
200 DIM A%(12)
300 B=VARPTR(A%(0))
400 FOR I=0 TO 11
500 READ C,D
550 E=D*256+C
600 IF E>32767 THEN A%(I)=E-65536 ELSE A%(I)=E
700 NEXT I
800 IF B>32767 THEN DEFUSR0=B-65536 ELSE DEFUSR0=B
900 'POKE 16526,B-INT(B/256)*256:'FOR LEVEL II
1000 'POKE 16527,INT(B/256):'FOR LEVEL II
1100 X=USR0(0):'X=USR(0):'FOR LEVEL II
1200 GOTO 1100
1300 DATA 33,64,60,17,0,60,1,192,3,237,176,33,192,63,62,32,6,64
1400 DATA 119,35,16,252,201,0
```

— — DISK BASIC

—— LEVEL II BASIC

NOTE DUMMY LAST VALUE FOR EVEN #

Figure 5-8. DATA to Array Embedded Machine Code

Passing Arguments and Multiple Subroutines

In the above examples, we illustrated the techniques with a very simple subroutine. However, you can incorporate all of the material presented in Chapter 4, into these embedded machine-language methods including passing multiple arguments and multiple subroutines back and forth.

Using embedded machine-language code can be a valuable tool since it blends the easy-to-use string handling, display and print formatting, and number-crunching abilities of BASIC with the high-speed of assembly language. It's probably well worth your effort to take portions of BASIC programs that are notoriously slow and rewrite them into short embedded assembly-language routines — code such as sorts, searchs, and graphics. Watch for further examples of embedded machine-language code in later chapters.

SECTION II

Assembly-Language Techniques

Chapter Six

Number Crunching

This chapter looks at what is commonly called "number crunching." This term refers to operations on data that is primarily numeric — adds, subtracts, multiplies, divides, trigonometric functions, integration, differentiation, and so forth. We only have room here to discuss basic number-crunching operations that come up frequently in non-math oriented programs — such things as adds, subtracts, multiplies, divides, and random number generation.

Addition and Subtraction

Addition and subtraction operations are built into the instruction set of the Z-80. All other math-related operations have to be built out of adds, subtracts, compares, and shifts.

Eight-Bit Adds

The basic adds and subtracts use two 8-bit operands. One of the operands is in the A register, while the other operand is either in the instruction itself (immediate addressing), in another CPU register (register addressing), in a memory location pointed to by the HL register (register indirect addressing), or in a memory register pointed to by the **effective address** of an indexed-type instruction.

Suppose we have 23H in the A register, 45H in the B register, 8000H in the HL register pair, 8040H in IX, 7FF0H in IY, and 45H in memory location 8000H. All of these adds accomplish the same thing, adding a 45H to the 23H in the A register, putting the result of 68H in the A register and setting the flags.

1.	ADD	A,45H	Immediate add of 45H to 23H in A
2.	ADD	A,B	Add of 45H in B to 23H in A
3.	ADD	A,(HL)	Add of 45H in 8000H to 23H in A
4.	ADD	A,(IX-40H)	Add of 45H in 8000H to 23H in A
5.	ADD	A,(IY+16)	Add of 45H in 8000H to 23H in A

Generally, in **arithmetic** instructions, all flags are affected. But how are the flags set? The C flag is set when there is a carry from bit 7 of the result, which does not occur here. The Z flag is set when the result is zero, which is not the case for the result of 68H. The P/V flag is the dual-purpose flag used either for **parity** or **overflow**. In arithmetic instructions, P/V is always used for overflow. There is no overflow here, and P/V is not set. The sign flag is set when the result has a one bit in bit 7, indicating that the result is negative (not the case here). The N flag is not set here since this is an add, and the half-carry flag is not set here because there is no carry from bit 3 of the result. (N and H are hardly ever used by the programmer, and there are no conditional branches on them — so ignore them!)

Hints and Kinks 6-1
P/V Flag

When we talk about parity in the TRS-80, we're not discussing farm prices. Parity refers to a count of the number of one bits in (usually) an 8-bit location or register. Parity is primarily used to check the data on I/O operations. Since I/O devices lose data more frequently than cpu/memory actions, a check on the number of one bits is a rough check on the validity of the data.

In the Z-80, the P/V flag is set for even parity. If the number of one bits is even after an instruction affecting parity, then the P/V flag is set; otherwise it is reset. Instructions that set parity in this fashion are IN R,(C), most shifts and rotates, and the logical instructions.

The P/V flag is used for overflow when (most) arithmetic instructions are executed. It can be tested by a conditional jump to ascertain whether or not the result was too large to be held as a two's complement number, either for 8-bit or (some) 16-bit instructions. Overflow occurs for 8 bits when a result is below -128 or above 127 and for 16 bits when a result is below -32768 or above 32767.

Many times an add will be used without any other following action. Sometimes, however, an add will be followed by a conditional branch of some type, as in the example from the MEMORY Subroutine from Chapter 14 shown in Figure 6-1. An ADD A,(HL) is done to add the contents of the memory location pointed to by HL to the contents of the A register. If the result is zero, a conditional jump is made to MEM008 to set the count to zero (!); if the result is positive (P), a JP is made to MEM010.

```
0013'   F1          00320           POP     AF
0014'   F5          00330           PUSH    AF
0015'   86          00340           ADD     A,(HL)          ;ADJUST COUNT
0016'   CA 001C'    00350           JP      Z,MEM008        ;GO IF ZERO
0019'   F2 001D'    00360           JP      P,MEM010        ;GO IF POSITIVE
001C'   AF          00370  MEM008:  XOR     A               ;COUNT OF 0
001D'   FE 64       00380  MEM010:  CP      100             ;TEST FOR LT 100
001F'   FA 0024'    00390           JP      M,MEM020        ;GO IF LT 100
0022'   3E 63       00400           LD      A,99            ;MAX COUNT
0024'   77          00410  MEM020:  LD      (HL),A          ;STORE COUNT
```

Figure 6-1. Arithmetic Followed by Conditional Branch

Many times programmers get confused about using conditional branches on flags. Much of the confusion is about when the flags get set and reset. **Unless an instruction description specifically says that the flags are affected by the instruction, the flags remain as they are!** This means that you can use flags from several instructions back for conditional branching as long as you haven't used instructions that affect the flags **in question**. Many times, however, you will use a conditional JP as the next instruction after the arithmetic instruction.

Eight-Bit Subtracts and Compares

Eight-bit subtract instructions operate in the same addressing modes as the ADD instruction. The flags are set in almost identical fashion, with a few exceptions. The SUB sets the C flag if there is **no borrow**. This means, in fact, that the C flag is reset if there's a carry out of bit 7 — just the reverse of the ADD!

Hints and Kinks 6–2
Carries

When is a carry set, and when is it reset on an add, subtract, or compare? Here's how you can find out manually: If an add is being performed, add the two operands in binary; any carry bit out of the high order will simply set the carry.

On a subtract or compare, convert the two operands to binary. Now take the <u>two's complement</u> of the subtrahend (the one to be subtracted). Now add the operands. <u>Complement</u> the state of the carry. The result will be the state of the carry after a subtract or compare. After running through several thousand test cases, we discovered that the carry was set on a subtract or compare if a larger <u>unsigned</u> number was subtracted from a smaller <u>unsigned</u> number. What about <u>signed</u> numbers? This will be left as an exercise for those readers who are not faint of heart

. . . .

```
                ØØØØØØ11   3
                ØØØØØØ1Ø   -(+2)
           CY   1111111Ø   2'S COMP
           1 <- ØØØØØØØ1   +1 RESULT
           |
           Ø

      ØØØØØØ1Ø   2                    1111111Ø   254
      ØØØØØØ11   -(+3)                11111111   -(+255)
 CY   111111Ø1   2'S COMP        CY   ØØØØØØØ1   2'S COMP
 Ø <- 11111111   -1 RESULT       Ø <- 11111111   -1 RESULT
 |                               |
 1                               1
```

You would think that adding -5 to 10 in A would be the same as subtracting 5 from 10 in A. However, it isn't, as the C is set for the ADD and reset for the SUB. (Also, the N and H flag logic is different for the ADD and SUB.) The result in A is still the same.

```
LD    A,10     ;10 TO A
ADD   A,-5     ;10 PLUS -5 TO A
JP    C,OUT    ;THIS JUMP IS MADE
.
.
.
LD    A,10     ;10 TO A
SUB   A,5      ;10 MINUS 5 TO A
JP    C,OUT    ;THIS JUMP IS NOT MADE
```

Compares operate in identical fashion to subtracts, except that the result is thrown away and is only used to set or reset the flags. Figure 6-2 shows use of the SUB and CP from the DECBIN subroutine of Chapter 13.

JUMP IF A NOW HOLDS MINUS NUMBER (IF A <u>WAS</u> LESS THAN 30H)

JUMP IF (A) = >10

```
85A3 7E        09160          LD    A,(HL)      ;GET CHARACTER
85A4 D630      09170          SUB   30H         ;CONVERT
85A6 FAB685    09180          JP    M,DEC070    ;GO IF LT "0"
85A9 FE0A      09190          CP    10          ;TEST FOR GT "9"
85AB F2B685    09200          JP    P,DEC070    ;GO IF GT "9"
85AE 5F        09210          LD    E,A         ;NOW IN E
85AF 1600      09220          LD    D,0         ;NOW IN DE
85B1 DD19      09230          ADD   IX,DE       ;MERGE
85B3 23        09240          INC   HL
85B4 10E2      09250          DJNZ  DEC040      ;GO IF MORE
85B6 78        09260 DEC070   LD    A,B         ;COUNT TO A
```

Figure 6-2. Use of SUB and CP

Sixteen-Bit Adds and Subtracts

Sixteen bit adds and subtracts can use either HL, IX, or IY as **16-bit accumulators** similar to the A register. However, the number of addressing modes that can be used for register pairs is limited to one, the one associated with adding another register pair to HL, IX, or IY. Also, while you can do an ADD or ADC to HL, IX, or IY, you aren't allowed a SUB or SBC for IX or IY.

Another difference in the 16-bit arithmetic instructions is that the flags **may** or **may not** be affected, according to the instruction. ADD HL,BC, for example, does not affect Z, P/V, or S, while ADC HL,BC **does** affect these flags. To see which flags are affected keep one eye on Appendix II while using these instructions with conditional JPs.

Figure 6-3 shows you a trick to use with the C flag. The ADD HL,DE does **not** affect the Z flag, but does affect the C flag. In this subroutine, the programmer did a decrement on the contents of HL by loading DE with -1 and performing the ADD HL,DE. The C flag will be set as long as HL is positive or zero, but will be reset when HL reaches -1.

```
                  00100          TITLE   DELASR
                  00110          ENTRY   DELAY
                  00120 ;************************************************************
                  00130 ;*               DELAY SUBROUTINE                           *
                  00140 ;*  DELAYS 1 TO 65536 MILLISECONDS.                         *
                  00150 ;*  ENTRY: (HL)=DELAY COUNT IN MILLISECONDS                 *
                  00160 ;*         0=65536                                          *
                  00170 ;*  ALL REGISTERS SAVED                                     *
                  00180 ;************************************************************
                  00190 ;
0000'   C5        00200 DELAY:   PUSH    BC              ;SAVE REGISTERS
0001'   D5        00210          PUSH    DE
0002'   E5        00220          PUSH    HL
0003'   11 FFFF   00230          LD      DE,-1           ;-1 FOR DECREMENT
0006'   2B        00240          DEC     HL              ;ADJUST FOR "JP NC"
0007'   06 83     00250 DEL010:  LD      B,131             ;INNER LOOP COUNT
0009'   10 FE     00260 DEL020:  DJNZ    DEL020              ;LOOP FOR DELAY
000B'   19        00270          ADD     HL,DE             ;DECREMENT OUTER LOOP CNT
000C'   DA 0007'  00280          JP      C,DEL010          ;CONTINUE
000F'   E1        00290          POP     HL              ;RESTORE REGISTERS
0010'   D1        00300          POP     DE
0011'   C1        00310          POP     BC
0012'   C9        00320          RET                     ;RETURN
                  00330          END
```

Figure 6-3. Using the C Flag for ADD HL,XX

Both the 8-bit and 16-bit adds and subtracts have instructions that add or subtract the carry (or borrow). ADC A,23H, for example, adds not only 23H to the contents of A, but also the current state of the carry. SBC HL,BC subtracts not only BC from HL, but also subtracts the carry.

These instructions are used primarily for **multiple-precision** operations involving more than 8 or 16 bits of data. The carry or borrow must be **propagated** from the lower order to the higher order byte or "word."

Hints and Kinks 6-3
Multiple-Precision Operations

How far can multiple-precision operations be carried out? It's entirely possible to implement arithmetic that works with 100 bytes of precision. The question is, is it necessary? To get a rough idea of the number of decimal digits that can be contained in any size binary number, take the number of bits and divide by 3.5 – 8 bits would be 2.3, 16 bits would be 4.5, 32 bits would be 9.1, and so forth (4.5, for example, would be somewhere between four and five decimal digits). You can see it doesn't take too many bits before we can express ten or twenty decimal digits of precision.

Aside from mathematical games or precise scientific applications, there's generally no need for large multiple-precision numbers. The compromise reached in all computers is to hold floating-point numbers with a dozen or so decimal digits precision and an exponent that represents a power of 16. This format would still call for multiple-precision operations on the ''fractional'' part, but it would be limited to operations involving approximately 3 to 6 bytes.

Even with a small number of bytes, multiple-precision multiplies and divides are tedious and slow. Rather than writing a bit-by-bit divide to work with 64 bits, many programmers would implement something like this: A four-byte number can be expressed as $AB*16^2$ + CD, where A, B, C, and D are the byte values. To multiply two such numbers in four-byte precision would involve the expansion –

```
ABCD*EFGH = AB*EF + CD*EF + AB*GH +
CD*GH
```

This would require four 16-bit multiplies, considerable multiple-precision shifting, and four multiple-precision adds! This is a lot of work for assembly-language programmers, who are notorious for looking for the easy way out!

Figure 6-4 shows a use of this in the SUB subroutine from Chapter 13. SUB subtracts the contents of a four-byte variable SEED through SEED+3 from the contents of DE,HL. DE,HL are treated as a four-byte variable. Note that initially the carry is cleared by an OR A instruction. One of the peculiarities of the Z-80 is that it has a SCF to set the carry, and a CCF to **complement** the carry, but no Reset Carry; the OR A **does** reset the C flag, however, and the contents of A remain unchanged. The next SBC is not preceded by an OR A to reset the carry, and the carry from the first SBC is subtracted from the result.

```
               08600 ;***************SUBTRACT SEED SUBROUTINE***************
               08610 ;*    SUBTRACTS FOUR BYTES OF SEED FROM (DE,HL).      *
               08620 ;*    ENTRY:  (SEED - SEED+3)=SEED #                  *
               08630 ;*            (DE,HL)=FOUR-BYTE VALUE                 *
               08640 ;*    EXIT:   (DE,HL)=RESULT OF SUBTRACT              *
               08650 ;*    ALL REGISTERS SAVED EXCEPT DE,HL                *
               08660 ;
8571 C5        08670 SUB      PUSH    BC              ;SAVE REGISTERS
8572 ED4BE385  08680          LD      BC,(SEED+2)     ;GET LS BYTE
8576 B7        08690          OR      A               ;RESET CARRY
8577 ED42      08700          SBC     HL,BC           ;SUBTRACT LS 2 BYTES
8579 EB        08710          EX      DE,HL           ;GET MS 2 BYTES
857A ED4BE185  08720          LD      BC,(SEED)       ;GET MS 2 BYTES
857E ED42      08730          SBC     HL,BC           ;SUBTRACT MS 2 BYTES AND CY
8580 EB        08740          EX      DE,HL           ;NOW ORIGINAL-SEED
8581 C1        08750          POP     BC              ;RESTORE REGISTERS
8582 C9        08760          RET                     ;RETURN
```

RESETS CARRY FOR FIRST SUBTRACT

CARRY CONTAINS RESULTS OF FIRST SUBTRACT

Figure 6-4. Four-Byte Subtract Example

Multiplies and Divides

All multiplies and divides on the Z-80 must be performed **in software**. There are a number of methods for performing software multiply and divides, ranging from successive addition and subtraction to high-speed **table lookups**. We'll discuss some of the more common ways here.

Software Multiplies

Successive Addition

The simpliest multiply is by **successive addition**, as shown in Figure 6-5 from Chapter 13. Here, the DE register contains the multiplicand and is added to HL (after HL is cleared) a number of times corresponding to the multiplier. Three adds are used to multiply by three.

```
81C9 ED53E585 02560        LD      (DOTO),DE        ;STORE DOT ON TIME
81CD 210000   02570        LD      HL,0
81D0 19       02580        ADD     HL,DE            ;FIND 3*DOTO
81D1 19       02590        ADD     HL,DE
81D2 19       02600        ADD     HL,DE
81D3 22E785   02610        LD      (DASHO),HL       ;STORE DASH ON TIME
```

Figure 6-5. Multiply by Successive Addition

A variation on this approach uses the multiplier in the B register to take advantage of the DJNZ instruction.

```
MULT LD   B,10   ;MULTIPLIER =10
     LD   DE,34  ;ARBRITRARY MULTIPLICAND
     LD   HL,0   ;CLEAR RESULT
LOOP ADD  HL,DE  ;SUCCESSIVE ADD
     DJNZ LOOP   ;GO IF NOT N TIMES
```

Shift and Add Multiply

Another type of multiply is the "shift and add." This method, as the name implies, is a combination of shifts and adds. You can factor any multiplier into a number of powers of two factors. You can multiply N by 20, for example, by adding 16*N + 4*N. Sixteen and 4 are powers of 2 that you can easily obtain by shifting.

This type of multiply works best for commonly used multipliers, like 10. The code in Figure 6-6 below, excerpted from the Decimal to Binary Conversion subroutine of Chapter 13, shows how you can use this method. The number is shifted by an ADD IX,IX, which shifts the number one bit position left. Four shifts result in N*8. The previously saved N*2 shift is then "popped" into DE and added in to give N*10.

TIMES 2 SAVED IN STACK

```
8598 DD29        09100 DEC040  ADD     IX,IX          ;INTERMEDIATE*2
859A DDE5        09110         PUSH    IX
859C DD29        09120         ADD     IX,IX          ;*4
859E DD29        09130         ADD     IX,IX          ;*8
85A0 D1          09140         POP     DE             ;*2
85A1 DD19        09150         ADD     IX,DE          ;*10
```

TIMES 2 RETRIEVED FROM STACK

Figure 6-6. Shift and Add Example

In the multiplies above, you'll have speed limitations in the successive addition case (for large multipliers, the process takes a very long time) and lack of generality in the shift and add case. For programs that perform many multiplies of different sizes of numbers, it's best to implement a bit-by-bit multiplication.

Bit-By-Bit Multiply

A bit-by-bit multiply, implemented as a subroutine, is presented in Figure 6-7. The multiplier is held in DE on entry and the multiplicand is in BC. On exit, the result is held in DE, HL, treated as a four-byte register. If the product of the multiply is less than 65536, then it is in HL alone with zero in DE.

```
8000                         ORG     8000H
                ;*************************************************************
                ;*                   SWEET 16 MULTIPLY                       *
                ;* ENTRY: (DE)=MULTIPLIER, 16 BITS UNSIGNED                  *
                ;*        (BC)=MULTIPLICAND, 16 BITS UNSIGNED                *
                ;* EXIT:  (DE,HL)=PRODUCT, 32 BITS UNSIGNED                  *
                ;*        (BC)=RETAINED                                      *
                ;*        (A)=DEVASTATED                                     *
                ;*        EVERYTHING ELSE IN REASONABLE ORDER                *
                ;* HISTORICAL NOTES: THIS MULTIPLY WAS WRITTEN WHILE         *
                ;*   PULLING TOGETHER THE FINAL PIECES OF THIS BOOK          *
                ;*   IN ABOUT 23577 MILLISECONDS.(NO, NOT THE 1ST TIME).*
                ;*************************************************************
                ;
8000 210000     SWE16   LD      HL,0                ;ZERO PRODUCT HALF
8003 3E10               LD      A,16                ;ITERATION COUNT
8005 EB         SWE010  EX      DE,HL                  ;SETUP FOR SHIFT
8006 29                 ADD     HL,HL                  ;SHIFT DE
8007 F5                 PUSH    AF                     ;SAVE POSSIBLE CARRY
8008 EB                 EX      DE,HL                  ;PUT DE BACK
8009 29                 ADD     HL,HL                  ;NOW BOTTOM HALF
800A 3001               JR      NC,SWE020              ;GO IF NO CARRY
800C 13                 INC     DE                     ;PROPOGATE
800D F1         SWE020  POP     AF                     ;GET C FROM DE
800E 3004               JR      NC,SWE030              ;GO IF NONE
8010 09                 ADD     HL,BC                  ;BIT WAS A ONE
8011 3001               JR      NC,SWE030              ;GO IF NO CARRY TO MSB
8013 13                 INC     DE                     ;PROPOGATE
8014 3D         SWE030  DEC     A                      ;DECREMENT COUNT
8015 20EE               JR      NZ,SWE010              ;GO IF MORE
8017 C9                 RET                         ;BACK TO CALLING PROG
0000                    END
00000 TOTAL ERRORS
```

Figure 6-7. Bit-by-Bit Multiply Code

The subroutine works by shifting DE, HL. The bit shifted out of DE is used to determine whether you should perform an add of BC to HL. HL holds the partial product. Sixteen shifts are performed, and each one causes either an add or no action dependent upon whether the carry is a one or zero. The action of the multiply is shown in Figure 6-8.

Hints and Kinks 6-4
Fast Multiplies

It seems that computers are constantly being pushed by the applications! By that I mean that there's a constant requirement for faster and more efficient computers to handle applications that haven't been computerized and could be, or to speed up existing applications.

One of the critical areas of application involves multiplies as they are used in all kinds of number crunching. Since we don't have a hardware multiply on the TRS-80, we have to make do with a software routine.

You can speed up multiplies considerably by making them ''in-line'' code. This method eliminates the usual 8- or 16-iteration loop by substituting 8 or 16 code segments. Multiplies of 150 microseconds for 8 bits are possible with this approach, a factor of two better than some iterative multiplies.

Other high-speed multiplies involve constructing large tables of partial results for various multiplier/multiplicand combinations.

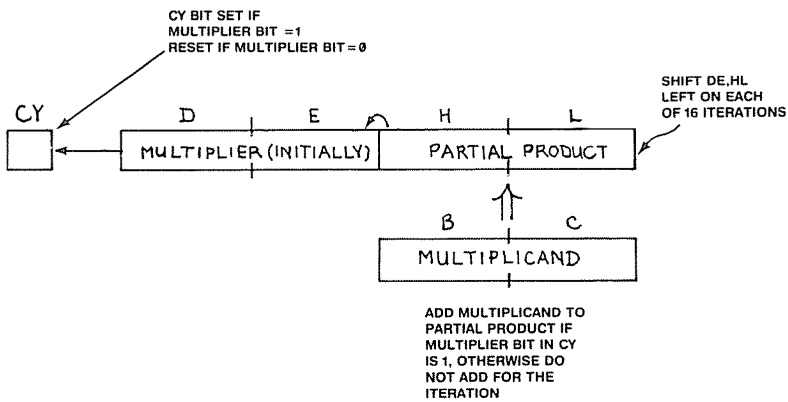
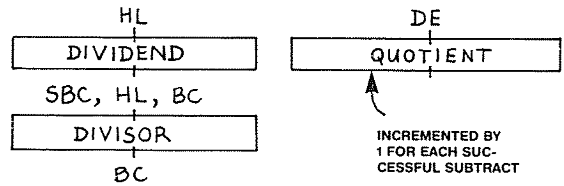


Figure 6-8. Bit-by-Bit Multiply Action

Software Divides

Successive Subtract Method

Software divides are handled in similar fashion to multiplies. Figure 6-9 shows a successive subtract method from the code of Chapter 13. It divides a 16-bit dividend in HL by a 16-bit divisor in BC. Each time the divisor is successfully subtracted from the dividend without producing a negative result (no carry), a count in DE is incremented by one. The count was initially set to -1. At the end of the divide, DE holds the count, which is the quotient. You could find the remainder by adding BC back to the residue in HL, although this is not done in the code.




```
81BA 21C003    02480          LD    HL,SPEEDF     ;1200/WPM=DOTO TIME
81BD 4F        02490          LD    C,A           ;WPM LS BYTE
81BE 0600      02500          LD    B,0           ;NOW IN BC
81C0 11FFFF    02510          LD    DE,-1         ;QUOTIENT
81C3 B7        02520 SPE015   OR    A               ;ZERO C
81C4 ED42      02530          SBC   HL,BC           ;DIVIDE BY SUCCESSIVE SUB
81C6 13        02540          INC   DE              ;BUMP QUOTIENT
81C7 30FA      02550          JR    NC,SPE015       ;GO IF NOT NEGATIVE
81C9 ED53E585  02560          LD    (DOTO),DE     ;STORE DOT ON TIME
```

Figure 6-9. Divide by Successive Subtraction

Bit-By-Bit Restoring Method

Another type of divide is implemented in the code from Chapter 14 shown in Figure 6-10. This is a bit-by-bit "restoring" divide, which divides the 16-bit dividend in HL by an 8-bit divisor in E. On exit, HL holds the quotient and BC holds a remainder. The action of the divide is shown in Figure 6-11.

```
                                TITLE   DIVISR
                                ENTRY   DIVIDE
                        ;*****************************************************
                        ;*               DIVIDE SUBROUTINE                   *
                        ;*  PERFORMS AN UNSIGNED DIVIDE OF 16 BITS BY 8-BIT  *
                        ;*  DIVISOR.                                         *
                        ;*  ENTRY:  (HL)=DIVIDEND                            *
                        ;*          (E)=DIVISOR                              *
                        ;*  EXIT:   (HL)=QUOTIENT                            *
                        ;*          (BC)=REMAINDER                           *
                        ;*  ALL REGISTERS SAVED EXCEPT HL,BC                 *
                        ;*****************************************************
                        ;
0000'   D5              DIVIDE: PUSH    DE              ;SAVE REGISTERS
0001'   DD E5                   PUSH    IX
0003'   E5                      PUSH    HL              ;DIVIDEND TO IX
0004'   DD E1                   POP     IX
0006'   21 0000                 LD      HL,0            ;ZERO HL
0009'   16 00                   LD      D,0             ;DIVISOR IN DE
000B'   06 10                   LD      B,16            ;ITERATION COUNT
000D'   29              DIV010: ADD     HL,HL             ;SHIFT 16 MS BITS
000E'   DD 29                   ADD     IX,IX             ;SHIFT 16 LS BITS
0010'   30 01                   JR      NC,DIV020         ;GO IF NO CARRY
0012'   23                      INC     HL                ;CARRY TO 16 MS BITS
0013'   DD 23           DIV020: INC     IX                ;SET Q BIT
0015'   B7                      OR      A                 ;CLEAR CARRY
0016'   ED 52                   SBC     HL,DE             ;TRY SUBTRACT
0018'   30 03                   JR      NC,DIV030         ;GO IF DIVIDE WENT
001A'   19                      ADD     HL,DE             ;RESTORE
001B'   DD 2B                   DEC     IX                ;RESET Q BIT
001D'   10 EE           DIV030: DJNZ    DIV010            ;GO IF NOT 16 ITERATIONS
001F'   E5                      PUSH    HL              ;REMAINDER
0020'   C1                      POP     BC
0021'   DD E5                   PUSH    IX              ;QUOTIENT
0023'   E1                      POP     HL
0024'   DD E1                   POP     IX              ;RESTORE REGISTERS
0026'   D1                      POP     DE
0027'   C9                      RET                     ;RETURN
                                END
```

Figure 6-10. Bit-by-Bit Divide Routine

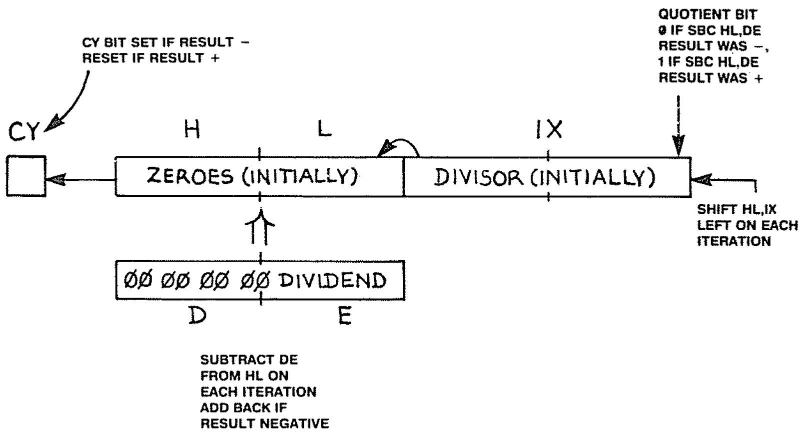


Figure 6-11. Bit-by-Bit Divide Action

Signed Vs. Unsigned Multiplies and Divides

All of the multiplies and divides we've discussed above were **unsigned**. That is, all operands were considered to be positive numbers with no sign bits. This means that the range of numbers held in a 16-bit result would be 0 through 65535, rather than -32768 through +32767.

You may wonder if a signed multiply or divide is possible. Yes, but the rules for a signed operation are somewhat sticky. It is best to find the **absolute value** of the operands, perform the unsigned multiply and divide, and then convert the result to the proper sign.

Overflow Limits

Another problem that we haven't discussed in the above code is **overflow**. Overflow may occur when the register(s) dedicated to the product or quotient is not large enough to hold the largest possible case.

For an unsigned multiply, overflow can occur if the results registers do not hold at least a number of bytes equal to the total number of bytes in the multiplicand and multiplier. For example, a multiply of a one-byte number by a two-byte number may produce a three-byte result (but not greater than three bytes).

For an unsigned divide, overflow may occur if the register(s) dedicated to the quotient is not equal to or greater than the number of bytes in the dividend. The worst case here is the divide-by-one case. Divide-by-zero is not a legitimate operation and will result in a quotient of all ones.

How do you detect overflow? Not by the P/V flag, which is usually unrelated to the multiply or divide actions! You must know the ranges of the numbers you'll be dealing with beforehand or be certain by testing the operands that no multiply or divide operation will cause overflow.

Random Number Generation

Many times it's necessary to generate "random" numbers. The uses for random numbers ranges from simple applications such as determining whether a plague will strike in "Hammurabi," to more complex simulation and modeling.

There are really two types of "random" numbers — pseudo-random numbers and true random numbers. True random numbers aren't predictable, but pseudo-random numbers are.

True Random Numbers

An example of a true random number generation occurs when a (perfect) die is rolled or when a (perfect) coin is flipped. The next toss of the die or flip of the coin is completely unpredictable. Over the long run the

distribution of the points coming up on the die will approach an even number of ones, twos, threes, fours, fives, and sixes, and there will be close to an equal number of heads and tails for the coin flip.

Can we get a true random number on the TRS-80? The answer is a qualified yes. One way to get a 7-bit random number is to read the contents of the R register in the Z-80. The R register refreshes the dynamic memories by counting from 0 to 127 and then recycling. If an external event occurs at irregular intervals that are very large compared to the R register counting, a true random number is obtained.

One such external event is a keypress. If we look at R at every keypress, we can obtain a good random number from 0 through 127. The code for such an operation looks for a keypress and then reads R into the A register.

```
READ CALL INPUT   ;READ KEYBOARD
     JR   Z,READ  ;GO IF NO KEYPRESS
     LD   A,R     ;NOW HAVE RANDOM
                  # 0-127 IN A
```

Pseudo-Random Numbers

A pseudo-random number **is** predictable. As a matter of fact, it's convenient to have code that will start from a given "seed" value and generate a whole string of numbers without repeating each time we start from the same seed. If we have such a routine, we can still get a good distribution of all numbers over the range and repeat the sequence any time we wish.

Hints and Kinks 6–5
Random Number Problems

One of the main rules in random number generation is this: <u>Never</u> use any old algorithm to generate what you think will be a string of random numbers. If you take the contents of HL, multiply by your Social Security number, and add your best 10 kilometer running time to produce a series of pseudo-random numbers, you're almost guaranteed of producing a series that ''decays'' down to 18766 after 32 numbers have been produced. It's best to consult a good programming text for some tried and true algorithms. Even there, you'll find vehement disagreement among computer scientists about which methods are the best. (Personally, I take my <u>wife's</u> Social Security number . . . .)

One of the common algorithms for generating random numbers is to multiply an odd power of five times the seed value. The multiply produces a 16-bit or greater product (whatever we care to make it). When overflow results, the product represents the remainder of a 64K (for 16 bits), with the divide due to the length of the register. This divide is called a **modulus** operation. The product is the next pseudo-random number, and the process is repeated indefinitely. The greater the length of the register holding the product, the longer the **cycle** before the pseudo-random number sequence repeats. For 32 bits, the cycle is millions of numbers.

Two pseudo-random sequences are used in the codes of Chapters 13 and 14. They're basically the same. One is shown in Figure 6-12.

```
                                 TITLE   RANDSR
                                 ENTRY   RAND
                                 EXT     SEED
                        ;*****************************************************
                        ;*               RANDOM NUMBER ROUTINE               *
                        ;*   GENERATES A PSEUDO-RANDOM NUMBER FROM 0 TO 65535 *
                        ;*   ENTRY: NO PARAMETERS                            *
                        ;*   EXIT:  (BC)=RANDOM # 0-65535                    *
                        ;*   ALL REGISTERS SAVED EXCEPT BC                   *
                        ;*****************************************************
                        ;
0000'   F5              RAND:    PUSH    AF              ;SAVE REGISTERS
0001'   D5                       PUSH    DE
0002'   E5                       PUSH    HL
0003'   ED 5B 0000*              LD      DE,(SEED)       ;GET SEED
0007'   2A 0002*                 LD      HL,(SEED+2)
000A'   06 07                    LD      B,7             ;COUNT FOR MULTIPLY BY 128
000C'   CD 0025'        RDM010:  CALL    SHIFT             ;SHIFT ONE BIT LEFT
000F'   10 FB                    DJNZ    RDM010            ;SEED*128
0011'   06 03                    LD      B,3             ;FOR SUBTRACT
0013'   CD 002B'        RDM020:  CALL    SUB               ;SUBTRACT ONE
0016'   10 FB                    DJNZ    RDM020            ;SEED*128-3*SEED=SEED*125
0018'   ED 53 0000*              LD      (SEED),DE       ;STORE NEW SEED
                        ;*****WARNING*****  POSSIBLE LOADER ERROR IN SOME VERSIONS
                        ;OF ASSEMBLER. ASSEMBLER LOAD ADDRESS SHOULD BE SEED+2 IS
                        ;SEED!
001C'   22 0002*                 LD      (SEED+2),HL
001F'   43                       LD      B,E             ;NOW IN B
0020'   4C                       LD      C,H             ;NOW IN BC
0021'   E1                       POP     HL              ;RESTORE REGISTERS
0022'   D1                       POP     DE
0023'   F1                       POP     AF
0024'   C9                       RET                     ;RETURN
                        ; SHIFT SUBROUTINE
0025'   29              SHIFT:   ADD     HL,HL           ;SHIFT HL
0026'   EB                       EX      DE,HL           ;GET MS BYTE
0027'   ED 6A                    ADC     HL,HL           ;SHIFT MS 2 BYTES
0029'   EB                       EX      DE,HL           ;NOW ORIGINAL*2
002A'   C9                       RET                     ;RETURN
                        ; SUBTRACT SEED SUBROUTINE
002B'   C5              SUB:     PUSH    BC              ;SAVE REGISTERS
002C'   ED 4B 0002*              LD      BC,(SEED+2)     ;GET LS BYTE
0030'   B7                       OR      A               ;RESET CARRY
0031'   ED 42                    SBC     HL,BC           ;SUBTRACT LS 2 BYTES
0033'   EB                       EX      DE,HL           ;GET MS 2 BYTES
0034'   ED 4B 0000*              LD      BC,(SEED)       ;GET MS 2 BYTES
0038'   ED 42                    SBC     HL,BC           ;SUBTRACT MS 2 BYTES AND CY
003A'   EB                       EX      DE,HL           ;NOW ORIGINAL-SEED
003B'   C1                       POP     BC              ;RESTORE REGISTERS
003C'   C9                       RET                     ;RETURN
                                 END
```

Figure 6-12.
Pseudo-Random Number Routine

The RAND routine performs a shift and **subtract** of the four-byte SEED value. Seven shifts of SEED in DE, HL are done by calling the SHIFT subroutine. This multiplies the old seed by 128. Then the old SEED is subtracted three times from the shift result to give SEED*125. The new seed is stored into SEED for the generation of the next pseudo-random number.

Towards Infinite Precision

A frequent question in assembly language is how to handle large numbers. **Multiple-precision** adds and subtracts are relatively easy to implement (by using ADC and SBC instructions), but multiplies and divides of many bytes are tedious to code.

When the range of numbers is very great, a better approach is to implement **floating-point** processing in the assembly language. Floating-point holds numbers in a scientific-notation format geared to microcomputers (see Figure 6-13). Unfortunately, the implementation of this code for such processing is fairly complex.

SIGN OF FRACTION

BYTE 0		BYTE 1
+ −	EXPONENT ± POWER OF 16 (EXCESS 64 CODE)	FRACTION MS BYTE
FRACTION NEXT SIG. BYTE		FRACTION LS BYTE
BYTE 2		BYTE 3

Figure 6-13. Floating-Point Number Representation (Typical)

Disk Editor/Assembler users have the **arithmetic routines** of the Disk Editor/Assembler **library** available to them to perform addition, subtraction, multiplication, division, and exponentiation of both **integer** and **real (floating-point)** numbers. The Disk Editor/Assembler manual describes the ways to use these options, but their use basically involves setting up arguments for the function registers and then referencing the library routine by a CALL. The library routine is then automatically loaded at **load time** by the Disk Editor/Assembler Loader.

Chapter Seven
Working With Character Data

Assembly-language routines to generate and process ASCII character data are our topics for this chapter. The first of these routines is the procedure for reading ASCII characters from the TRS-80 keyboard. Although ROM routines can be used, constructing your own keyboard **scanning** and conversion routines are not difficult and offer a lot of versatility.

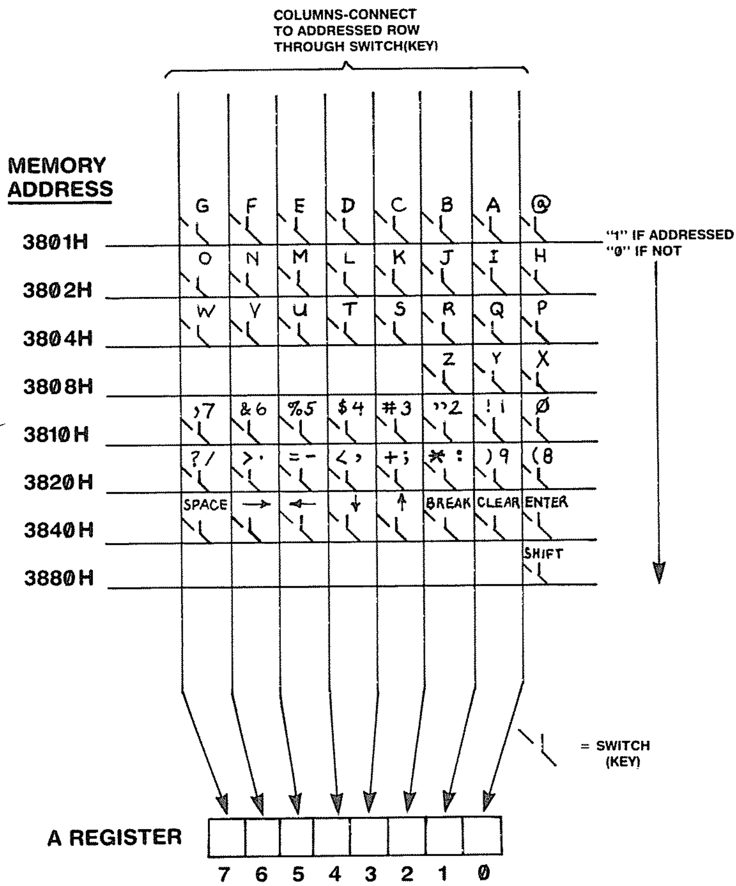
Other common character-processing routines found in assembly-language programs are string input routines to read in user character strings, message output routines to display messages on the screen, character output routines to echo input characters, and routines to convert between binary data and ASCII **decimal** strings.

We'll get to all of these routines in this chapter, paying close attention to the **keyboard scanning** process and conversion to ASCII.

Keyboard Operation and Scanning

There are built-in routines in Level II BASIC and Disk BASIC to read the keyboard and convert a keypress into a character. The addresses and calls are documented in other Radio Shack literature. However, we can bypass these and read the keyboard directly. In doing so, we can make the routine as versatile as we want by making the keys represent any character or function!

The keyboard is really a **matrix** of switches as shown in Figure 7-1. There's nothing magical about its operation. Pressing a key simply connects a column line and a row line together for as long as the key is held down. There's a certain amount of **keybounce** associated with the connection. If we could observe the connection from a microscopic scale in slow motion, we'd see a definite "make-break-make . . . " before the connection was firmly established. The same thing would happen when the key was released. This process is shown in Figure 7-2.



14352

38FF is all

Note: Model III configuration identical except that LEFT SHIFT on column Ø, RIGHT SHIFT on column 1.

Figure 7-1. TRS-80 Model I Keyboard Configuration (Upper Case)

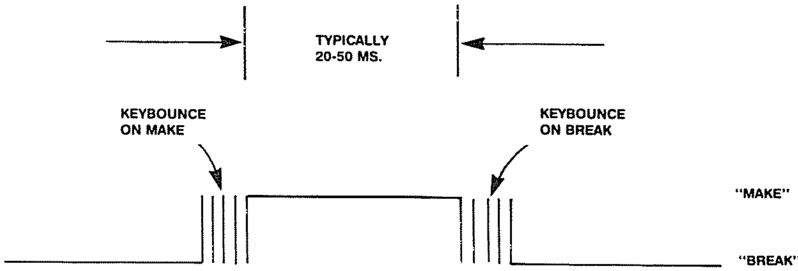


Figure 7-2. Keyboard Bounce

Our job, and it's fairly simple, is to convert a switch connection defined by a row, column into an ASCII character and to **debounce** the contact to avoid reading the same key many times.

Scanning

Each of the eight rows is addressed by a unique address, as shown in Figure 7-1. The address of the first row is 3801H, the second 3802H, the third 3804H, the fourth 3808H, the fifth 3810H, the sixth 3820H, the seventh 3840H, and the last 3880H. How do we address the rows? By simply doing a "load" from the memory location representing the row. To address the third row, we'd do

```
READ LD  A,(3804H) ;READ KEYBOARD ROW 2
```

Here, we've called the row "row 2", as we counted from "row 0".

When the LD is executed, the A register is loaded with a byte that represents the state of the column lines at that instant in time. A column line will hold a one bit **if a key along the addressed row is being held down**. If more than one key along that row is being held down, there will be several one bits. Note that the keys associated with other rows are not detected at all.

The process of addressing eight rows, one at a time, and looking for a one bit representing a keypress is called **keyboard scanning**. Since an LD instruction takes about 8 microseconds (8 millionths of a second), you can do this keyboard scan very quickly relative to a keypress. You might hold down the typical key for 50 milliseconds or so, representing the time of 250 complete scans!

The processing for detecting any keypress, then, is really very simple and goes something like this:

1. Read row 0, address 3801H. Look for a non-zero value. If zero, go on else go to step 9.
2. Read row 1, address 3802H. Look for a non-zero value. If zero, go on else go to step 9.
3. Read row 2, address 3804H. Look for a non-zero value. If zero, go on else go to step 9.

•

•

•

7. Read row 6, address 3840H. Look for a non-zero value. If zero, go on else go to step 9.
8. Scan complete and no key press. Go back to step 1.
9. Keypress here. Row number is known. Find column number by finding which bit of the eight columns is a one. Read the state of row 7, the SHIFT key and record it if necessary. Convert the key to ASCII or another code and take action on it.

Conversion

When the keyboard scan detects a non-zero value in any row, the next step to perform is conversion of the row, column representation into ASCII or some similar code. For example, the intersection of row 2, column (bit) 3 is associated with the key marked S on the keyboard.

The usual conversion of this key will, of course, be to an ASCII S, or 53H. There's no reason, however, that you can't convert this key to a code that means "scroll up," "insert," "get the next disk sector," or anything else you want. The point is that the conversion routine simply converts one of 53 keys to any value the programmer wants. Usually these values are ASCII.

The **algorithm** for the conversion is this:

1. Take the least significant byte of the row value — 1,2,4,8,16,32, or 64. Convert to the row number of 0,1,2,3,4,5, or 6.
2. Multiply the row number by 8. We now have 0,8,16, 24,32,40, or 48.
3. Add in the column bit number of 0 through 7. We now have a unique value from 0 through 55, representing the key that has been pressed.
4. If the SHIFT key is to be used, add 0 if no SHIFT or 56 if shift. We now have a unique value of 0 through 111 representing the key that has been pressed and the "upper-" or "lower-case" status.
5. Use the value obtained above to get one of 56 or 112 bytes from a keyboard conversion table. The byte obtained will be the ASCII value of the key, or whatever code we want to use for the key.

```
Hints and Kinks 7-1
Keyboard Algorithm

The index to the keyboard conversion table is
given by:

Index = (Log (base 2) (Row Address-3800H))*8
+Column Number + 56*(Shift)

Here Shift=0 for no shift and 1 for shift

Suppose that the $ key was pressed in Figure
7-1. We'd have:

Index = (Log (base 2) (3810H-3800H))*8 +
Column Number + 56*(Shift)

Index = (Log (base 2) (16))*8 + 4 + 56*1 =
4*8 + 4 +56 = 92
```

Debouncing

Debouncing is necessary because of the high speed of keyboard scanning. If we scanned the keyboard, converted a key, stored the value, and then came back to repeat the process for the next key, we'd probably be able to start the next scan after 200 microseconds. Since the typical key is held down for 50 milliseconds, we'd pick up the key 250 times again!

We need to **delay** a certain amount of time after we first detect and process the key. This delay will be long enough so that the key is released in the interim. If we are speaking of average typing speeds of 40 words per minute, the number of characters per second are about 4 or less. This is one every 250 milliseconds or so. Delaying 100 milliseconds then, should more than handle average typing speeds and yet bypass the period during which the key is held down.

> Hints and Kinks 7-2
> ''Auto'' Key
>
> One of the easier things to implement in an assembly-language keyboard routine is an ''automatic'' key function. If you hold the key longer than the 100 millisecond delay, it is reread. This results in an ''auto'' key function for any key, which will operate at 10 characters/second.

There are more sophisticated keyboard decoding routines that handle "*n*-key rollover" in which you can press a new key while you are still holding the old, but the delay technique above is fine for most processing.

A Typical Keyboard Subroutine

We now have all the elements we need to read and convert the keyboard. Figure 7-3 shows a complete subroutine that will perform the task.

```
F000                        ORG     0F000H
                     ;**************************************************************
                     ;*                 READ KEYBOARD ROUTINE                      *
                     ;*  ENTRY: NO PARAMETERS                                      *
                     ;*  EXIT: (A)=KEY CONVERTED TO KBLUT VALUE                    *
                     ;*             OR ZERO IF NO KEY HAS BEEN PRESSED             *
                     ;**************************************************************
                     ;
                     ; SCAN SEVEN ROWS
F000 210138          READKB  LD      HL,3801H        ;ROW 0 ADDRESS
F003 7E              REA010  LD      A,(HL)            ;GET ROW VALUE
F004 B7                      OR      A                 ;TEST FOR ZERO
F005 2005                    JR      NZ,REA020         ;GO IF KEY THERE
F007 CB25                    SLA     L                 ;SHIFT ROW ADDRESS
F009 F8                      RET     M                 ;GO IF LAST ROW, NO PRESS
F00A 18F7                    JR      REA010            ;MORE ROWS TO GO
                     ; CONVERT ROW, COLUMN TO INDEX
F00C 4F              REA020  LD      C,A             ;ROW VALUE
F00D AF                      XOR     A               ;ZERO A
F00E CB3D            REA025  SRL     L                 ;SHIFT ADDRESS
F010 3804                    JR      C,REA035          ;GO IF DONE
F012 C608            REA030  ADD     A,8               ;ROW*8
F014 18F8                    JR      REA025            ;CONTINUE
F016 06FF            REA035  LD      B,0FFH          ;COLUMN COUNT
F018 04              REA040  INC     B                 ;BUMP COUNT
F019 CB39                    SRL     C                 ;SHIFT ROW VALUE
F01B 30FB                    JR      NC,REA040         ;CONTINUE TILL "1" OUT
F01D 80                      ADD     A,B             ;NOW ROW*8+COL IN A
F01E 4F                      LD      C,A             ;TRANSFER TO C
                     ; FIND TABLE ENTRY
F01F 3A8038                  LD      A,(3880H)       ;SHIFT ROW ADDRESS
F022 B7                      OR      A               ;TEST FOR SHIFT
F023 2802                    JR      Z,REA045        ;GO IF NONE
F025 3E38                    LD      A,56
F027 81              REA045  ADD     A,C             ;ADD HASH+SHIFT*56
F028 4F                      LD      C,A             ;NOW ROW*8+COL*SHIFT*56
F029 0600                    LD      B,0             ;NOW IN BC
F02B 213AF0                  LD      HL,KBLUT        ;ADDRESS OF LOOK UP TABLE
F02E 09                      ADD     HL,BC           ;POINT TO CODE
F02F 7E                      LD      A,(HL)          ;GET VALUE
                     ; DEBOUNCE DELAY
F030 210021                  LD      HL,8448         ;100 MS DELAY
F033 01FFFF                  LD      BC,-1           ;DECREMENT
F036 09              REA050  ADD     HL,BC             ;DECREMENT COUNT
F037 38FD                    JR      C,REA050          ;GO IF NOT DONE
F039 C9                      RET                     ;RETURN WITH CODE IN A
                     ; LOOK UP TABLE
0008                 KBLUT   DEFS    8               ;ROW 0 LC
0008                         DEFS    8               ;ROW 1 LC
0008                         DEFS    8               ;ROW 2 LC
0008                         DEFS    8               ;ROW 3 LC
0008                         DEFS    8               ;ROW 4 LC
0008                         DEFS    8               ;ROW 5 LC
0008                         DEFS    8               ;ROW 6 LC
0008                         DEFS    8               ;ROW 0 UC
0008                         DEFS    8               ;ROW 1 UC
0008                         DEFS    8               ;ROW 2 UC
0008                         DEFS    8               ;ROW 3 UC
0008                         DEFS    8               ;ROW 4 UC
0008                         DEFS    8               ;ROW 5 UC
0008                         DEFS    8               ;ROW 6 UC
0000                         END
00000 TOTAL ERRORS
```

Figure 7-3. Keyboard Read Routine

The subroutine is divided into scanning, converting to an index value, finding the **lookup table** value, and debounce delay.

The scanning cycles through the addresses of 3801H through 3840H. HL is initialized with 3801H. Thereafter, only L is shifted left to get the "02H", "04H", etc. For each address, an LD A,(HL) is done to read the row. If the result is zero, the scan continues. If all addresses through 3840H are used and no key is found, the subroutine exits with zero in A.

If a non-zero value is found, the row address is converted to 8 times the row number by successive adds of 8. The column number is then added to this by simply adding the value obtained from the read. We now have 0 through 55.

The shift key is read by a LD A,(3880H). If there's a shift, a value of 56 is added to the value from above. If there's no shift, 0 is added.

The index value is then used to pick up one byte from the KBLUT, the keyboard look up table. The last action is to delay 100 milliseconds by a timing loop.

The KBLUT should be assembled with standard Radio Shack ASCII and **control codes**. Any other codes could be substituted by the user. The orientation of the codes is opposite that of Figure 7-1. The first 8 bytes, for example, would be the codes for @, A, B, C, D, E, F, and G.

Other Keyboard Subroutines

We used some variations on the above routine in the programs of Chapters 13 and 14. Chapter 13 uses a special keyboard routine that is geared to a quick scan and return if no key has been pushed. The quick scan is implemented by an LD A,(387FH) shown in Figure 7-4. This reads in **all** of the rows at one time and **merges** all column bits. If there is a zero in A after this instruction, no key is being pressed. If the result is non-zero, the usual row-by-row scan is used.

```
             ;****************KEYBOARD INPUT SUBROUTINE****************
             ;*   IF DEBOUNCE DELAY LESS THAN ELAPSED TIME, SCANS     *
             ;*   KEYBOARD AND STORES POSSIBLE INPUT CHARACTER IN     *
             ;*   CIRCULAR INPUT BUFFER.                              *
             ;*   ENTRY: NO PARAMETERS                                *
             ;*   EXIT: NO PARAMETERS                                 *
             ;*   ALL REGISTERS SAVED                                 *
             ;*   CLEAR CHARACTER CAUSES RESTART AT MORO15H, SP RESET*
             ;
8429 F5      INPUT   PUSH    AF              ;SAVE REGISTERS
842A 3A7F38          LD      A,(387FH)       ;ALL IN ONE SWELL FOOP
842D B7              OR      A               ;TEST FOR ANY KEY
842E CAB984          JP      Z,INP065        ;GO IF NONE
8431 C5              PUSH    BC
8432 E5              PUSH    HL
8433 2AED85          LD      HL,(TSLC)       ;GET TIME SINCE LAST CHARACTER
8436 016400          LD      BC,DBDEL        ;MINIMUM DELAY
```

READS ALL KEYBOARD ROWS, MERGES ALL COLUMN BITS

Figure 7-4. Quick Scan

The INPUT subroutine of Chapter 14 (Figure 7-5) is a specialized keyboard input. It looks only for a 0 through 8 key by reading row 4 and row 5. Row 4 addresses keys 0 through 7, while row 5 addresses only key 8 in bit 0. If a key is found, a 100 millisecond delay is done by calling the DELAY subroutine.

```
                                   TITLE   INPUSR
                                   ENTRY   INPUT
                                   EXT     DELAY
                         ;*************************************************
                         ;*             INPUT SUBROUTINE                  *
                         ;*  SCANS KEYBOARD FOR INPUT OF 0,1,2,3,4,5,6,7, OR  *
                         ;*  8 KEYS. IGNORES ALL OTHERS. WAITS FOR KEY PRESS. *
                         ;*  ENTRY: NO PARAMETERS                         *
                         ;*  EXIT: (A)=0,1,2,3,4,5,6,7, OR 8 IN BINARY    *
                         ;*  ALL REGISTERS EXCEPT A SAVED                 *
                         ;*************************************************
                         ;
0000'  C5                INPUT:  PUSH    BC              ;SAVE REGISTERS
0001'  E5                        PUSH    HL
0002'  3A 3810           INP002: LD      A,(3810H)       ;KEYBOARD ROW 4
0005'  B7                        OR      A               ;TEST FOR NON-ZERO
0006'  28 09                     JR      Z,INP010        ;GO IF NON
0008'  0E FF                     LD      C,0FFH          ;COLUMN COUNT
000A'  0C                INP005: INC     C                 ;BUMP COUNT
000B'  0F                        RRCA                      ;SHIFT OUT
000C'  30 FC                     JR      NC,INP005         ;GO IF NOT CARRY
000E'  79                        LD      A,C             ;COUNT TO A
000F'  18 09                     JR      INP020          ;CONTINUE
0011'  3A 3820           INP010: LD      A,(3820H)       ;KEYBOARD ROW 5
0014'  E6 01                     AND     1               ;GET "8" KEY ONLY
0016'  28 EA                     JR      Z,INP002        ;GO IF NONE
0018'  3E 08                     LD      A,8             ;FOR 8 KEY
001A'  21 0064           INP020: LD      HL,100          ;100 MILLISEC DELAY
001D'  CD 0000*                  CALL    DELAY           ;DELAY
0020'  E1                        POP     HL              ;RESTORE REGISTERS
0021'  C1                        POP     BC
0022'  C9                        RET
                                 END
```

READ 0-7 LOGIC

READ 8 LOGIC

Figure 7-5. Special Keyboard Read

Input Subroutines

Once you have implemented a keyboard input routine, it's fairly easy to write code to read in a string of characters. This is usually a subroutine that reads in the string but doesn't do **syntax checking** or make any judgments about the validity of the data that has been read in; that task is left to other routines that will process the input character string.

The "input string" subroutine must, however, make some checks. First, there must be some terminating input character that signifies the end of the input process. If there were not, the input-string subroutine would call the keyboard-input subroutine again for the next character, when the user was actually through with input. The terminating character in the TRS-80 is almost always ENTER.

Second, there may be a maximum number of characters to be input. You may need to specify the number because the **input buffer** is of limited size, or because all input should be less than a certain number of characters.

Other options for the input string routine might include a means to **backspace** (rubout) an incorrect character or characters from the string and a returned count of the number of characters input.

Hints and Kinks 7-3

Backspacing

Backspacing is not implemented in the input routines of Chapters 14 and 15. The reason is that most input in routines of those chapters involves either one or two characters which are easily redone if an error is made.

Also, implementation of backspacing is best done when the input involves an entire line of data, as it does in BASIC. When a line is input, an input buffer is filled with the character data before any processing is done. Processing is done after the terminating character (ENTER in BASIC) has been entered. When the ''input line'' approach is used, it is a simple matter to detect a backspace (left arrow or rubout), adjust the buffer pointer to point back at the last character, and then overwrite the last character.

Have I given enough excuses?

Figure 7-6 shows a somewhat atypical input string subroutine from the MORG program of Chapter 13. It calls a keyboard input routine called INPUTW that returns the next keyboard character. INPUTW specifically waits until a key has been pressed and always returns a character. The INPUTS subroutine terminates on an ENTER character, and each new character is compared to ENTER, which is EQU(ated) earlier in the program. (Note here that the code for ENTER in this keyboard program is 02H.) If the character input is an ENTER, the subroutine is exited. Otherwise a character count is incremented in the C register, and the character is displayed on the screen by the DISCHR subroutine.

```
             ;**************INPUT STRING SUBROUTINE*****************
             ;*   INPUTS STRING OF CHARACTERS AT CURRENT COMMUNICA-   *
             ;*   TION AREA. TERMINATED BY ENTER.                     *
             ;*   ENTRY: (B)=MAXIMUM NUMBER                           *
             ;*   (CURCUR)=CURRENT CURSOR POSITION                    *
             ;*   EXIT:  (B)=ACTUAL NUMBER INPUT                      *
             ;*          (HL)=FIRST CHARACTER LOCATION                *
             ;*          NZ IF GT MAXIMUM NUMBER                      *
             ;*          Z IF LE MAXIMUM NUMBER                       *
             ;*   ALL REGISTERS SAVED EXCEPT HL,BC,A                  *
             ;
840D 2AE985  INPUTS  LD      HL,(CURCUR)     ;CURRENT CURSOR POSITION
8410 E5              PUSH    HL              ;SAVE
8411 04              INC     B               ;BUMP MAXIMUM
8412 0E00            LD      C,0             ;INITIALIZE COUNT OF CHARS
8414 C5      INS010  PUSH    BC              ;SAVE COUNTS
8415 CDF384          CALL    INPUTW            ;GET CHARACTER
8418 C1              POP     BC                ;RESTORE COUNTS
8419 FE02            CP      ENTER             ;TEST FOR DONE
841B 2809            JR      Z,INS030          ;GO IF ENTER
841D 0C              INC     C                 ;BUMP CHARACTER COUNT
841E CDBE83          CALL    DISCHR            ;DISPLAY
8421 10F1            DJNZ    INS010            ;GO IF NOT MAXIMUM
8423 3EFF            LD      A,0FFH          ;-1 TO A
8425 B7              OR      A               ;RESET Z FLAG
8426 E1      INS030  POP     HL              ;RETRIEVE START
8427 41              LD      B,C             ;GET CHARACTER COUNT
8428 C9              RET                     ;RETURN
             ;
```

Figure 7-6. Input String Routine

Hints and Kinks 7-4
Scanning Versus Waiting

The usual keyboard input routine is a ''read-one-character-and-wait-until-input'' type. It will never return until that character has been input. The rationale is that this is the most frequent type of input; the program asks for user input before processing.

The scanning type of input routine is used less frequently. Here, the program is processing merrily along but is keeping an eye out for a user stimulus. This stimulus is usually a user ''abort'' action, although it could be normal keyboard input, as in the case of the MORG program of Chapter 13. The goal in this type of input routine is to make certain that the polling of the keyboard input is done periodically at regular measured intervals (so that no characters are missed) and yet done fast enough that other processing can continue without being overburdened by the keyboard poll.

In many other systems, input of infrequent (compared to CPU processing) keyboard characters would be accomplished by an interrupt from the keyboard when the next character was ready. That way, normal processing would continue until the keyboard interrupt was received, eliminating the polling task. One way to time the polling on a disk system is by using the system's real-time-clock. It provides 25 millisecond increments, which lend themselves nicely to timing any type of polling function.

The B register initially held the maximum number of characters to be input plus one. After each character is input, B is decremented by a DJNZ. If less than the maximum has been input, the code goes to the next character. If the maximum number of characters has been input, the subroutine is terminated with an error flag of "NZ".

The characters are stored in the video display memory. This causes no problem as long as the characters are valid ASCII characters. However, non-standard characters may be changed in the upper-case versions of the TRS-80. The DISCHR routine is called to display the input characters because input in assembly language does not automatically display the characters as a BASIC input does!

Display of Characters

This brings us to a discussion of the next topic — what has to be done to display character data in assembly language? BASIC has built-in print driver routines that handle all output of characters. Character strings are "formatted" and tabbed, new lines are output, the screen "scrolls" automatically, etc.

Unless you make calls to the BASIC routines to perform these functions, you must implement all **display processing** via assembly-language routines. Fortunately, display processing is not that involved, and we'll look at some of the techniques here.

Displaying A Message

To display a message on the screen, a string of ASCII characters from a DEFM is output one at a time to the screen memory area. It's convenient to have a terminating character to end the message. The terminator we've used in the programs of Chapters 13 and 14 is a zero (null) at the end of each message. We used the zero because it is not a valid ASCII character and we can easily test it.

An alternative to using zero is to use the disk Editor/Assembler "DC" pseudo-op, which sets the high-order bit of the **last** character to one. This can also be easily detected (and masked out) for the "end of message."

Hints and Kinks 7–5
DC Pseudo–Op

Perhaps the DC pseudo–op should have been used in the code of Chapters 13 and 14 since it is specifically for generating character strings for output. The output routine would then test the sign bit of every character and terminate when a ''minus'' was detected. The scheme used in the code of the applications programs wastes one byte for each message. Thank goodness memory is inexpensive!

The screen area for the message is usually pointed to by the HL register or another register pair. It is incremented by one for every new ASCII byte output. Figure 7-7 shows the DSPMES subroutine from Chapter 13. This subroutine uses HL to point to the ASCII message and BC to address the "starting screen address." The starting screen address would be 3C00H through 3FFFH.

```
               ;**************DISPLAY MESSAGE AT LOCATION N*************
               ;*   DISPLAYS MESSAGE AT GIVEN SCREEN POSITION. TER-    *
               ;*   MINATES ON NULL (ZERO).                            *
               ;*   ENTRY: (HL)=MESSAGE LOCATION                       *
               ;*          (BC)=SCREEN POSITION                        *
               ;*   ALL REGISTERS SAVED.                               *
               ;
836D F5        DSPMES  PUSH    AF              ;SAVE REGISTERS
836E C5                PUSH    BC
836F E5                PUSH    HL
8370 7E        DSP005  LD      A,(HL)           ;GET MESSAGE CHAR
8371 B7                OR      A                ;TEST FOR 0
8372 2809              JR      Z,DSP010         ;RETURN IF DONE
8374 02                LD      (BC),A           ;STORE CHARACTER
8375 03                INC     BC               ;BUMP SCREEN POINTER
8376 23                INC     HL               ;BUMP MESSAGE POINTER
8377 ED43E985          LD      (CURCUR),BC      ;SAVE POINTER
837B 18F3              JR      DSP005           ;CONTINUE
837D E1        DSP010  POP     HL              ;RESTORE REGISTERS
837E C1                POP     BC
837F F1                POP     AF
8380 C9                RET                     ;RETURN
```

Figure 7-7. Display Message Routine

DSPMES also stores the "current cursor position" in variable CURCUR. This variable can be tested for "end of line" or "end of screen" (scrolling) conditions.

Displaying an Input Character

When you input a character from a keyboard routine, it must be immediately "echoed" to the screen by a "display character" routine. This subroutine would typically take a "current cursor position" and use it as the address of the screen position to store the input character. Figure 7-8 shows such a routine that uses variable CURCUR as the current cursor position.

```
             ;**************DISPLAY CHARACTER SUBROUTINE**************
             ;*   OUTPUTS ONE CHARACTER TO CURRENT CURSOR POSITION   *
             ;*   ON SCREEN. MOVES CURSOR TO NEXT POSITION UNLESS    *
             ;*   LAST CHARACTER POSITION OF LINE 11. IF LATTER,     *
             ;*   SCROLLS UP FIRST.                                  *
             ;*   ENTRY: (CURCUR)=CURRENT CURSOR POSITION            *
             ;*          (A)=CHARACTER TO BE OUTPUT                  *
             ;*   ALL REGISTERS SAVED.                               *
             ;
83BE C5      DISCHR  PUSH  BC            ;SAVE REGISTERS
83BF E5              PUSH  HL
83C0 2AE985          LD    HL,(CURCUR)   ;GET CHARACTER POSITION
83C3 77              LD    (HL),A        ;STORE CHARACTER
83C4 01FF3E          LD    BC,3C00H+767  ;LAST CP OF LINE 11
83C7 23              INC   HL            ;BUMP CURSOR
83C8 22E985          LD    (CURCUR),HL   ;STORE
83CB B7              OR    A             ;RESET CARRY
83CC ED42            SBC   HL,BC         ;TEST FOR LAST
83CE 2003            JR    NZ,DIS010     ;RETURN IF NO SCROLL
83D0 CDD683          CALL  SCROLL        ;SCROLL UP
83D3 E1      DIS010  POP   HL            ;RESTORE REGISTERS
83D4 C1              POP   BC
83D5 C9              RET                 ;RETURN
```

Figure 7-8. Display Character Routine

Scrolling

DISCHR also tests CURCUR for a condition where **scrolling** is required. This condition normally occurs when the last character position on the screen, location 3FFFH, has been used. At that point you must scroll up the screen by moving lines 1 through 15 into lines 0 through 14, and "blank" the last line.

It's possible to scroll the first five or ten lines since we can format the screen any way we want — it's simply a matter of coding! The SCROLL routine in Figure 7-9 scrolls the first 12 lines on the screen when the CURCUR reaches the end of the 12th line. Figure 7-10 shows the action. Scrolling on the TRS-80 is easy because of the LDIR block-move instruction and the fact that the screen is **memory mapped**. FILLCH is a "Fill Character" routine to fill the last line of the screen with blanks.

```
                 ;
                 ;***************SCROLL SCREEN SUBROUTINE****************
                 ;*   SCROLLS LINES 1-11 UP TO LINES 0-10. FILLS LINE 11 *
                 ;*   WITH BLANKS.                                       *
                 ;*   ENTRY: NO PARAMETERS                               *
                 ;*   ALL REGISTERS SAVED.                               *
                 ;
83D6 F5          SCROLL  PUSH    AF              ;SAVE REGISTERS
83D7 C5                  PUSH    BC
83D8 D5                  PUSH    DE
83D9 E5                  PUSH    HL
83DA 11003C              LD      DE,SCREEN       ;START OF SCREEN
83DD 21403C              LD      HL,LINE1        ;LINE 1
83E0 010003              LD      BC,1024-256     ;# TO MOVE
83E3 EDB0                LDIR                    ;MOVE EM
83E5 11C03E              LD      DE,LINE11       ;START OF LINE 11
83E8 3E20                LD      A,' '           ;SPACE
83EA 014000              LD      BC,64           ;# TO FILL
83ED CD8385              CALL    FILLCH          ;FILL LINE
83F0 21C03E              LD      HL,LINE11       ;START OF LINE 11
83F3 22E985              LD      (CURCUR),HL     ;RESET
83F6 E1                  POP     HL              ;RESTORE REGISTERS
83F7 D1                  POP     DE
83F8 C1                  POP     BC
83F9 F1                  POP     AF
83FA C9                  RET                     ;RETURN
```

"BLOCK" MOVE TO DO THE ACTUAL SCROLLING

FILLS 12TH LINE WITH BLANKS AFTER SCROLL

Figure 7-9. Scroll Screen Routine

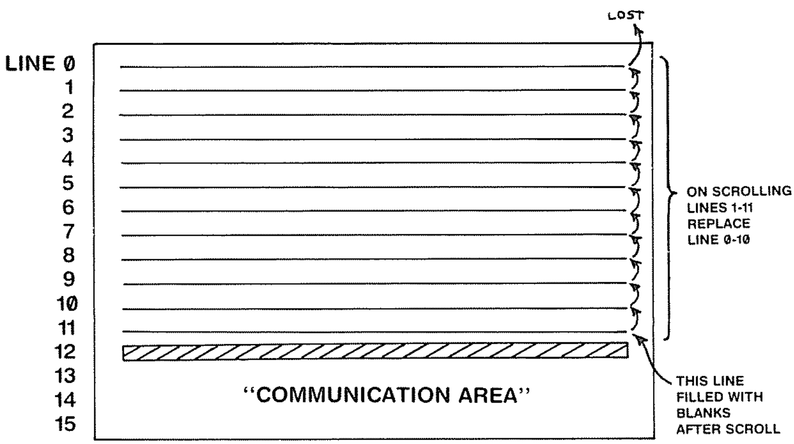
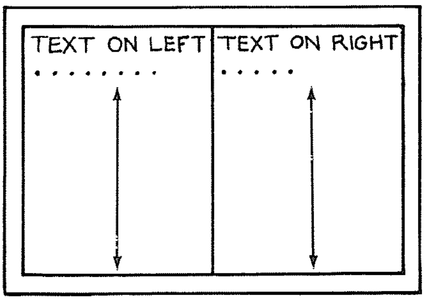


Figure 7-10. Scroll Action

Hints and Kinks 7-6
Multiple Scrolling

The TRS-80 screen can be divided up into any number of separate scroll areas by using the techniques shown in Figures 7-9 and 7-10. Implementing a ''split screen'' capability, however, is another problem. Split screens split the screen into a right and left hand portion and can be very useful for comparing old text with new text or working with two ''documents'' at a time.



One way to implement a split screen of this type would be to look for a screen address with the last 5 bits equal to 1 1111. This would denote the 31st character of either the left or right hand segments. This scheme works because lines start at ''40H boundaries,'' as they are 64 characters wide; line addresses always start at XX00H, XX40H, XX80H, or XXC0H. When this address was detected, the left or right hand pointer could be reset to the beginning of the line by adding 20H.

Scrolling is also more tedious (read ''messy''), because the areas involved are not contiguous. A scroll would have to be handled one 32-character line at a time.

All of this is possible, but it might be best to orient your applications towards multiple screens stacked on top of each other, rather than side by side - unless you're the stubborn type . . . .

Conversion From ASCII Decimal to Binary

A character string input from a keyboard or "input string" routine usually contains a mix of string data such as names and addresses and numeric data. The string data can be left as is in ASCII form for storage in arrays or records. However, you have to convert the numeric data to binary for processing. Any program that works with numeric data that is user entered invariably has a "decimal-to-binary" conversion routine.

The algorithm for converting from ASCII characters of 0 through 9 to binary goes something like this (and is shown in Figure 7-11):

1. Clear a result total.
2. Multiply the result total by 10.
3. Get the leftmost ASCII character.
4. Subtract 30H to get a binary value of 0 through 9.
5. Add the value to the result total.
6. Repeat steps 2 through 5 for the next leftmost character until done.

NUMBER IN BUFFER
(ASCII)

1	2	3	4	5

STEP	RESULT	RESULT * 10	NEXT CHAR	−30H	ADD TO RESULT
1	0	0	31H	1	1
2	1	10	32H	2	12
3	12	120	33H	3	123
4	123	1230	34H	4	1234
5	1234	12340	35H	5	12345
					(BINARY)

Figure 7-11. Decimal-to-Binary Action

You can carry out this conversion process for a string of ASCII digits as long as need be. Practical sizes, however, are limited to values that can be held in 16 or 32 bits. Since 16 bits can hold up to 65535, a decimal-to-binary routine that works in 16 bits can process most applications.

Decimal-to-binary routines also usually do testing of the ASCII characters to determine that they are valid ASCII values of 30H (0) through 39H (9). If they are found to be invalid values, the conversion is terminated with an "error" flag.

Figure 7-12 shows a DECBIN subroutine from Chapter 13 that converts a string of ASCII characters representing a decimal value to a binary value of 0 through 65535. A "shift and add" technique is used to multiply by 10. The routine is entered with HL pointing to the buffer containing the string and B containing the number of characters to be converted.

CLEAR RESULT TOTAL

```
               08970 ;********DECIMAL TO BINARY CONVERSION SUBROUTINE*********
               08980 ;*   CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING    *
               08990 ;*   DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535.   *
               09000 ;*   ENTRY: (HL)=BUFFER CONTAINING ASCII                 *
               09010 ;*          (B)=NUMBER OF CHARACTERS                     *
               09020 ;*   EXIT:  (HL)=BINARY # 0-65535                        *
               09030 ;*          NZ IF INVALID ASCII CHARACTER OTHERWISE Z    *
               09040 ;*   ALL REGISTERS SAVED EXCEPT A,HL                     *
               09050 ;
8590 C5        09060 DECBIN  PUSH   BC          ;SAVE REGISTERS
8591 D5        09070         PUSH   DE
8592 DDE5      09080         PUSH   IX
8594 DD210000  09090         LD     IX,0        ;SET RESULT
8598 DD29      09100 DEC040  ADD    IX,IX        ;INTERMEDIATE*2
859A DDE5      09110         PUSH   IX
859C DD29      09120         ADD    IX,IX        ;*4
859E DD29      09130         ADD    IX,IX        ;*8
85A0 D1        09140         POP    DE           ;*2
85A1 DD19      09150         ADD    IX,DE        ;*10
85A3 7E        09160         LD     A,(HL)      ;GET CHARACTER
85A4 D630      09170         SUB    30H         ;CONVERT
85A6 FAB685    09180         JP     M,DEC070    ;GO IF LT "0"
85A9 FE0A      09190         CP     10          ;TEST FOR GT "9"
85AB F2B685    09200         JP     P,DEC070    ;GO IF GT "9"
85AE 5F        09210         LD     E,A         ;NOW IN E
85AF 1600      09220         LD     D,0         ;NOW IN DE
85B1 DD19      09230         ADD    IX,DE       ;MERGE
85B3 23        09240         INC    HL
85B4 10E2      09250         DJNZ   DEC040       ;GO IF MORE
85B6 78        09260 DEC070  LD     A,B         ;COUNT TO A
85B7 B7        09270         OR     A           ;SET OR RESET Z FLAG
85B8 DDE5      09280         PUSH   IX          ;RESULT TO HL
85BA E1        09290         POP    HL
85BB DDE1      09300         POP    IX          ;RESTORE REGISTERS
85BD D1        09310         POP    DE
85BE C1        09320         POP    BC
85BF C9        09330         RET                ;RETURN
```

MULTIPLY RESULT TOTAL BY 10

GET BINARY VALUE

ADD TO RESULT TOTAL

Figure 7-12. Decimal-to-Binary Routine

The conversion loop is executed "B" times. After the subtract of 30H, an error return is made if the result is negative (less than 30H), or greater than 9 (greater than 39H). An NZ condition is present on return if there was an invalid character.

Converting From Binary to Decimal ASCII

This processing converts binary data back into displayable or printable form. As in the case of DECBIN above, the same size limitations apply. Usually 16 or 32 bits of binary data are converted to ASCII decimal digits.

The "normal" algorithm for this conversion is the following:

1. Divide the binary data by 10.
2. Save the quotient of the divide for the next divide.
3. Add 30H to the remainder to produce an ASCII 0 through 9.
4. Store the ASCII character at the next **rightmost** character position in a buffer.
5. Repeat steps 1 through 4 until the quotient is zero. These steps are shown in Figure 7-13.

NUMBER IN REGISTER
(BINARY)

12,345

STEP	DIVIDE BY 10	Q	R	ADD 30H TO R	STORE IN BUFFER
1	12345/10	1234	5	35H	5 (ASCII)
2	1234/10	123	4	34H	4 5
3	123/10	12	3	33H	3 4 5
4	12/10	1	2	32H	2 3 4 5
5	1/10	0	1	31H	1 2 3 4 5

Figure 7-13. Binary-to-Decimal Action

The BINDEC subroutine of Figure 7-14, however, employs an alternative approach. It uses a "divide by powers of 10" method of conversion. Starting with 10000, it performs successive subtractions on the binary number to divide by the power of 10. The quotient for each subtract is the power of 10 digit, which is then converted to ASCII. Successively smaller powers of 10 down to 1 are used. The algorithm for this is shown in Figure 7-15.

```
8000            00100           ORG     8000H
                00110 ;*************************************************************
                00120 ;*                    BINARY TO DECIMAL SR                  *
                00130 ;*   ENTRY:(HL)=16-BIT BINARY VALUE                         *
                00140 ;*          (IX)=POINTER TO START OF CHARACTER BUFFER       *
                00150 ;*   EXIT: (BUFFER)=FILLED WITH FIVE ASCII CHARACTERS,      *
                00160 ;*            LEADING ZEROES                                *
                00170 ;*          (IX)=BUFFER+5                                   *
                00180 ;*************************************************************
                00190 ;
8000 FD212580   00200 BINDEC  LD      IY,PTABLE       ;POWER OF 10 TABLE
8004 AF         00210 BIN010  XOR     A                 ;DIGIT COUNT TO 0
8005 FD5601     00220         LD      D,(IY+1)          ;GET MS BYTE
8008 FD5E00     00230         LD      E,(IY+0)          ;GET LS BYTE
800B B7         00240 BIN020  OR      A                   ;CLEAR CARRY
800C ED52       00250         SBC     HL,DE               ;SUBTRACT POWER OF 10
800E 3803       00260         JR      C,BIN030            ;GO IF NEGATIVE
8010 3C         00270         INC     A                   ;BUMP DIGIT COUNT
8011 18F8       00280         JR      BIN020              ;CONTINUE
8013 19         00290 BIN030  ADD     HL,DE             ;RESTORE TO POSITIVE
8014 C630       00300         ADD     A,30H             ;CONVERT TO ASCII
8016 DD7700     00310         LD      (IX+0),A          ;STORE IN BUFFER
8019 DD23       00320         INC     IX                ;BUMP BUF POINTER
801B FD23       00330         INC     IY                ;BUMP PWR 10 PNTR
801D FD23       00335         INC     IY
801F 7B         00340         LD      A,E               ;GET LS BYTE
8020 FE01       00350         CP      1                 ;TEST FOR 5 DIGITS
8022 20E0       00360         JR      NZ,BIN010         ;GO IF NOT 5
8024 C9         00370         RET                     ;RETURN
8025 1027       00380 PTABLE  DEFW    10000
8027 E803       00390         DEFW    1000
8029 6400       00400         DEFW    100
802B 0A00       00410         DEFW    10
802D 0100       00420         DEFW    1
0000            00430         END
00000 TOTAL ERRORS
```

Figure 7-14. Binary-to-Decimal Routine

POWER OF 10

	10000	1000	100	10	1
NUMBER: 12345 −10000					
2345 −10000	once				
−XXXXX +10000					
2345 −1000		once			
1345 −1000		twice			
345 −1000					
−XXXX +1000					
345 −100			once		
245 −100			twice		
145 −100			three		
45 −100					
−XXX +100					
45 −10				once	
35 −10				twice	
25 −10				three	
15 −10				four	
5 −10					
−X +10					
5 −1					once
4 −1					twice
3 −1					three
2 −1					four
1 −1					five
0 −1					
−X +1					
0					

Figure 7-15. "Powers of Ten" Binary to Decimal

This subroutine converts a 16-bit binary number into 5 ASCII digits and returns a pointer to the last digit plus one. There are no error conditions for conversion of the 16-bit value.

Hints and Kinks 7-7
Converting Between ASCII and Binary or Hex

Converting from an ASCII string of characters representing binary or hexadecimal digits or the other way around isn't done as frequently as decimal/ASCII conversion. However, each conversion is relatively easy, since no multiplication must be done; the process simply involves stripping off one or four bits and translating them one at a time to ASCII, or vice versa.

To convert from ASCII to binary, get the ASCII character and subtract 30H. You now have a binary one or zero. Shift to the next bit position and repeat the process for the number of characters in the string.

To convert from binary to ASCII, get the next bit and add 30H. You now have an ASCII 0 or 1. Store in the output buffer and go on to the next bit.

To convert from ASCII to hexadecimal, do this: Get the next ASCII character. This will be 30H-39H, or 41H(A)-46H(F). Subtract 30H. If the result is greater than 9, subtract 7 (this adjusts for the ASCII characters between 9 and 0). You now have a hex 0 through F. Store in the next 4 bits and repeat the process for the next ASCII character.

To convert from hexadecimal to ASCII: Get the next group of four bits. Add 30H. If the result is greater than 39H, add 7. You now have an ASCII 0 through 9, A, B, C, D, E or F. Store in the buffer, and go on to the next group of four bits.

Chapter Eight
Working With Tables

Tables are some of the most common **data structures** used in assembly-language programming. Tables are so common that large programs that use many tables to define program operations are said to be **table driven** as opposed to more "processing-oriented." We'll discuss types of tables followed by some searching and sorting operations that can be performed in assembly language.

What are Tables?

A loose definition of a table is any collection of data items arranged in one contiguous block of memory. The data items may be **ordered** by some **key** value or **unordered**. Each **entry** in a table may consist of one or more bytes. The entry may contain a number of **fields** that are associated with the entry, or there may only be one field or item. The entry itself may be **fixed length** or **variable** length, and the table may also be either fixed or variable length.

Fixed Length Entry/Fixed Length Table

Let's take a look at a typical simple table. The MOVETB from Chapter 15 is a five-entry fixed-length table; it's shown in Figure 8-1. Each entry is two bytes long, making the total table length 10 bytes. Each entry of the MOVETB is an address defining the location of a "space cell" in a tic-tac-toe game (see Chapter 15).

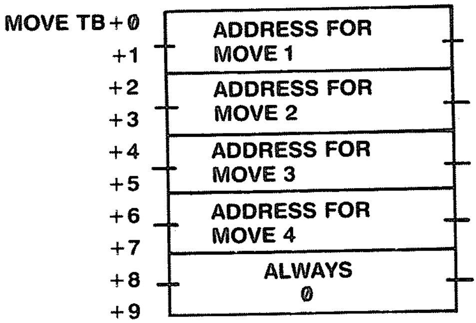


Figure 8-1. Fixed-Length Entry/ Fixed-Length Table

The entries in MOVETB are **indexed** by the move number of the tic-tac-toe game. The (computer) moves are numbered 1 through 4, so to get the address associated with any move, the formula is

```
MOVETB ENTRY ADDRESS =
MOVETB + 2*(MOVE # -1)
```

Typical code for accessing the table is shown in Figure 8-2 from the MAIN3 subroutine of Chapter 15. Here, A is loaded with the move number, MOVENO. This is decremented by one to find the previous move number and then multiplied by two. This **index value** is then loaded into the BC register pair. IX is then loaded with the address of the table start MOVETB, and BC is added to IX. IX now points to the MOVETB entry for the previous move. The address in the entry is picked up in HL by two LD instructions that use the index register.

```
00B5'    3A 0000*        ; ALL SPACE CELLS OF THIS ENTRY ARE 0. REMOVE THIS
                         ; ENTRY BY ZEROING PREVIOUS LINK AND CONCEDE.
                                 LD      A,(MOVENO)      ;GET MOVE #
00B8'    3D                      DEC     A               ;FIND LAST MOVE INDEX
00B9'    07                      RLCA                    ;INDEX*2
00BA'    4F                      LD      C,A             ;NOW IN C
00BB'    06 00                   LD      B,0             ;NOW IN BC
00BD'    DD 21 0000*             LD      IX,MOVETB       ;MOVE TABLE
00C1'    DD 09                   ADD     IX,BC           ;POINT TO LINK ADDRESS
00C3'    DD 66 00                LD      H,(IX)          ;MS BYTE
00C6'    DD 6E 01                LD      L,(IX+1)        ;LS BYTE
00C9'    AF                      XOR     A               ;ZERO
00CA'    77                      LD      (HL),A          ;ZERO LINK
```

Figure 8-2. Accessing a Simple Table

Here we didn't order the data in the table (namely, the five entries were not arranged in numerical sequence) but related them to an external index and accessed them by that index value.

Fixed Length Entry/ Variable Length Table

Another table that illustrates what we mean by "table driven" is shown in Figure 8-3. The GRIDTB of the figure is used in Chapter 15 to define a tic-tac-toe grid. There are 16 entries in the table, each one defining a line segment to be drawn by the DRAWL subroutine of that chapter. Each entry consists of five bytes and there are four fields within each entry.

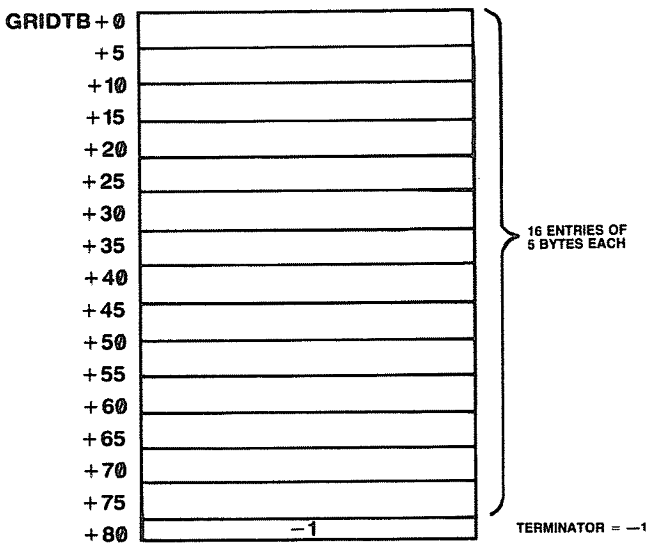


Figure 8-3. Fixed-Length Entry/ Variable-Length Table

The first, second, and third fields of the entry are each one byte long, and the fourth field is two bytes long as shown in Figure 8-4. (The first field defines the graphics character to be stored; the second is 0 for a horizontal line or 1 for a vertical line; the third represents the number of character positions in the line; and the fourth field is the starting screen address for the line.)

+0	GRAPHICS CHARACTER
+1	0=HORIZ 1=VERT
+2	#CHARACTER POS'NS
+3	STARTING
+4	SCREEN ADDRESS

Figure 8-4. Entry Fields in GRIDTB

The last byte of the table is a **terminator** of -1. The terminator marks the end of the table and may be any value that is not a legitimate value for an entry in the table. You use the table by setting a pointer to the beginning and picking up the five bytes of each entry. After you have drawn the line, you increment the pointer by 5 and the code loops back to pick up the next entry. This continues until the -1 terminator is found. The code for accessing this table is shown in Figure 8-5.

```
                      ; DRAW GRID HERE
0014'  DD 21 0000*              LD      IX,GRIDTB ;TABLE FOR GRID
0018'  DD 7E 00       ARTOO5:   LD      A,(IX)    ;GET CHARACTER
001B'  FE FF                    CP      0FFH      ;TEST FOR TERMINATOR
001D'  28 16                    JR      Z,ARTOO8  ;GO IF DONE
001F'  DD 4E 01                 LD      C,(IX+1)  ;LOAD HORIZ/VERT
0022'  DD 46 02                 LD      B,(IX+2)  ;LOAD # OF CHAR POSNS
0025'  DD 6E 03                 LD      L,(IX+3)  ;START OF LINE, LSB
0028'  DD 66 04                 LD      H,(IX+4)  ;START OF LINE, MSB
002B'  CD 0000*                 CALL    DRAWL     ;DRAW LINE
002E'  01 0005                  LD      BC,5      ;5 BYTES PER LINE
0031'  DD 09                    ADD     IX,BC     ;POINT TO NEXT LINE
0033'  18 E3                    JR      ARTOO5    ;GO FOR NEXT LINE
```

Figure 8-5. Table Use With Terminator

This somewhat "sloppy" table groups similar data together and helps "modularize" code. We could have incorporated all of the code to draw the grid into **in-line** code, but it's a lot neater and efficient to put it into a table such as GRIDTB.

Variable Length Entry/ Variable Length Table

The tables above had **fixed-length entries**. In the case of MOVETB, the **number of entries** was fixed at five. GRIDTB had a variable number of entries, the end of the table being denoted by a terminator of -1.

An example of a table consisting of **variable-length entries** with a variable number of entries is shown in Figure 8-6. The PTABLE, or Permutation Table, is used in Chapter 15 to hold configurations of tic-tac-toe games.

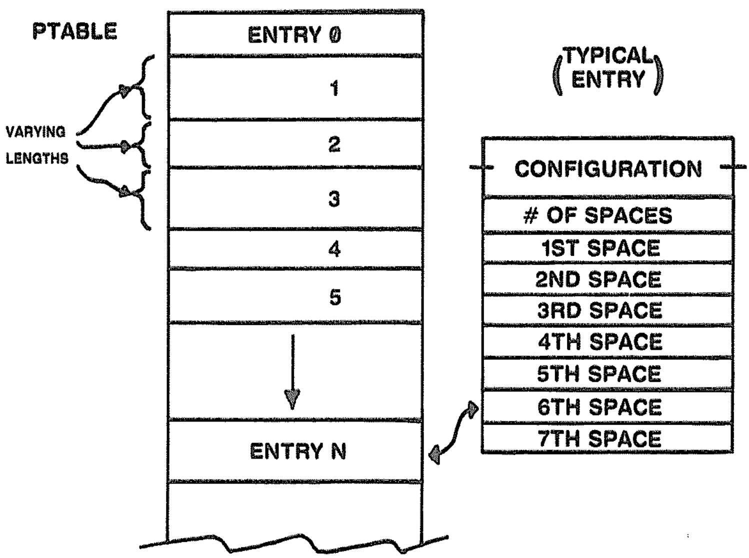


Figure 8-6. Variable-Length Entry/ Variable-Length Table

The configuration for each tic-tac-toe game is held in the first two bytes of each entry in a special binary-coded form. The next byte defines the number of "spaces" in the tic-tac-toe configuration, from 3 to 9. The next 3 to 9 bytes represent a count related to each space. The length of each entry is therefore 2+1+N where N is 3 to 9 or 6 to 12 bytes. When a search is made of the PTABLE, the length of the current entry must be computed by adding 3 plus the number found in the third byte.

PTABLE also has a variable number of entries, as the number of possible tic-tac-toe permutations is calculated **dynamically** (during program execution) by the tic-tac-toe program. In this case there is no terminator since a search of the PTABLE for a specific configuration must be successful. You don't have to have a terminator because the program will have a severe logic error if an entry is not found. The only action an unsuccessful search might produce would be a print-out message: FINISH DEBUGGING THE PROGRAM!!

Jump Tables

Another common type of table is a "jump table" or "branch table." This type of table is shown in Figure 8-7 in code from the MORG program of Chapter 14. It is used in similar fashion to a BASIC "computed GOTO" (ON N GOTO 100, 200, 300. . .). Its entries define the addresses of processing routines for the program where there are many different types of processing to be performed.

15 ONE-BYTE ENTRIES IN FTAB

```
                01130 ;
                01140 ; FUNCTION TABLE
                01150 ;
80BE C4         01160 FTAB     DEFB    'D'+80H      ;DEFINE MESSAGE
80BF D3         01170          DEFB    'S'+80H      ;DEFINE SPEED
80C0 D2         01180          DEFB    'R'+80H      ;TRANSMIT RANDOM
80C1 B0         01190          DEFB    '0'+80H      ;TRANSMIT MESSAGE 0
80C2 B1         01200          DEFB    '1'+80H      ;                 1
80C3 B2         01210          DEFB    '2'+80H      ;                 2
80C4 B3         01220          DEFB    '3'+80H      ;                 3
80C5 B4         01230          DEFB    '4'+80H      ;                 4
80C6 B5         01240          DEFB    '5'+80H      ;                 5
80C7 B6         01250          DEFB    '6'+80H      ;                 6
80C8 B7         01260          DEFB    '7'+80H      ;                 7
80C9 B8         01270          DEFB    '8'+80H      ;                 8
80CA B9         01280          DEFB    '9'+80H      ;                 9
80CB D0         01290          DEFB    'P'+80H      ;SET PRINT
80CC CE         01300          DEFB    'N'+80H      ;RESET PRINT
000F            01310 FTABS    EQU     $-FTAB       ;SIZE OF FUNCTION TABLE
                01320 ;
                01330 ; BRANCH TABLE
                01340 ;
80CD EB80       01350 BTAB     DEFW    DEFINE       ;DEFINE MESSAGE
80CF 9181       01360          DEFW    SPEED        ;DEFINE SPEED
80D1 F081       01370          DEFW    RANDOM       ;TRANSMIT RANDOM
80D3 3782       01380          DEFW    XMIT         ;TRANSMIT MESSAGE 0
80D5 3782       01390          DEFW    XMIT         ;                 1
80D7 3782       01400          DEFW    XMIT         ;                 2
80D9 3782       01410          DEFW    XMIT         ;                 3
80DB 3782       01420          DEFW    XMIT         ;                 4
80DD 3782       01430          DEFW    XMIT         ;                 5
80DF 3782       01440          DEFW    XMIT         ;                 6
80E1 3782       01450          DEFW    XMIT         ;                 7
80E3 3782       01460          DEFW    XMIT         ;                 8
80E5 3782       01470          DEFW    XMIT         ;                 9
80E7 8B82       01480          DEFW    PRINT        ;SET PRINT
80E9 B182       01490          DEFW    NOPRNT       ;RESET PRINT
```

15 TWO-BYTE ENTRIES IN BTAB. POSITION OF EACH CORRESPONDS TO FTAB ENTRY

Figure 8-7. Jump Table

The index of the processing routine is determined by finding an entry in another table, which typically holds a set of one-character function codes.

In the code, FTAB holds all of the possible input characters that define special functions in the MORG program. The 80H represents a SHIFT. When the user inputs a SHIFT D, for example, processing of the Define Message must take place; when a SHIFT 8 is input, message 8 must be transmitted.

The FTAB is first **scanned** for the input code character. If it's found, the **index** to FTAB is then used to get the branch address from the BTAB. The code for the scan is shown in Figure 8-8.

SETUP AND SCAN
FTAB FOR INPUT CODE
CHARACTER

```
807D B0               MOR022  OR     B                  ;MERGE SHIFT BIT
807E 21CC80                   LD     HL,FTAB+FTABS-1    ;FUNCTION TABLE END ADD
8081 060F                     LD     B,FTABS            ;FUNCTION TABLE SIZE
8083 BE               MOR025  CP     (HL)                 ;TEST FOR FUNCTION
8084 2825                     JR     Z,MOR030             ;GO IF FOUND
8086 2B                       DEC    HL                   ;POINT TO NEXT FUNCTION
8087 10FA                     DJNZ   MOR025               ;CONTINUE
8089 E67F                     AND    7FH                ;RESET BIT 7
                      ; NOT FUNCTION HERE - TRANSMIT A SINGLE CHARACTER
808B 32DC85                   LD     (TMP1),A           ;STORE IN TEMP BUFFER
808E 3EFF                     LD     A,OFFH             ;-1
8090 32DD85                   LD     (TMP1+1),A         ;STORE TERMINATOR
8093 21DC85                   LD     HL,TMP1            ;ADDRESS OF "MESSAGE"
8096 7E               MOR027  LD     A,(HL)               ;GET NEXT CHARACTER
8097 FE20                     CP     ' '                  ;TEST FOR NON-ASCII
8099 FA6480                   JP     M,MOR020             ;GO IF END OF MESSAGE
809C CDBE83                   CALL   DISCHR               ;DISPLAY CHARACTER
809F CD8183                   CALL   LPRINT               ;PRINT IF REQUIRED
80A2 CDE082                   CALL   SNDCHR               ;SEND CHARACTER
80A5 23                       INC    HL                   ;POINT TO NEXT
80A6 CD2984                   CALL   INPUT                ;TEST FOR INPUT
80A9 18EB                     JR     MOR027               ;CONTINUE SENDING
80AB 48               MOR030  LD     C,B                ;INDEX+1 NOW IN C
80AC 0D                       DEC    C                  ;ADJUST FOR INDEX
80AD 0600                     LD     B,0                ;INDEX NOW IN BC
80AF CB21                     SLA    C                  ;2*INDEX NOW IN BC
80B1 DD21CD80                 LD     IX,BTAB            ;BRANCH TABLE
80B5 DD09                     ADD    IX,BC              ;POINT TO BRANCH
80B7 DD6601                   LD     H,(IX+1)           ;GET MSB OF ADDRESS
80BA DD6E00                   LD     L,(IX)             ;GET LSB OF ADDRESS
80BD E9                       JP     (HL)               ;BRANCH OUT
```

CONVERT FTAB INDEX
TO BTAB INDEX, GET
ADDRESS AND BRANCH
OUT

Figure 8-8. Jump Table Use

At MOR025 , A holds the input character, and B holds the FTAB size, FTABS. FTABS is automatically computed at assembly time. The HL register holds a pointer to the end of the FTAB, set up by loading HL with FTAB+FTABS-1.

Hints and Kinks 8-1

Automatic Table Size

Programmers commonly use an expression like FTABS EQU $-FTAB to have the assembler calculate table size. Entries can then be added in the table without having to change program constants. FTABS will be stored in the assembler symbol table with a value equal to the number of bytes in the table. As symbol table entries are 16 bits, this method works even for very long tables.

A CP is done to search the table. If the entry in FTAB does not compare, HL is decremented to point to the next lower entry. A DJNZ is then done back to MOR025. If the contents of B have been decremented down to zero, all entries have been compared and the search is unsuccessful. If the entry is found, the instruction at location MOR030 is executed.

At MOR030, the B register contains an **index** value to the FTAB character of 0 through 15 (FTABS). This index is transferred to BC, and then multiplied by two (SLA C). The IX register is now loaded with the start of BTAB, and BC is added to IX to point to the entry in BTAB corresponding to the FTAB entry. Note at this point, that IX only points to the BTAB entry; we have not picked up any address.

The next two instructions load H with the most significant byte of the entry and load L with the least significant byte of the entry. HL now contains the address of the processing routine from BTAB and a JP (HL) causes a jump out to the processing routine.

Using a branch table in this fashion is much cleaner than doing the equivalent **in-line** code of —

```
CP  'D'+80H   ;TEST FOR DEFINE
JP  Z,DEFINE  ;GO IF DEFINE
CP  'S'+80H   ;TEST FOR SPEED
```

```
JP  Z,SPEED  ;GO IF SPEED
.
.
```

Hints and Kinks 8-2
When to Use Branch Tables

Admittedly, there's a lot of ''overhead'' in branch table processing. One major difficulty is picking up a branch address. It would be nice to be able to use indexed addressing to pick up a 16-bit value instead of picking up the value one byte at a time.

Let's analyze the approach here. One way to branch out would have been

```
CP  'D'+80H   ;IS THIS DEFINE MESSAGE
JP  Z,DEFINE  ;GO IF YES
CP  'S'+80H   ;IS THIS DEFINE SPEED
JP  Z,SPEED   ;GO IF YES
```

This approach is clean and simple to debug. You would have used five bytes for each branch. The code shown in Figure 8-8 uses about 75 bytes, counting table storage. In this one example, then, 15 branch points would have been the ''break even'' point in terms of memory storage. Of course, there's also the factors of difficulty of coding and debugging, but it seems reasonable to use branch tables for anything over 20 or so entries. (Or 15 or so if you're writing a book on assembly language.).

Scanning

In the code above, we searched or scanned the FTAB for the key value. In this particular case, we scanned **backwards**. The usual procedure is to scan **forwards** through the table. The code of Figure 8-9 is a "Find Message" subroutine that searches a table of messages for a given message number. The format of the Message Table MTAB is shown in Figure 8-10. It consists of ASCII characters, message numbers of 0 through 9, or -1 terminator values.

```
                ;*******************FNDMSG SUBROUTINE*******************
                ;*   FINDS MESSAGE BY SCANNING MBUF FOR 0-9. 0-9 IS     *
                ;*   MESSAGE #. ASCII CHARACTERS ARE CHARACTERS TO      *
                ;*   BE SENT. -1 IS PADDING AT END OF MESSAGE.          *
                ;*   ENTRY: (A)=MESSAGE #                               *
                ;*   EXIT: (HL)=POINTER TO MESSAGE IF Z OR              *
                ;*               POINTER TO NEXT AVAILABLE IF NZ        *
                ;*               RESET                                  *
                ;*   ALL REGISTERS SAVED EXCEPT HL                      *
                ;
82C7 C5         FNDMSG  PUSH    BC              ;SAVE BC
82C8 CDD782             CALL    FNDSR           ;SEARCH
82CB 2807               JR      Z,FND020        ;GO IF FOUND
82CD F5                 PUSH    AF              ;SAVE NOT FND FLAG
82CE 3EFF               LD      A,0FFH          ;FOR FIRST AVAILABLE
82D0 CDD782             CALL    FNDSR           ;SEARCH
82D3 F1                 POP     AF              ;GET FLAG
82D4 C1         FND020  POP     BC              ;RESTORE BC
82D5 2B                 DEC     HL              ;ADJUST HL
82D6 C9                 RET                     ;RETURN
82D7 21EA88     FNDSR   LD      HL,MBUF         ;START OF MSG BUFFER
82DA 010B0A             LD      BC,2571         ;SIZE OF MBUF
82DD EDB1               CPIR                    ;COMPARE
82DF C9                 RET                     ;RETURN
```

LOOK FOR
GIVEN MESSAGE #

GIVEN MESSAGE #
NOT FOUND, LOOK
FOR NEXT AVAILABLE

Figure 8-9. Scanning Example

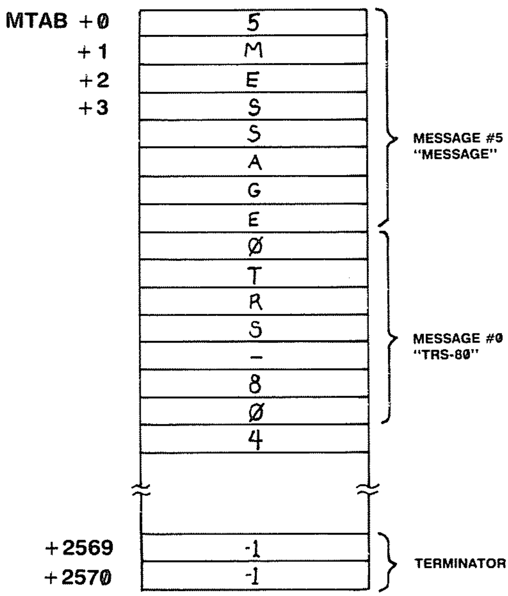


Figure 8-10. MTAB Format

You do the scan by a "block compare" instruction that compares the contents of A (the message #) with each byte of the table starting with MBUF and continuing through 2571 bytes. If the entry in MBUF compares, the Z flag is set, and HL points to the entry **plus one**.

Hints and Kinks 8–3
Block Compares

The setup for a block compare looks like this:

```
(HL) = Start of data for compare
(BC) = Number of bytes in comparison area
(A)  = Search value
```

You then execute a CPIR instruction. It'll go through the block of memory for the number of bytes specified in BC and compare each memory byte with the search value. If a match is found, CPIR will stop with the Z flag set, and HL pointing one byte past the matching value. (The reason for this is that the increment of HL is done before the comparison.) If the search value is not found in the memory block, the Z flag will not be set after the CPIR, and HL will point to the last byte plus one.

The CPDR performs much the same action but searches the block backwards. The CPI and CPD perform one comparison at a time, requiring a loop to be made back to the CPI or CPD.

The first part of the code looks for the given message number. If it isn't found (NZ), another CALL is made to FNDSR to search for the first -1 terminator. On exit, HL points to the message number if found or the first -1 byte. The Z flag is set if the message number was found.

The block compare instructions may be used conveniently to search a table of one-byte entries forwards or backwards (CPDR). They're very fast in comparison to other code sequences.

Ordered Tables

All of the above examples involved tables of **unordered** data. Scanning forwards or backwards through the table located one entry, an item at a time, until a match was found, or until the end (or beginning) of the table was reached. In the following discussion, we'll be concerned with operations on **ordered** tables.

Hints and Kinks 8-4

Ordered Tables

Tables are usually ordered in ascending order based on numeric ''weight.'' One-byte entries pose no problem. Two-byte entry tables for numeric values will probably be in standard Z-80 16-bit address format, least significant byte followed by most significant byte. Greater than two-byte entries are not often found for numeric tables, but are common for strings.

ASCII strings in tables are also usually ordered on the basis of numeric weight. This means that you will use the hex equivalent of the ASCII value in determining whether one value is smaller than the next. This puts upper-case characters before lower case (TRS-80 before tRS-80, for example and BIV,ROY G. after BIV ROY G).

The techniques for ordered tables (data entries are in **ascending** or **descending** order) are geared towards fast **searches** to find specific data items and **sorts** to organize the data in orderly fashion.

Searching

Suppose that we have a table of **ordered data**. We'll assume the table is made up of fixed-length entries, and it's a fixed number of entries. How do we search the entries in the table?

Sequential Search

The first way, of course, would be to scan the table **sequentially** as we have been doing. Even though assembly language is fast, the **search time** for a sequential search may become quite significant when large amounts of data are to be processed. We might be using the sequential search for a sort, for example. The entries in a table would be consecutively searched for the next smallest item and put into a second, sorted table. If there were 1000 entries, we'd have to search all 1000 to find the next smallest item, and we'd have to do that for 1000 items. This would mean 1000*1000 **iterations**, or about 1,000,000 iterations. If each iteration took 70 microseconds (about 12 instruction times), the total sort time would be 70 microseconds * 1,000,000, or 70 seconds! Although this is the blink of an eye compared to the equivalent BASIC code, there is an occasional need for fast searches of ordered data. One of the "standard" high-speed search methods is the **binary search**.

Binary Search

In this search, the midpoint of the entries is compared to the **search key**. If the entry is greater than the search key, the top half of the table is discarded and the next comparison is made in the midpoint of the bottom half; if the entry is less than the sort key, the bottom half of the table is discarded and the next comparison is made in the midpoint of the top half. This division by 2 continues for smaller and smaller segments of the table until the entry is found or until the last segment (one entry!) has been compared and no match has been made (see Figure 8-11).

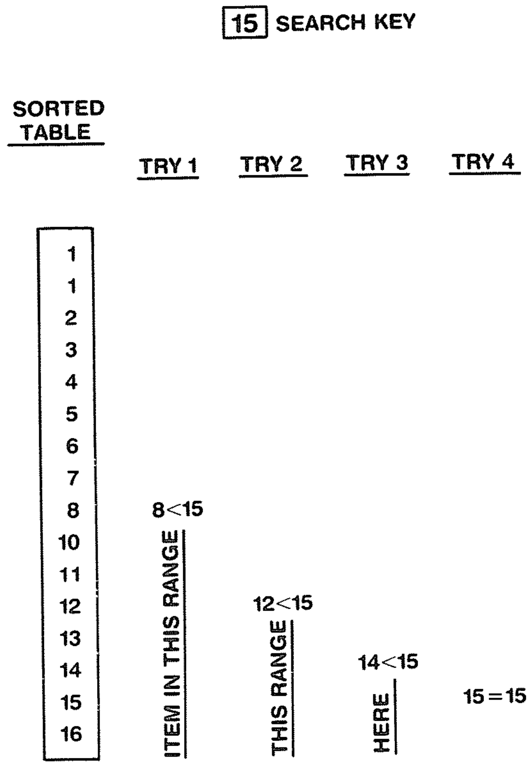


Figure 8-11. Binary Search Algorithm

Figure 8-12 shows this binary search technique implemented in an assembly-language program. BINSRC is designed to search a table made up of 16-bit entries for a given search value. The entries are ordered in ascending fashion and are in standard Z-80 16-bit format, least significant byte followed by most significant byte.

```
8000                     ORG     8000H
               ;*****************************************************
               ;*       BINARY SEARCH OF 16-BIT ENTRY TABLE         *
               ;* ENTRY: (DE)=# OF ENTRIES                          *
               ;*        (BC)=SEARCH VALUE                          *
               ;*        (HL)=START OF TABLE                        *
               ;* EXIT:  (HL)=ENTRY LOCATION OR -1 IF NOT FOUND     *
               ;*        ALL OTHER REGISTERS SAVE                   *
               ;*****************************************************
8000 F5        BINSRC    PUSH    AF              ;SAVE REGISTERS
8001 D5                  PUSH    DE
8002 DDE5                PUSH    IX
8004 226680              LD      (START),HL      ;SAVE TABLE START
8007 ED535E80            LD      (HI),DE         ;HIGH VALUE
800B 110000              LD      DE,0            ;ZERO
800E ED536080            LD      (LO),DE         ;LOW VALUE
```

```
8012 ED5B6080 00260 BIN010  LD     DE,(LO)         ;GET LOW VALUE
8016 2A5E80   00270         LD     HL,(HI)         ;GET HIGH VALUE
8019 B7       00280         OR     A               ;CLEAR CARRY
801A ED52     00290         SBC    HL,DE           ;FIND HIGH-LOW
801C CB3C     00300         SRL    H               ;MSB/2
801E CB1D     00310         RR     L               ;(HI-LOW)/2
8020 226480   00320         LD     (INCM),HL       ;SAVE INCREMENT/2
8023 19       00330         ADD    HL,DE           ;LO+1/2INC
8024 226280   00340         LD     (MID),HL        ;SAVE MIDPOINT
8027 29       00350         ADD    HL,HL           ;MID*2 FOR WORD ADDRESS
8028 ED5B6680 00360         LD     DE,(START)      ;START OF TABLE
802C 19       00370         ADD    HL,DE           ;START + MIDPOINT ADD
802D E5       00380         PUSH   HL              ;PUT IN IX
802E DDE1     00390         POP    IX
8030 DD6601   00400         LD     H,(IX+1)        ;GET MSB BYTE
8033 DD6E00   00410         LD     L,(IX)          ;GET LS BYTE
8036 B7       00420         OR     A               ;CLEAR CARRY
8037 ED42     00430         SBC    HL,BC           ;TEST VALUE
8039 2A6280   00440         LD     HL,(MID)        ;SETUP FOR STORE
803C 2813     00450         JR     Z,BIN030        ;GO IF FOUND
803E 300C     00460         JR     NC,BIN020       ;GO IF LOW
8040 226080   00470         LD     (LO),HL         ;NEW LOW
8043 2A6480   00480 BIN015  LD     HL,(INCM)       ;GET INCREMENT
8046 7C       00490         LD     A,H             ;0 FOR SMALLEST TEST
8047 B5       00500         OR     L               ;CLEAR CARRY
8048 280F     00520         JR     Z,BIN040        ;GO IF END
804A 18C6     00530         JR     BIN010          ;LOOP
804C 225E80   00540 BIN020  LD     (HI),HL         ;NEW HIGH
804F 18F2     00550         JR     BIN015          ;LOOP
8051 DDE5     00560 BIN030  PUSH   IX             ;ADDRESS
8053 E1       00570         POP    HL             ;FOR RETURN
8054 DDE1     00580 BIN035  POP    IX             ;RESTORE REGISTERS
8056 D1       00590         POP    DE
8057 F1       00600         POP    AF
8058 C9       00610         RET                   ;RETURN
8059 21FFFF   00620 BIN040  LD     HL,-1          ;DUMMY FOR NOT FOUND
805C 18F6     00630         JR     BIN035         ;GO TO RETURN
0002          00640 HI      DEFS   2
0002          00650 LO      DEFS   2
0002          00660 MID     DEFS   2
0002          00670 INCM    DEFS   2
0002          00680 START   DEFS   2
0000          00690         END
00000 TOTAL ERRORS
```

Figure 8-12. Binary Search Routine

The routine is entered with DE holding the number of **entries** in the table, BC holding the 16-bit search value, and HL pointing to the table start. A binary search will be made of the table. If the value in BC is found, HL is returned with a pointer to the table entry. If there is no corresponding table entry, HL is returned with a "not found" value of -1.

Data in this table must be 16-bit **unsigned** values. All compares will be of unsigned data (8000H is greater than 7FFFH and FFFFH is greater than FFFEH, for example).

BINSRC works as follows. Five variables are used — HI, LO, MID, INCM, and START. START is simply the table start from entry. HI holds the current high index value defining the top of the range. LO holds the low index value. MID holds (HI-LO)/2 + LO, which points to the next "test" value in the middle of the range. INCM is (HI-LO)/2. INCM gets smaller and smaller as the search "zeroes in" on a value.

The routine starts with HI equal to the number of entries and LO equal to 0. For each iteration, HI-LO is found, divided by two, and added to LO. This gives the **index value** for the comparison. This value is multiplied by two and added to START to find the actual table address.

After the address is found, the table entry is compared to the BC search value. If the table value is less, the MIDpoint value replaces LO; if the table value is more, the MIDpoint value replaces HI. Either way, one half of the current range is discarded.

The "halving" process continues until the search value is found (BIN030) or until the increment INCM is zero, indicating that HI-LO=1. (The test is made after the comparison.)

With the proper parameter passing setup, BINSRC could be used to search a BASIC integer array, as the array would be made up of 16-bit values. Use VARPTR to find the array address.

There are other types of searches, but the binary search and sequential searches are the most commonly used in assembly language. Many searches use the sequential scan since it's the easiest to implement.

Sorting

Now the question arises, "How did the data in a table get sorted in the first place?" There are a number of ways you can sort data — the brute-force two-buffer sort, bubble sorts, binary-insertion sorts, the Shell-Metzner sort, and many others. We'll discuss the first two, which should handle most assembly-language applications.

The Brute-Force Two-Buffer Sort

In this sort we require two buffers. The second is the same size as the first. Here you scan the first "buffer," which is the table of items to be sorted, sequentially for the smallest data item. When it's found, it's put into the second buffer in the next position. The item is then "blanked out" in the first buffer.

This sort requires a lot of memory space because of the two buffers and is relatively slow because of the complete scan required for every item. On the other hand, it's simple in concept and easy to debug.

The assembly-language subroutine for the sort is shown in Figure 8-13. Here again, the sort works with 16-bit data values. Why 16 bits? The 8-bit case is virtually useless, and a version that handles long strings is much more complicated. The 16-bit version is complicated enough due to 16-bit comparisons that must be done and other unwieldy 16-bit operations.

```
8000                         ORG     8000H
                     ;*************************************************************
                     ;*          TWO BUFFER SORT OF 16-'IT ENTRIES                *
                     ;* ENTRY: (IX)=BUFFER 1 ADDRESS                              *
                     ;*        (IY)=BUFFER 2 ADDRESS                              *
                     ;*        (DE)=NUMBER OF ENTRIES                             *
                     ;* EXIT:  BUFFER1 ENTRIES SORTED IN BUFFER2. BUFFER1         *
                     ;*        DESTROYED                                          *
                     ;* NOTE: ENTRIES OF 0 ARE NOT ALLOWED                        *
                     ;*       ALL REGISTERS *DESTROYED*                           *
                     ;*************************************************************
8000 DD225B80        TWOBUF  LD      (BUF1),IX       ;SAVE START
8004 ED535D80                LD      (COUNT),DE      ;SAVE COUNT
8008 DD2A5B80        TWB005  LD      IX,(BUF1)       ;LOAD START
800C 210000                  LD      HL,0            ;ZERO
800F 226180                  LD      (CURLOC),HL     ;CURRENT LOCATION
8012 21FFFF                  LD      HL,0FFFFH       ;FOR SMALLEST
8015 225F80                  LD      (CURVAL),HL     ;CURRENT VALUE
8018 ED5B5D80                LD      DE,(COUNT)      ;GET COUNT
801C DD6601          TWB010  LD      H,(IX+1)        ;GET MSB
801F DD6E00                  LD      L,(IX)          ;GET LSB
8022 7C                      LD      A,H             ;TEST FOR 0 ENTRY
8023 B5                      OR      L
8024 2811                    JR      Z,TWB030        ;GO IF ZERO
8026 ED4B5F80                LD      BC,(CURVAL)     ;GET CURRENT VALUE
802A B7                      OR      A               ;CLEAR CARRY
802B ED42                    SBC     HL,BC           ;COMPARE
802D 3008                    JR      NC,TWB030       ;GO IF CURRENT LARGER
802F 09                      ADD     HL,BC           ;RESTORE NEW VALUE
8030 225F80                  LD      (CURVAL),HL     ;NEW SMALLEST
8033 DD226180                LD      (CURLOC),IX     ;NEW LOCATION
8037 DD23            TWB030  INC     IX              ;POINT TO NEXT
8039 DD23                    INC     IX
803B 1B                      DEC     DE              ;DECREMENT COUNT
803C 7A                      LD      A,D             ;TEST FOR ZERO
803D B3                      OR      E
803E 20DC                    JR      NZ,TWB010       ;GO IF NOT END
8040 2A6180                  LD      HL,(CURLOC)     ;GET CURRENT LOCATION
8043 7C                      LD      A,H             ;TEST FOR 0
8044 B5                      OR      L
8045 C8                      RET     Z               ;RETURN IF ALL BLANKED
8046 AF                      XOR     A               ;ZERO TO A
8047 3E00                    LD      A,0             ;0 TO A, DON'T SET CARRY
8049 77                      LD      (HL),A          ;BLANK ENTRY
804A 23                      INC     HL
804B 77                      LD      (HL),A
804C 2A5F80                  LD      HL,(CURVAL)     ;GET CURRENT VALUE
804F FD7401                  LD      (IY+1),H        ;STORE MSB
8052 FD7500                  LD      (IY),L          ;STORE LSB
8055 FD23                    INC     IY              ;BUMP BUFFER2 PNTR
8057 FD23                    INC     IY
8059 18AD                    JR      TWB005          ;CONTINUE
0002                 BUF1    DEFS    2
0002                 COUNT   DEFS    2
0002                 CURVAL  DEFS    2
0002                 CURLOC  DEFS    2
0000                         END
00000 TOTAL ERRORS
```

Figure 8-13. Two-Buffer Sort Routine

The routine is entered with IX and IY pointing to the two buffers. IX points to the buffer containing the unsorted data; IY points to a second buffer holding the sorted data. As the entries in the IX buffer are moved to the IY buffer, they are "blanked" by filling with a zero value. For this reason, zero is not a valid value in the IX buffer, unless you'd like it to be ignored.

DE contains the number of entries in each of the buffers. The routine first scans the IX buffer until it finds the smallest entry. A dummy count of 0FFFFH is initially put in variable CURVAL to guarantee that a smaller entry will be found. Each entry in the IX buffer is compared with the smallest entry. At the end of the scan, the smallest entry is in CURVAL and its address is in CURLOC. The entry is now blanked by filling its position with 0000H. The entry is then stored in the next position in the IY buffer. The process is repeated until all entries in the IX buffer have been blanked. Figure 8-14 shows the sort action.

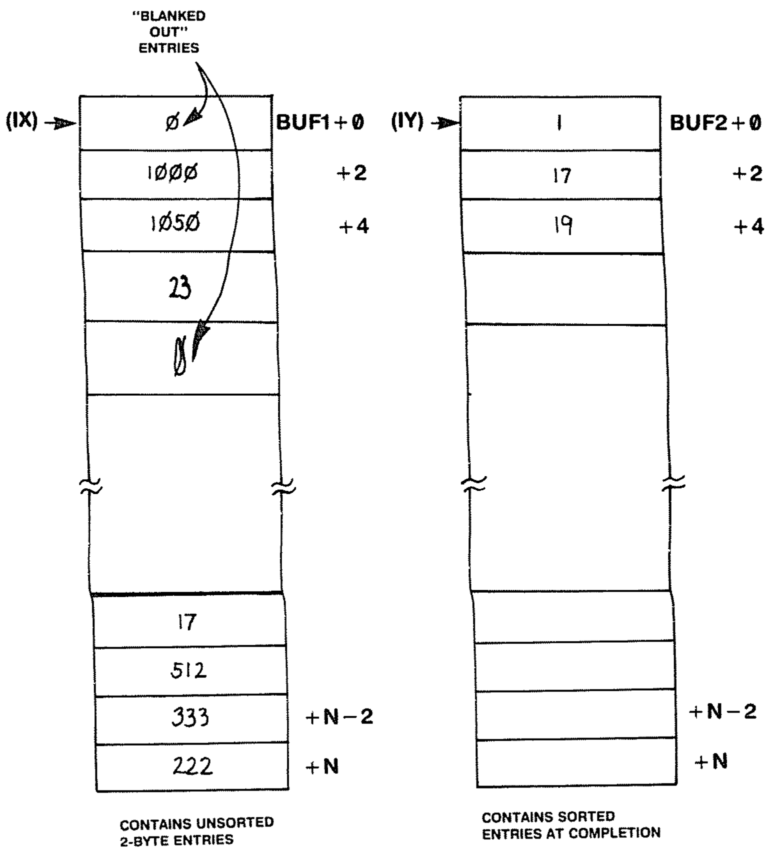


Figure 8-14. Two-Buffer Sort Action

Hints and Kinks 8–5
Two–Buffer Sort Speed and Storage

The two–buffer sort is very expensive in terms of memory storage since it must use another buffer the same size as the first to do its dirty work. How fast is it? Because the sort must pass completely through the first buffer for each entry, it requires *n* passes to sort, where *n* is the number of entries. Each iteration requires approximately 35 instructions. If we estimate about 5 microseconds per instruction, that means each pass takes about 175 microseconds. For a thousand entry table, this is a little under a fifth of a second. Quite a difference from BASIC – we can afford to be ''sloppy'' in high–speed code such as this. Nevertheless, many sorts are run ''off–line'' in commercial programming departments during hours when only the maintenance crew is around.

The Bubble Sort

The "bubble" sort is another sorting technique. It uses only one buffer and is therefore efficient in memory storage. The bubble sort works with two entries of the table at a time. It switches the entries if the second entry is of lower value than the first. In this way, the "lighter" entries bubble to the top. After each pass through the table, another pass is made. If no switches have occurred after any pass, the sort is done (See Figure 8-15).

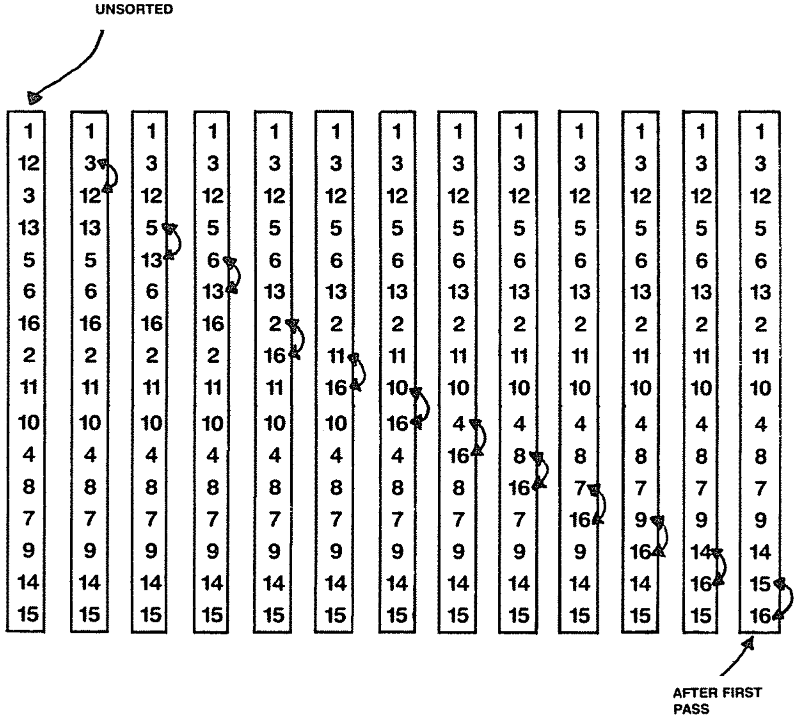


Figure 8-15. Bubble Sort Action

The assembly-language code for the bubble sort is shown in Figure 8-16. The bubble sort is an elegant, clean code but is somewhat slow for all its grace. Pass after pass is made through the buffer, and each adjacent set of entries is compared. If the "top" element is larger than the "bottom," the two are swapped. A change flag (CHNG) is set if any swap is made. If no swap has been made in any pass, the swapping is over, and the sort is done. The BUBBLE routine here is a good example of the power of indexing.

The BUBBLE routine is a great one to run with the contents of the buffer equated to video display memory if you'd like to see a graphic example of the sorting process. Set IX equal to 3C00H and DE equal to **200H** (there are two bytes per entry). You'll have to call the routine from another bit of assembly-language code, or "dummy up" a CALL from DEBUG, such as `CD 00 80 C3 03 A0` (CALL 8000H and Endless Loop at A003H).

```
8000                        ORG     8000H
                  ;*************************************************************
                  ;*                       BUBBLE SORT                         *
                  ;*  ENTRY: (IX)=BUFFER                                       *
                  ;*         (DE)=NUMBER OF ENTRIES IN 16-BIT ENTRY TABLE      *
                  ;*  EXIT:  BUFFER CONTAINS SORTED ENTRIES                    *
                  ;*  ALL REGISTERS *DESTROYED*                                *
                  ;*************************************************************
8000 DD224A80     BUBBLE    LD      (BUFFER),IX     ;SAVE START
8004 1B                     DEC     DE              ;DECREMENT COUNT
8005 ED534C80               LD      (COUNT),DE      ;SAVE FOR PASSES
8009 AF           BUB010    XOR     A                 ;"CHANGE" FLAG
800A 324E80                 LD      (CHNG),A          ;TO ZERO
800D ED4B4C80               LD      BC,(COUNT)        ;LOAD #-1
8011 DD2A4A80               LD      IX,(BUFFER)       ;GET START OF BUFFER
8015 DD6601       BUB015    LD      H,(IX+1)            ;GET MSB OF FIRST
8018 DD6E00                 LD      L,(IX)              ;GET LSB OF FIRST
801B DD5603                 LD      D,(IX+3)            ;GET MSB OF SECOND
801E DD5E02                 LD      E,(IX+2)            ;GET LSB OF SECOND
8021 B7                     OR      A                   ;RESET CARRY
8022 ED52                   SBC     HL,DE               ;COMPARE BY SUB
8024 2814                   JR      Z,BUB020            ;DON'T SWAP IF EQUAL
8026 3812                   JR      C,BUB020            ;GO IF FIRST SMALLER
8028 19                     ADD     HL,DE               ;RESTORE HL
8029 DD7201                 LD      (IX+1),D            ;SWAP FIRST WITH SECOND
802C DD7300                 LD      (IX),E
802F DD7403                 LD      (IX+3),H
8032 DD7502                 LD      (IX+2),L
8035 3E01                   LD      A,1                 ;NON-ZERO
8037 324E80                 LD      (CHNG),A            ;SET CHANGE FLAG
803A DD23         BUB020    INC     IX                  ;BUMP PNTR
803C DD23                   INC     IX
803E 0B                     DEC     BC                  ;DECREMENT COUNT
803F 78                     LD      A,B                 ;TEST COUNT
8040 B1                     OR      C
8041 20D2                   JR      NZ,BUB015           ;GO IF NOT LAST
8043 3A4E80                 LD      A,(CHNG)          ;GET FLAG
8046 B7                     OR      A                 ;TEST CHANGE
8047 C8                     RET     Z                 ;RETURN IF NONE
8048 18BF                   JR      BUB010            ;AT LEAST ONE CHANGE
0002              BUFFER    DEFS    2
0002              COUNT     DEFS    2
0001              CHNG      DEFS    1
0000                        END
00000 TOTAL ERRORS
```

Figure 8-16. Bubble Sort Routine

Hints and Kinks 8-6
Bubble Sort Speed and Storage

The bubble sort is very efficient in terms of memory storage since it works within the given buffer and uses very little storage elsewhere. It's a good sort to use when space is at a premium.

However, speed can be another matter. There is no fixed number of iterations through the table with a bubble sort. The best case would be one pass, when all entries are already sorted. The worst case (I believe) is the case where the entries are completely reversed with the ''lighter'' elements at the bottom of the table, water-sogged. If even one element in the table must be moved from bottom to top, it will take *n*–1 iterations, where *n* is the size of the table in entries. A rough estimate of the time for our bubble sort would be 25 instructions per iteration at 5 microseconds per instruction, or 125 microseconds per iteration. Moving one element from bottom to top in a 1000-entry table would require about an eighth of a second. (Still not bad compared to BASIC . . . .)

There are a number of other sorts that could be used in assembly language, but they're somewhat more complex to code and should be used only if you'd really like to crank out every last bit of speed in your assembly-language sorting. Hopefully these sorts have whetted your interest.

Chapter Nine

Graphics Display Processing

Several chapters ago we looked at some of the techniques used to display character data on the screen. Here, we'll talk about graphics display processing — how to put graphics characters on the screen, how to draw shapes and lines, and how to "animate" screen images.

Graphics Characteristics

Character Data Storage

Most of you are familiar with the scheme used in the TRS-80 for displaying graphics, but we'll go over it again before we talk about graphics processing. There're 1024 **character positions** on the screen, arranged in 16 lines of 64 characters each. In the "upper case" version of the TRS-80, video memory stores only seven bits of data for each character stored. As a matter of fact, all character data must have the most significant bit, (bit 7), set to 0, so the video memory only uses six bits to represent the ASCII character codes. The "missing bit" is bit 6. Figure 9-1 shows how the 64 valid ASCII codes are stored in memory.

CHARACTER	ASCII	MEMORY*	CHARACTER	ASCII	MEMORY*
SP	2ØH	2ØH	@	4ØH	ØØH
!	21	21	A	41	Ø1
"	22	22	B	42	Ø2
#	23	23	C	43	Ø3
$	24	24	D	44	Ø4
%	25	25	E	45	Ø5
&	26	26	F	46	Ø6
'	27	27	G	47	Ø7
(	28	28	H	48	Ø8
)	29	29	I	49	Ø9
*	2A	2A	J	4A	ØA
+	2B	2B	K	4B	ØB
,	2C	2C	L	4C	ØC
–	2D	2D	M	4D	ØD
.	2E	2E	N	4E	ØE
/	2F	2F	O	4F	ØF
0	3Ø	3Ø	P	5Ø	1Ø
1	31	31	Q	51	11
2	32	32	R	52	12
3	33	33	S	53	13
4	34	34	T	54	14
5	35	35	U	55	15
6	36	36	V	56	16
7	37	37	W	57	17
8	38	38	X	58	18
9	39	39	Y	59	19
:	3A	3A	Z	5A	1A
;	3B	3B	↑	5B	1B
<	3C	3C	↓	5C	1C
=	3D	3D	←	5D	1D
>	3E	3E	→	5E	1E
?	3F	3F	___	5F	1F

*MEMORY IS UPPER CASE WITHOUT LOWER-CASE OPTION

Figure 9-1. Video Memory Codes

When the video memory byte holding character data is read, the memory form of the data is converted back to ASCII by setting bit 6 if bit 5 AND bit 7 are 0s. Since bit 7 is never set for character data, the video-memory codes are reconverted back to ASCII as shown in Figure 9-2.

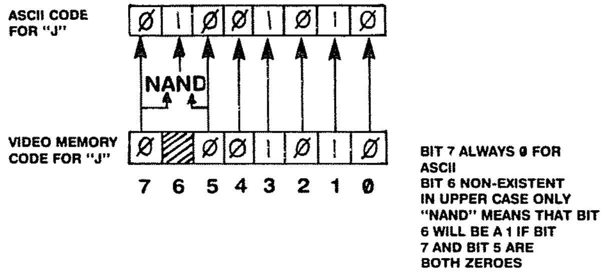


Figure 9-2. Reconversion From Video Memory

When character data from 00000000 through 00011111 (the control codes) or from 01100000 through 01111111 (lower case) are stored in video memory, they lose bit 6 and become the characters shown in Figure 9-1. Attempting to store codes 0 through 127 results in reading back 2 sets of the 64 valid character codes. The moral to this long tale is "Never store anything in upper case video memory except ASCII data (or graphics data, as we'll see) unless you want it to be converted to ASCII!"

Hints and Kinks 9-1
Lower Case Modification

The lower case modification adds another RAM chip to the video memory and a new character generator chip. The RAM chip adds an eighth bit and eliminates the special video memory storage codes; all data is then stored in video memory in 8-bit format. The NAND logic for bit 6 is also deleted.

Storage of Graphics Data

Graphics data is **stored** exactly the same as character data. Bit 6 is lost in upper case versions of the TRS-80. However, here bit 7 is **set** to indicate that the data does not represent 64 ASCII-like codes, but 64 graphics configurations. The graphics configurations allowed are shown in Figure 9-3, along with their data values.

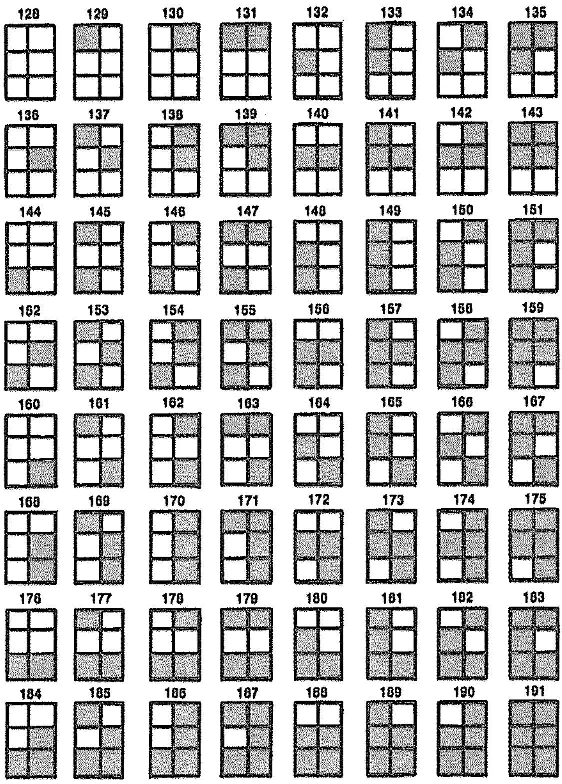


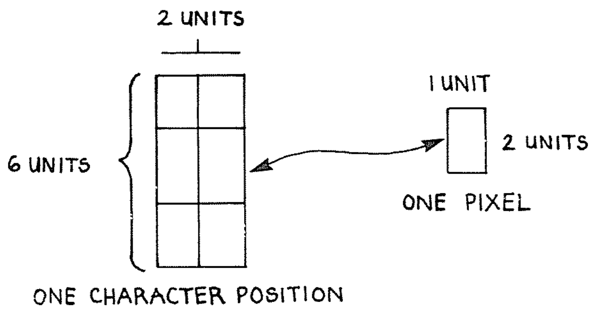
Figure 9-3. Graphics Codes

When the data is read back from video memory for **refresh** (hardware display of memory on the screen), the CPU hardware logic examines bit 7 to see if it's a 0 or 1. If it's a zero, the 6 bits are input to a graphics **character generator** chip that converts the 6 bits to a 5 by 7 graphics character. If it's a one, the character generation is disabled, and each of the six bits produces one **pixel** worth of data, on or off, as shown in Figure 9-4.

Hints and Kinks 9–2
Aspect Ratio

In graphics work, be certain to plot lines and figures using video display worksheets or graph paper that shows the correct ''aspect ratio'' of pixels.

Approximate Relationship (Within 15% Error):



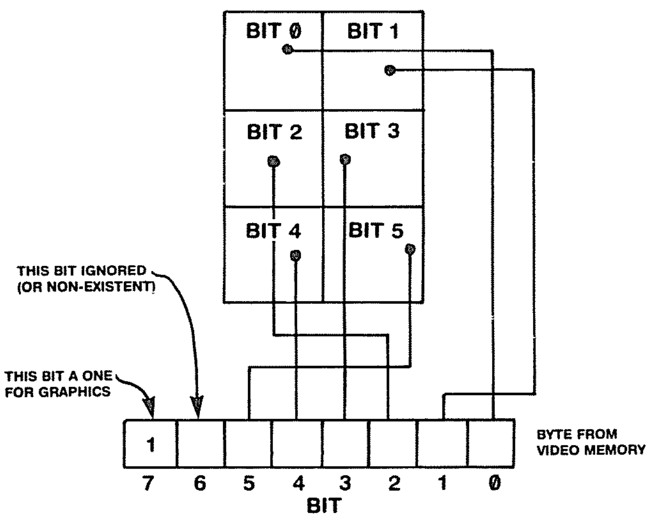


Figure 9-4. Pixel Representation

As each character position on the screen is divided into six pixels, there are 1024*6=6144 pixel positions on the screen, divided into 128 columns and 48 rows, as shown in Figure 9-5.

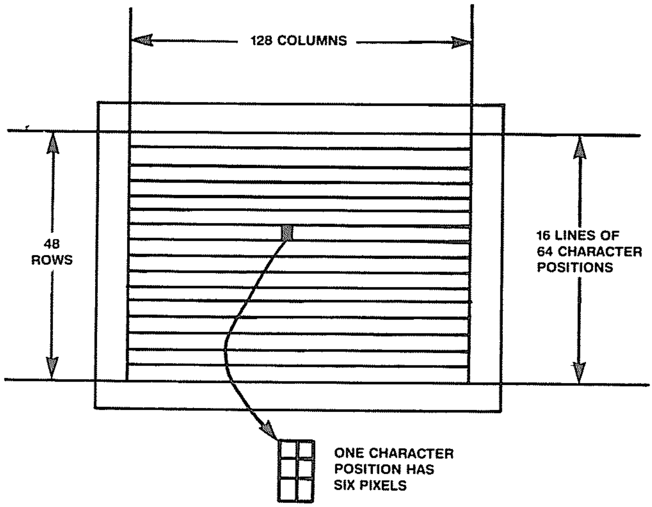


Figure 9-5. Graphics and Character Positions

Character Position and Pixel Addressing

We saw in Chapter 7 that it's very easy to output character data to the screen. The ASCII character you want to display is simply stored in the proper character position. Compute the character-position address by adding the character position number 0 through 1023 to the address of the start of video memory, 3C00H.

Even when you plan to store a character at a specific line and character position within a line, the address computation is simple. The formula for a video-memory address for a line character-position address is:

```
ADDRESS = LINE*64 + CHAR POS + 3C00H
```

where LINE is the line number 0 through 15 and CHAR POS is the character position within the line 0 through 63.

Addressing pixels, however, isn't that easy. As a matter of fact, it's a chore due to the **memory mapping** of the video memory. As our seven-year old TRS-80 afficianado puts it, "Six into eight don't go!"

Random addressing of pixels requires an x,y address, where x = 0 through 127 and y = 0 through 47. This is familiar to most readers as the BASIC SET/RESET format. Each pixel is in one of the 1024 bytes of video memory; within the byte, the pixel may be in **bit positions** 0 through 5. The task in addressing a pixel is to compute the **byte address** of the byte containing the pixel, followed by the **bit address** of the pixel within the byte.

If we do some thinking about this problem, we can see that the byte address is given by:

```
BYTE ADDRESS=3C00H + (Y/3)Q*64 + (X/2)Q
```

This formula says if we take the Y address and divide by three, the quotient (Q) will represent the **line number** (0 through 15) containing the pixel. Multiplying this line number by 64 will give the **displacement** from the start of video memory of the line. If the X address is divided by two, the quotient will give the **character position** containing the pixel along the line. The actual byte address is now 3C00H + line displacement + character displacement.

We now have the byte address, but what about the **bit address** within the byte? The row address is given by:

```
BIT ADDRESS = (Y/3)R*2 + (X/2)R
```

The remainder (R) of Y/3 gives the **row number** of the bit. Since there are two bits per row, this must be multiplied by two to give the row displacement. The remainder of X/2 gives the **column number** of the bit. Adding this to the row displacement gives the bit position 0 through 6 of the bit.

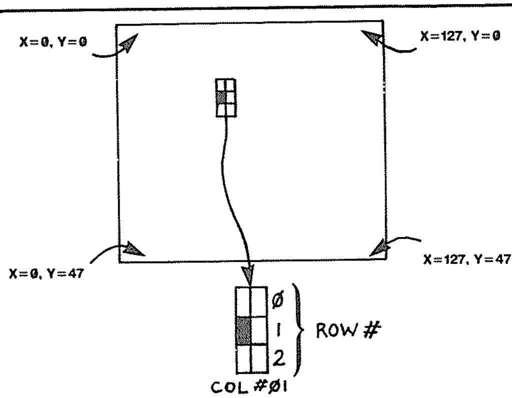
The formulas above are not easy to visualize, but if you draw a sketch of the screen showing character positions and pixel positions and try some examples, you'll see how they were derived. Figure 9-6 shows a recap of pixel addressing.

Hints and Kinks 9-3
Higher Speed SET/RESET

Is a higher-speed SET/RESET possible? A brute force method might employ a table look up indexed by Y*128+X. Each entry in the table would be a byte address of 0–1023 and a bit address of 0–7. This could be stored in two bytes, and the entire table would require 6144*2=12288 bytes.

You could modify this table scheme on a line basis by dividing Y by 16 (easy to do) and finding (Y/16)R*128+X. This index would access a 384-entry table of two bytes each (1068 bytes) containing the byte address and bit address. The actual byte address could be found by: TABLE BYTE ADDRESS + (Y/3)Q*64 + 3C00H.

Estimated speed for this SET/RESET would be somewhere around 7500 points per second.



1. LINE # = $\frac{Y}{3}$ QUOTIENT (EACH LINE OF 16 HAS 3 ROWS OF Xs)
2. ROW # OF CHARACTER = $\frac{Y}{3}$ REMAINDER (ROW # WITHIN THE CHARACTER)
3. CHARACTER POSITION = $\frac{X}{2}$ QUOTIENT (FROM START OF LINE)
4. COLUMN # OF PIXEL = $\frac{X}{2}$ REMAINDER (0 OR 1)
5. BYTE DISPLACEMENT FROM START OF VIDEO MEMORY = (LINE #)*64 + CHARACTER POSITION
6. ACTUAL LOCATION = (LINE #)*64 + CHARACTER POSITION * 3C00H
7. BIT POSITION WITHIN CHARACTER = (ROW #)* 2 + COLUMN NUMBER

Figure 9-6. Pixel Addressing

Random Vs. Character Position Graphics

There are two basic methods of writing out graphics data to the screen: "character position" graphics and "random" graphics.

In random graphics, a point is addressed by its x,y coordinates and either set, reset, or tested. This method of displaying graphics data requires a **graphics driver** program to convert the x,y coordinates into a byte and bit address as we discussed above. Random graphics are used for plotting, animation, line drawing, and the like.

There's a lot of graphics display work, however, that can be done by character-position-oriented graphics. This still involves setting pixels, but the pixels are not addressed randomly. The pixel positions to be set (or reset) are known beforehand, and data is output on a character position basis to produce the graphics.

We'll discuss these methods before getting into the more complicated random techniques.

Character-Oriented Graphics

Horizontal and Vertical Line Drawing

The simplest graphics processing involves drawing a horizontal or vertical line. Since there are three rows in a graphics character position, the horizontal line height can be either 1/3, 2/3, or 3/3 of a character position height. The width of a vertical line can be 1/2 or 2/2 of a character position width.

Figure 9-7 shows some code from the MORG program of Chapter 13 that draws a thick line across line 12 of the screen. LINE12 is equated to 3F00H. The FILLCH (Fill Character) subroutine is used to fill an 8FH for 64 bytes, starting at 3F00H.

```
8026 3E8F      00470       LD      A,0BFH          ;ALL ON GRAPHICS CHAR
8028 11003F    00480       LD      DE,LINE12       ;LINE 12
802B 014000    00490       LD      BC,64           ;# OF BYTES
802E CD8385    00500       CALL    FILLCH          ;DRAW LINE
```

Figure 9-7. Horizontal Line Drawing

When you need to draw a vertical line, the process is similar, except that the "increment" is 64 instead of one. This sets the next character position under the last to draw the line vertically. Code for drawing a vertical line is shown in Figure 9-8 from the DRAWL subroutine of Chapter 14. The graphics character to be used is in the A register.

```
                    00350   ;VERTICAL LINE HERE
000F'   F1          00360   DRA060: POP     AF          ;RESTORE CHARACTER
0010'   11 0040     00370           LD      DE,64       ;INCREMENT
0013'   77          00380   DRA065: LD      (HL),A        ;STORE GRAPHICS
0014'   19          00390           ADD     HL,DE         ;POINT TO NEXT
0015'   10 FC       00400           DJNZ    DRA065        ;GO IF MORE
```

Figure 9-8. Vertical Line Drawing

If you're doing a large amount of line drawing in a program, then it'd be convenient to automate the process somewhat. One approach is to make a table of lines to be drawn. This makes it easy to define new displays and to correct existing ones (another example of a "table-driven" approach).

> Hints and Kinks 9–4
> Graphics Driver
>
> A table-driven ''line drawer'' could be expanded even further. A graphics driver subroutine could decode such table entries as ''draw horizontal line,'' ''draw vertical line,'' ''draw rectangle with given corners,'' ''draw filled in box,'' ''draw diagonal line,'' and so forth. This would be a useful program if you continually do display work, but probably not worth the time otherwise.

We used this method in Chapter 14 for the Tic-Tac-Toe program. The programmer used a "grid table," GRIDT, to define all the lines to be drawn in the Tic-Tac-Toe grid. (There were more than four lines, due to some segments that occupied less than a character position.)

Each entry in the GRIDTB is five bytes long (see Figure 8-4) and defines the graphics character to be used (one byte); a code for horizontal/vertical (one byte); the number of character positions to be used (one byte); and the line starting position (two bytes). The table is terminated by a minus one. The DRAWL subroutine shown in Figure 9-9 is called for each entry in the table to draw a single line, horizontal or vertical.

Although the routines here involved only **setting** pixels, it's easy to see how codes for resetting pixels could also be incorporated into code or table-driven routines.

Resetting the pixels could be done either by clearing the screen with graphics 80H characters (about 1/25th second) and redrawing a new line or by going back over the old line with a "fill" byte of 80H. The routines above allow horizontal or vertical lines to be drawn at rates of about 3000 lines per second (assuming average line lengths of 20 character positions) which would permit fast game displays or other display processing.

```
                       TITLE   DRAWSR
                       ENTRY   DRAWL
               ;*******************************************************
               ;*              DRAW LINE SUBROUTINE                   *
               ;*    DRAWS HORIZONTAL OR VERTICAL LINE                *
               ;*    ENTRY:  (A)=GRAPHICS CHARACTER TO BE USED        *
               ;*            (C)=0 IF HORIZONTAL, NON-ZERO IF VERTICAL*
               ;*            (HL)=SCREEN START POSITION               *
               ;*            (B)=# OF CHARACTER POSITIONS TO BE USED  *
               ;*    EXIT: ALL REGISTERS SAVED                        *
               ;*******************************************************
               ;
0000'  C5      DRAWL:  PUSH    BC              ;SAVE REGISTERS
0001'  D5              PUSH    DE
0002'  E5              PUSH    HL
0003'  F5              PUSH    AF              ;SAVE CHARACTER
0004'  79              LD      A,C             ;GET HORIZ/VERT FLAG
0005'  B7              OR      A               ;TEST
0006'  20 07           JR      NZ,DRA060       ;GO IF VERTICAL
               ; HORIZONTAL LINE HERE
0008'  F1              POP     AF              ;RESTORE CHAR
0009'  77      DRA050: LD      (HL),A            ;STORE GRAPHICS
000A'  23              INC     HL                ;BUMP POINTER
000B'  10 FC           DJNZ    DRA050            ;GO IF MORE
000D'  18 08           JR      DRA090          ;DONE
               ;VERTICAL LINE HERE
000F'  F1      DRA060: POP     AF              ;RESTORE CHARACTER
0010'  11 0040         LD      DE,64           ;INCREMENT
0013'  77      DRA065: LD      (HL),A            ;STORE GRAPHICS
0014'  19              ADD     HL,DE             ;POINT TO NEXT
0015'  10 FC           DJNZ    DRA065            ;GO IF MORE
0017'  E1      DRA090: POP     HL              ;RESTORE REGISTERS
0018'  D1              POP     DE
0019'  C1              POP     BC
001A'  C9              RET                     ;RETURN
                       END
```

Figure 9-9. DRAWL Subroutine

Drawing Patterns and Figures

You can also use graphics done on a character-oriented basis to draw patterns or figures. In this case, you must "plot" the figure on a display worksheet or graph paper, convert it to the proper graphics code, and then output it to the correct character positions.

One of the programming problems that seems to come up time and time again is display of "large format" characters. The Tic-Tac-Toe program of Chapter 15 uses large characters for messages, and it's frequently done in other game programs. Let's look at the techniques involved in this program; it'll illustrate the general approach for displaying other types of patterns.

The first task in producing patterns such as alphanumeric characters is to draw out the patterns to be displayed. Figure 9-10 shows a typical pattern for the "large characters" used in Chapter 14. Each character is an 8 by 6 matrix. The pixels are filled in to produce a pleasing alphabetic or other character. You could use any size matrix, but a multiple of two works out conveniently for the horizontal dimension while a multiple of three works out nicely for the vertical dimension, due to the 2 by 3 **mapping** of pixels to a character position.

Hints and Kinks 9–5

Large Character Format

The approach of defining a matrix of dots to represent alphanumeric and special characters is used to generate characters for video displays, dot matrix printers, and the like. Reference descriptions of ''character generator'' chips to find characters already defined in 5 by 7, 6 by 8, and other formats. You can then easily convert them to tables such as the one used in the Tic-Tac-Toe Program, saving you the work of drawing them out manually.

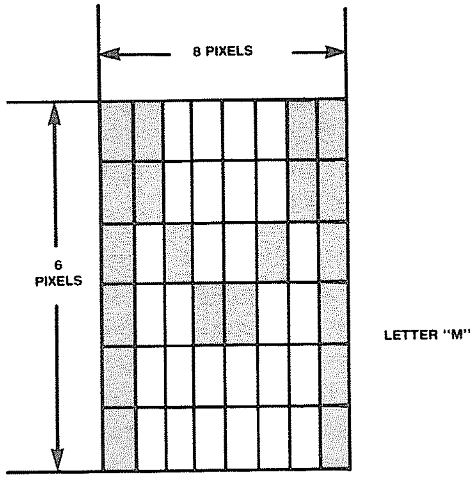


Figure 9-10. "Large" Character Pattern

After you've established the pattern, you convert it to a graphics-character data value. In this case, there are four graphics characters horizontally and two vertically (a total of eight). Each graphics character is then converted to eight bytes as shown in Figure 9-11.

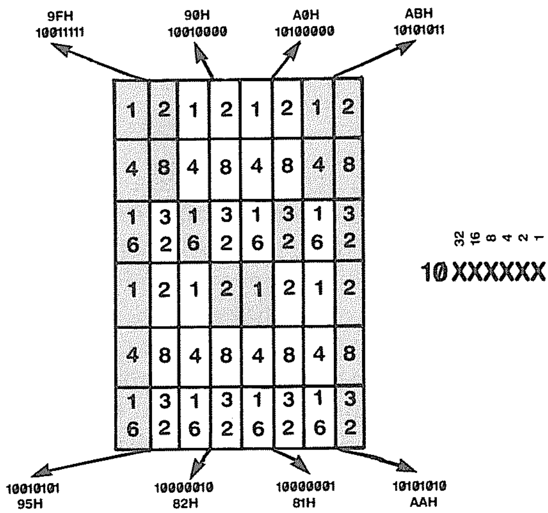


Figure 9-11. Pattern Conversion

Now the eight bytes per character are stored in a table. If the patterns have some numerical sequence, you may store them in logical fashion, such as 0 through 9. In this case, the characters were stored from A through Z, followed by space, -, ?, and !. A second table, CTAB, then holds the ASCII equivalent in the same order, as shown in Figure 9-12. To locate any character data, CTAB is first scanned for the character. When it is found, its index is multiplied by 8 to give the location of the character data in DOTTAB. **Character data in DOTTAB doesn't have the most significant bit set; it's set in the graphics driver routine (LARGEC).**

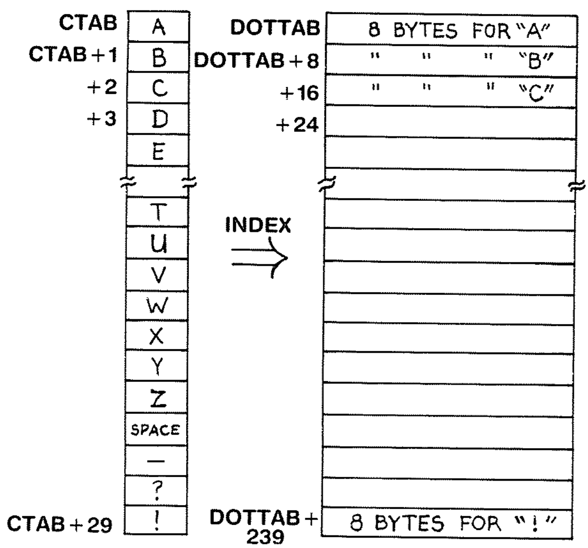


Figure 9-12. CTAB/DOTTAB Relationship

Outputting a large-format character proceeds like so:

1. Search CTAB for the character.
2. Use the CTAB index*8 to find the character data in DOTTAB.
3. Output the first four bytes from DOTTAB to a screen character position start through start+3.

4. Output the next four bytes from DOTTAB to the screen start + 64 through screen start + 67.

The LARGEC subroutine from Chapter 15 is shown in Figure 9-13. It uses the steps above to output a large character to a given screen position. A contains the character to be output in ASCII, while IY contains the screen address. In use, you would write an ASCII character string to the screen by repeated calling of LARGEC with the next ASCII character. Between calls, the screen pointer is incremented by two to provide spacing between characters.

```
                          TITLE   LARGSR
                          ENTRY   LARGEC
                  ;*********************************************************
                  ;*               LARGE CHARACTER DISPLAY                 *
                  ;*  STORES LARGE CHARACTER AT GIVEN SCREEN POSITION      *
                  ;*  8 BY 6 MATRIX USED                                   *
                  ;*  ENTRY:  (A)=CHARACTER TO BE STORED IN ASCII          *
                  ;*          (IY)=SCREEN POSITION                         *
                  ;*  EXIT:   ALL REGISTERS SAVED                          *
                  ;*********************************************************
                  ;
0000'  F5         LARGEC: PUSH    AF              ;SAVE REGISTERS
0001'  C5                 PUSH    BC
0002'  E5                 PUSH    HL
0003'  DD E5              PUSH    IX
0005'  DD 21 0049'        LD      IX,CTAB         ;CHARACTER TABLE ADDRESS
0009'  DD BE 00   LAR020: CP      (IX)              ;TEST FOR CHARACTER
000C'  DD 23              INC     IX                ;BUMP TO NEXT
000E'  20 F9              JR      NZ,LAR020         ;GO IF NOT FOUND
0010'  DD E5              PUSH    IX              ;TRANSFER IX TO HL
0012'  E1                 POP     HL
0013'  01 004A'           LD      BC,CTAB+1       ;CHAR TABLE START+1
0016'  B7                 OR      A               ;CLEAR CARRY
0017'  ED 42              SBC     HL,BC           ;FIND DISPLACEMENT
0019'  E5                 PUSH    HL              ;TRANSFER HL TO IX
001A'  DD E1              POP     IX
001C'  DD 29              ADD     IX,IX           ;INDEX*2
001E'  DD 29              ADD     IX,IX           ;INDEX*4
0020'  DD 29              ADD     IX,IX           ;INDEX*8
0022'  01 0067'           LD      BC,DOTTAB       ;8 BY 6 MATRIX
0025'  DD 09              ADD     IX,BC           ;POINT TO PATTERN
0027'  CD 0038'           CALL    MATSR           ;STORE TOP ROW
002A'  01 003C            LD      BC,60           ;LINE ADJUST
002D'  FD 09              ADD     IY,BC           ;ADD TO SCREEN PNTR
002F'  CD 0038'           CALL    MATSR           ;STORE BOTTOM ROW
0032'  DD E1              POP     IX              ;RESTORE REGISTERS
0034'  E1                 POP     HL
0035'  C1                 POP     BC
0036'  F1                 POP     AF
0037'  C9                 RET                     ;RETURN
0038'  06 04      MATSR:  LD      B,4             ;FOR ROW
003A'  DD 7E 00   MAT010: LD      A,(IX)            ;GET ELEMENTS
003D'  CB FF              SET     7,A               ;SET GRAPHICS
003F'  FD 77 00           LD      (IY),A            ;STORE
0042'  DD 23              INC     IX                ;BUMP ELEMENT PNTR
0044'  FD 23              INC     IY                ;BUMP SCREEN PNTR
0046'  10 F2              DJNZ    MAT010            ;GO IF NOT ENTIRE ROW
0048'  C9                 RET                     ;RETURN
```

Figure 9-13.
LARGEC Routine

The MATSR subroutine within LARGEC writes out four bytes for the row. It is called twice, once with the screen start address, and once with the screen start address + 64. The remainder of the logic in the program involves searching CTAB and accessing DOTTAB.

```
C000            00100          ORG     0C000H
                00400 ;*******************************************************
                00410 ;* SUBROUTINE TO SET OR RESET A PIXEL GIVEN X (0-127)   *
                00420 ;* IN H REGISTER AND Y (0-47) IN L REGISTER.           *
                00430 ;* (A)=0 FOR SET, 1 FOR RESET.                         *
                00440 ;*******************************************************
                00450 ;
C000 F5         00460 SETRST  PUSH    AF                  ;SAVE SET/RESET FLAG
C001 5C         00470         LD      E,H                 ;X
C002 7D         00480         LD      A,L                 ;Y
C003 CB3B       00490         SRL     E                   ;GET CHAR POSITION (0-63) IN E
C005 1600       00500         LD      D,0                 ;SET COL# TO 0
C007 3001       00510         JR      NC,SET10            ;GO IF COL#=0
C009 14         00520         INC     D                   ;COL#=1
C00A 06FF       00530 SET10   LD      B,0FFH              ;-1 TO B
C00C 04         00540 SET20   INC     B                     ;BUMP QUOTIENT IN B=LINE#
C00D D603       00550         SUB     3                     ;SUCCESSIVE SUBT FOR /3
C00F F20CC0     00560         JP      P,SET20               ;GO IF NOT NEGATIVE
C012 C603       00570         ADD     A,3                 ;ADD BACK FOR REMAINDER=ROW#
C014 07         00580         RLCA                        ;(ROW#)*2
C015 82         00590         ADD     A,D                 ;(ROW#)*2+COL#=BIT POS
C016 4F         00600         LD      C,A                 ;SAVE BIT POS IN C
C017 68         00610         LD      L,B                 ;LINE #
C018 2600       00620         LD      H,0                 ;NOW IN HL
C01A 0606       00630         LD      B,6                 ;SHIFT COUNT
C01C 29         00640 SET30   ADD     HL,HL                 ;MULTIPLY LINE#*64
C01D 10FD       00650         DJNZ    SET30                 ;LOOP TIL DONE
C01F 1600       00660         LD      D,0                 ;DE NOW HAS CHAR POS
C021 19         00670         ADD     HL,DE               ;(LINE#)*64+CHAR POS IN HL
C022 11003C     00680         LD      DE,3C00H            ;START OF VIDEO
C025 19         00690         ADD     HL,DE               ;(LINE#)*64+CHAR POS+3C00H
C026 0600       00700         LD      B,0                 ;BC NOW HAS BIT POS
C028 F1         00710         POP     AF                  ;GET SET/RESET FLAG
C029 B7         00720         OR      A                   ;TEST FLAG
C02A 200C       00730         JR      NZ,RESET            ;GO IF RESET
C02C DD2144C0   00740         LD      IX,MASK             ;START OF MASK TABLE
C030 DD09       00750         ADD     IX,BC               ;POINT TO MASK
C032 7E         00760         LD      A,(HL)              ;LOAD PIXEL
C033 DDB600     00770         OR      (IX)                ;SET PIXEL
C036 77         00780 SET36   LD      (HL),A              ;STORE IN VIDEO
C037 C9         00790         RET                         ;RETURN
C038 DD214AC0   00800 RESET   LD      IX,MASK1            ;RESET MASK TABLE
C03C DD09       00810         ADD     IX,BC               ;POINT TO MASK
C03E 7E         00820         LD      A,(HL)              ;LOAD PIXEL
C03F DDA600     00830         AND     (IX)                ;RESET PIXEL
C042 18F2       00840         JR      SET36               ;GO TO STORE, RETURN
C044 81         00850 MASK    DEFB    81H                 ;MASK TABLE
C045 82         00860         DEFB    82H
C046 84         00870         DEFB    84H
C047 88         00880         DEFB    88H
C048 90         00890         DEFB    90H
C049 A0         00900         DEFB    0A0H
C04A FE         00910 MASK1   DEFB    0FEH
C04B FD         00920         DEFB    0FDH
C04C FB         00930         DEFB    0FBH
C04D F7         00940         DEFB    0F7H
C04E EF         00950         DEFB    0EFH
C04F DF         00960         DEFB    0DFH
0000            00970         END
00000 TOTAL ERRORS
```

Figure 9-14. SETRST Routine

Drawing Random Points

Now let's get back to the problem of setting and resetting random points. We know from our previous discussion how to compute the bit and byte address of the pixel, given an x,y coordinate. Figure 9-14 shows a subroutine that will set or reset any given pixel. Entry is made with X (0 through 127) in H and Y (0 through 47) in L. The A register is 0 for a set function and 1 for a reset function.

First compute the address of the byte containing the pixel by the algorithm previously described. Then use the bit position to access a "mask" table of one of six entries. There are two mask tables: one for setting a pixel; one for resetting a pixel. The byte containing the pixel is then ANDed with the mask and stored again to set or reset one of the six pixel bits in the byte location.

You should have first set all character positions that are to be used for graphics to a graphics "null" character 80H to ensure that graphics mode and not character mode is in force.

Hints and Kinks 9-6

If an 80H is Not Used . . .

If you attempt a SET/RESET for a pixel in a character position that has not been initialized by 80H, strange results may occur in an upper case TRS-80. A blank character position could have either 20H (ASCII space) or 80H (graphics null) in video memory. If it has 20H, attempting to SET a point and <u>then</u> turning on bit 7 results in 101X XXXX, where X is the pixel that has been set. This results in two pixels being displayed, one for the SET and one from the 20H! Always clear the area to be used for graphics with 80H before display work!

Animation

This routine takes about 400 microseconds to set or reset a pixel, making the number of pixels that can be processed per second about 2500. This is 20 times or so faster than SET/RESET in BASIC, but still somewhat slow for truly high-speed processing. If a "frame" of a picture was all 6144 pixels, for example, then only one third of a frame per picture could be processed per second. That's also assuming that no other "number-crunching" was being done.

Of course, if the number of active pixels per frame is fewer, then we can process something close to **animation** rates of 16 frames per second. If each new frame of data RESET one half of the points and SET a number of points equal to half on the screen, we'd be SETting and RESETting the average number of points per frame, then, for each frame. If we can process 2500 points per second, that gives us about 156 points per frame, which is somewhat sparse for many pictures, but about the best that can be done. Another problem here is obtaining the **data base** for the animation. (The thought of **digitizing** 100,000 points for 40 seconds worth of Star Warp, or some other game, gives me pause . . . .)

Line Drawing

We've discussed the simple procedures of drawing horizontal and vertical straight lines, but how about **angled lines**?

The BASIC approach to this problem would undoubtedly involve trigonometry and be very slow. However, there is a non-trigonometric approach that would work well in assembly language to permit high-speed processing. The one we'll illustrate here involves minimum math and a somewhat different approach, so bear with us — it'll be fairly high speed and worth the effort.

Figure 9-15 shows a line on the TRS-80 screen. It starts in the upper left and is drawn diagonally at a shallow angle. The distance from X1 to X2 is equal to some "delta X" expressed in pixel units. In this case, X1 is 10, X2 is 20, delta X is 20-10 or 10 units. The distance from Y1 to Y2 is equal to some "delta Y," expressed in the same units. In this case delta Y is 21-15, or 6 units.

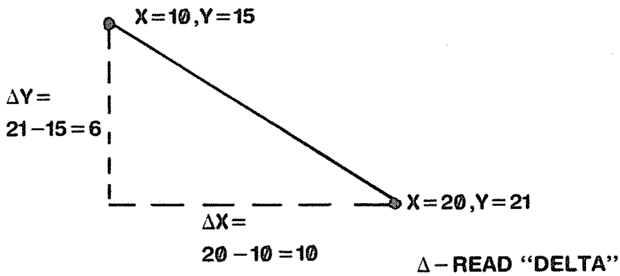


Figure 9-15. Angled Line

If we SET a point at every X value — 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20 — what points would have to be set for Y values? We can easily find out by computing the amount that Y has to be incremented for every new X position. This is:

```
DELTA Y/DELTA X = 6/10 PIXEL
```

In other words, every time we increment X by 1, we increment Y by .6. The points that would be set for this line would therefore be:

X	Y	X	Y
10	15	16	18.6
11	15.6	17	19.2
12	16.2	18	19.8
13	16.8	19	20.4
14	17.4	20	21
15	18.0		

Notice that we wrote some points more than once, but at least the algorithm is straightforward.

There is a minor problem here — how do we work with fractions in assembly language? In BASIC we have the mechanism built into a FOR . . . TO . . . STEP loop. How do we implement it here?

Scaling

We'll **scale** Y upwards by 256. This means we'll hold Y as a number multiplied by 256. The high-order 8 bits will be the integer portion of the number, and the low-order 8 bits will be the fractional part, as shown in Figure 9-16. When we find delta Y/delta X, we'll get a result that is actually (delta Y*256)/X as shown in Figure 9-17. We can then add the result to the 16-bit scaled representation of Y to get each new increment of Y. After every add, we'll take the new value of X and the **integer** (higher 8 bits) of Y to set the new point. The whole process for the points we've been discussing is shown in Figure 9-18.

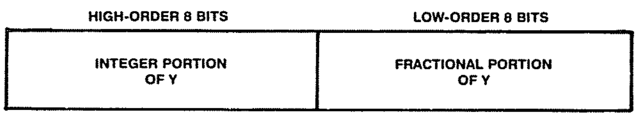


Figure 9-16. Scaling Example

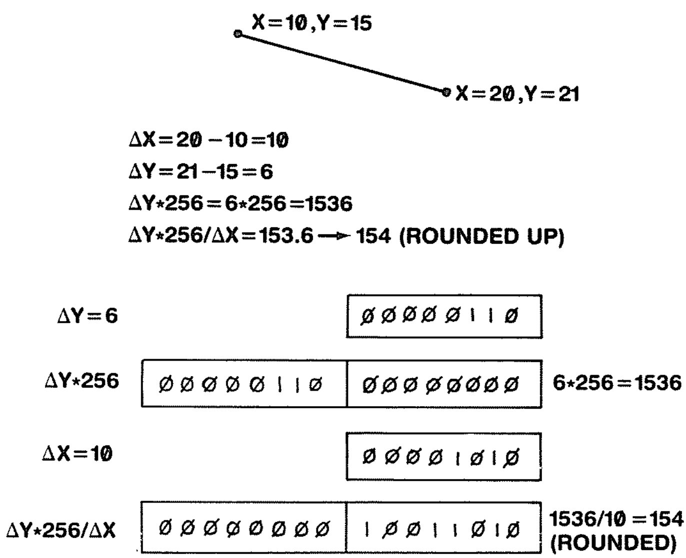
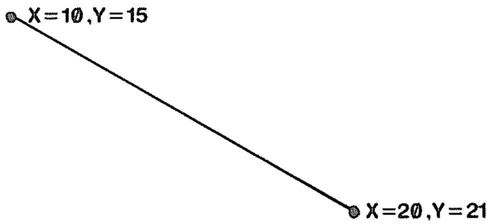


Figure 9-17. Scaling With Division



ΔX=20 −10 =10
ΔY=21−15=6
ΔY*256=6*256=1536
ΔY*256/ΔX=153.6 ⟶ 154 (ROUNDED)

X	Y		16-BIT SCALED VALUE	Y INTEGER
1Ø	ØØØØ1111	00000000	15*256=384Ø	15
11	ØØØØ1111	10011010	384Ø + 154 = 3994	15
12	ØØØ1ØØØØ	00110100	3994 + 154 = 4148	16
13	ØØØ1ØØØØ	11001110	4148 + 154 = 43Ø2	16
14	ØØØ1ØØØ1	01101000	43Ø2 + 154 = 4456	17
15	ØØØ1ØØ1Ø	00000010	4456 + 154 = 461Ø	18
16	ØØØ1ØØ1Ø	10011100	461Ø + 154 = 4764	18
17	ØØØ1ØØ11	00110110	4764 + 154 = 4918	19
18	ØØØ1ØØ11	11010000	4918 + 154 = 5Ø72	19
19	ØØØ1Ø1ØØ	01101010	5Ø72 + 154 = 5226	2Ø
2Ø	ØØØ1Ø1Ø1	00000100	5226 + 154 = 538Ø	21

Figure 9-18. Basic Line Drawing Algorithm

Hints and Kinks 9-7
Scaling

The problem of scaling was not a trivial one in early computer work. (*Digital Computer Programming* by McCracken, 1957, devotes a chapter to ''Decimal Point Location Methods.'') Higher-level languages solved the problem of working with mixed numbers, and most ''number crunching'' applications use a language other than assembly language to avoid laborious scaling processing.

Now this algorithm works fine for the one case we've been discussing, but how about the **general case**? When the angle is more acute, as shown in Figure 9-19, the situation changes. Now delta Y is greater than delta X, and we must increment Y by one and X by some fractional value. Very well — for this case we'll do just that. But wait a minute — how many cases are there?

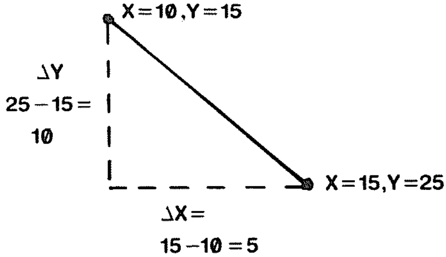


Figure 9-19. Second Case of Angled Line

There are actually eight cases, when we consider the size of delta X to delta Y, and the direction of the line. They're shown in Figure 9-20. Four of these increment X, and four increment Y.

ΔX	ΔY	IΔXI:IΔYI	LINE	X INCREMENT	Y INCREMENT
+	+	+		+1	ΔY/ΔX
+	+	−		ΔX/ΔY	+1
+	−	+		+1	ΔY/ΔX
+	−	−		ΔX/ΔY	−1
−	+	+		−1	ΔY/ΔX
−	+	−		ΔX/ΔY	+1
−	−	+		−1	ΔY/ΔX
−	−	−		ΔX/ΔY	−1

NOTES:
1. ΔX IS X2 − X1
2. ΔY IS Y2 − Y1
3. IΔXI IS ABSOLUTE VALUE
4. IΔYI IS ABSOLUTE VALUE
5. IΔXI:IΔYI IS IΔXI COMPARED TO IΔYI. If +, IΔXI IS LARGER THAN IΔYI, IF −, IΔYI IS LARGER THAN IΔXI
6. INCREMENTS SHOW WHICH VARIABLE WILL BE STEPPED ONE UNIT AT A TIME, AND WHICH WILL BE FRACTIONALLY STEPPED

Figure 9-20. Line Drawing Configurations

The algorithm for the LINE subroutine goes like this:

1. Find delta X by subtracting X2-X1. Call this value DX.
2. Find delta Y by subtracting Y2-Y1. Call this value DY.

3. If the absolute value of DX is greater than the absolute value of DY, PUT 1*256 in XI (X increment), absolute value of DX+1 in CT (count), and absolute value of DY*256/DX in YI (Y increment). X will vary in steps of one in this case.
4. If the absolute value of DX is less than or equal to the absolute value of DY, put 1*256 in YI, absolute value of DY+1 in CT, and absolute value of DX*256/DY in XI. Y will vary in steps of one in this case.
5. If DX is negative, negate XI.
6. If DY is negative, negate YI.
7. Increment X and Y from their starting values for a number of steps equal to CT.

This algorithm (with some nuances) is shown in the BASIC subroutine of Figure 9-21. It does all of the number crunching through step 4 above. You could implement the code in assembly language, but my trial effort was about 120 lines (!). The parameters from the BASIC processing are POKEed into locations FF00H through FF0AH for use by the assembly-language LINE program that does the high-speed line drawing. In this way you have the best of both BASIC and assembly language (and I can avoid explaining 120 lines of code!)

```
10000 DX=X2-X1
10010 DY=Y2-Y1
10020 IF (DX=0) AND (DY=0) THEN XI=0:YI=0:CT=1:GOTO 10070
10030 IF ABS(DX)>ABS(DY) THEN XI=256:CT=ABS(DX)+1:YI=ABS(DY*256/DX)
10040 IF ABS(DX)<=ABS(DY) THEN YI=256:CT=ABS(DY)+1:XI=ABS(DX*256/DY)
10050 IF ABS(DX)<>(CT-1)*XI/256 THEN XI=XI+1
10060 IF ABS(DY)<>(CT-1)*YI/256 THEN YI=YI+1
10070 IF DX<0 THEN POKE &HFF09,1 ELSE POKE &HFF09,0
10080 IF DY<0 THEN POKE &HFF0A,1 ELSE POKE &HFF0A,0
10090 X=X1*256
10100 Y=Y1*256
10110 POKE &HFF00,X-INT(X/256)*256
10120 POKE &HFF01,INT(X/256)
10130 POKE &HFF02,Y-INT(Y/256)*256
10140 POKE &HFF03,INT(Y/256)
10150 POKE &HFF04,XI-INT(XI/256)*256
10160 POKE &HFF05,INT(XI/256)
10170 POKE &HFF06,YI-INT(YI/256)*256
10180 POKE &HFF07,INT(YI/256)
10190 POKE &HFF08,CT
10200 DEFUSR0=&HFF0B
10210 A=USR0(0)
10220 RETURN
```

Figure 9-21. Line Routine in BASIC

The LINE routine is shown in Figure 9-22. It performs steps 5 through 7 of the algorithm. LINE calls the SETRST subroutine described previously. LINE will draw a typical 20-point line in about 10 milliseconds, not including the BASIC processing portion. It will also draw vertical or horizontal lines and points, but not as efficiently as the other techniques described.

```
FF00                        ORG     0FF00H
                  ;**********************************************************
                  ;* LINE SUBROUTINE. DRAWS A STRAIGHT LINE BETWEEN ANY     *
                  ;* TWO GIVEN POINTS. OPERATES IN CONJUNCTION WITH BASIC   *
                  ;* DRIVER PROGRAM. ENTER WITH BLOCK SETUP AS FOLLOWS:     *
                  ;* BLOCK+0,+1: SCALED X VALUE/+2,+3: SCALED Y VALUE/      *
                  ;* +4,+5: ABSOLUTE X INCREMENT, SCALED/+6,+7: ABSOLUTE    *
                  ;* Y INCREMENT, SCALED/+8:COUNT/+9: 1 IF NEGATE X INC     *
                  ;* ELSE 0/+A: 1 IF NEGATE Y INC ELSE 0                    *
                  ;**********************************************************
                  ;
000B              BLOCK   DEFS    11
FF0B 3A09FF       LINE    LD      A,(BLOCK+9)     ;GET INCREMENT SENSE
FF0E B7                   OR      A               ;TEST
FF0F 280D                 JR      Z,LIN010        ;GO IF XINC +
FF11 210000               LD      HL,0            ;ZERO HL
FF14 ED5B04FF             LD      DE,(BLOCK+4)    ;GET XINC
FF18 B7                   OR      A               ;ZERO C
FF19 ED52                 SBC     HL,DE           ;NEGATE
FF1B 2204FF               LD      (BLOCK+4),HL    ;STORE - XINC
FF1E 3A0AFF       LIN010  LD      A,(BLOCK+10)    ;GET INCREMENT SENSE
FF21 B7                   OR      A               ;TEST
FF22 280D                 JR      Z,LIN020        ;GO IF YINC +
FF24 210000               LD      HL,0            ;ZERO HL
FF27 ED5B06FF             LD      DE,(BLOCK+6)    ;GET YINC
FF2B B7                   OR      A               ;ZERO C
FF2C ED52                 SBC     HL,DE           ;NEGATE
FF2E 2206FF               LD      (BLOCK+6),HL    ;STORE - YINC
FF31 DD2100FF     LIN020  LD      IX,BLOCK          ;POINT TO BLOCK
FF35 DD6601               LD      H,(IX+1)          ;GET X
FF38 DD6E03               LD      L,(IX+3)          ;GET Y
FF3B AF                   XOR     A                 ;FOR SET
FF3C CD5FFF               CALL    SETRST            ;SET POINT
FF3F 2A00FF               LD      HL,(BLOCK)        ;CURRENT X
FF42 ED5B04FF             LD      DE,(BLOCK+4)      ;INCREMENT
FF46 19                   ADD     HL,DE             ;BUMP
FF47 2200FF               LD      (BLOCK),HL        ;STORE
FF4A 2A02FF               LD      HL,(BLOCK+2)      ;CURRENT Y
FF4D ED5B06FF             LD      DE,(BLOCK+6)      ;INCREMENT
FF51 19                   ADD     HL,DE             ;BUMP
FF52 2202FF               LD      (BLOCK+2),HL      ;STORE
FF55 3A08FF               LD      A,(BLOCK+8)       ;COUNT
FF58 3D                   DEC     A                 ;DECREMENT
FF59 3208FF               LD      (BLOCK+8),A       ;SAVE
FF5C 20D3                 JR      NZ,LIN020         ;GO IF MORE
FF5E C9                   RET                     ;RETURN TO BASIC
                  ;**********************************************************
                  ;* SUBROUTINE TO SET OR RESET A PIXEL GIVEN X (0-127)     *
                  ;* IN H REGISTER AND Y (0-47) IN L REGISTER.              *
                  ;* (A)=0 FOR SET, 1 FOR RESET.                            *
                  ;**********************************************************
                  ;
FF5F F5           SETRST  PUSH    AF              ;SAVE SET/RESET FLAG
FF60 5C                   LD      E,H             ;X
FF61 7D                   LD      A,L             ;Y
FF62 CB3B                 SRL     E               ;GET CHAR POSITION (0-63) IN E
FF64 1600                 LD      D,0             ;SET COL# TO 0
FF66 3001                 JR      NC,SET10        ;GO IF COL#=0
FF68 14                   INC     D               ;COL#=1
FF69 06FF         SET10   LD      B,0FFH          ;-1 TO B
FF6B 04           SET20   INC     B                 ;BUMP QUOTIENT IN B=LINE#
```

```
FF6C D603     00710          SUB     3            ;SUCCESSIVE SUBT FOR /3
FF6E F26BFF   00720          JP      P,SET20      ;GO IF NOT NEGATIVE
FF71 C603     00730          ADD     A,3         ;ADD BACK FOR REMAINDER=ROW#
FF73 07       00740          RLCA                ;(ROW#)*2
FF74 82       00750          ADD     A,D         ;(ROW#)*2+COL#=BIT POS
FF75 4F       00760          LD      C,A         ;SAVE BIT POS IN C
FF76 68       00770          LD      L,B         ;LINE #
FF77 2600     00780          LD      H,0         ;NOW IN HL
FF79 0606     00790          LD      B,6         ;SHIFT COUNT
FF7B 29       00800 SET30    ADD     HL,HL        ;MULTIPLY LINE#*64
FF7C 10FD     00810          DJNZ    SET30        ;LOOP TIL DONE
FF7E 1600     00820          LD      D,0         ;DE NOW HAS CHAR POS
FF80 19       00830          ADD     HL,DE       ;(LINE#)*64+CHAR POS IN HL
FF81 11003C   00840          LD      DE,3C00H    ;START OF VIDEO
FF84 19       00850          ADD     HL,DE       ;(LINE#)*64+CHAR POS+3C00H
FF85 0600     00860          LD      B,0         ;BC NOW HAS BIT POS
FF87 F1       00870          POP     AF          ;GET SET/RESET FLAG
FF88 B7       00880          OR      A           ;TEST FLAG
FF89 200C     00890          JR      NZ,RESET    ;GO IF RESET
FF8B DD21A3FF 00900          LD      IX,MASK     ;START OF MASK TABLE
FF8F DD09     00910          ADD     IX,BC       ;POINT TO MASK
FF91 7E       00920          LD      A,(HL)      ;LOAD PIXEL
FF92 DDB600   00930          OR      (IX)        ;SET PIXEL
FF95 77       00940 SET36    LD      (HL),A      ;STORE IN VIDEO
FF96 C9       00950          RET                 ;RETURN
FF97 DD21A9FF 00960 RESET    LD      IX,MASK1    ;RESET MASK TABLE
FF9B DD09     00970          ADD     IX,BC       ;POINT TO MASK
FF9D 7E       00980          LD      A,(HL)      ;LOAD PIXEL
FF9E DDA600   00990          AND     (IX)        ;RESET PIXEL
FFA1 18F2     01000          JR      SET36       ;GO TO STORE, RETURN
FFA3 81       01010 MASK     DEFB    81H         ;MASK TABLE
FFA4 82       01020          DEFB    82H
FFA5 84       01030          DEFB    84H
FFA6 88       01040          DEFB    88H
FFA7 90       01050          DEFB    90H
FFA8 A0       01060          DEFB    0A0H
FFA9 FE       01070 MASK1    DEFB    0FEH
FFAA FD       01080          DEFB    0FDH
FFAB FB       01090          DEFB    0FBH
FFAC F7       01100          DEFB    0F7H
FFAD EF       01110          DEFB    0EFH
FFAE DF       01120          DEFB    0DFH
0000          01130          END
00000 TOTAL ERRORS
```

Figure 9-22. LINE Routine

Chapter Ten
Cassette Output, Music, and Parallel Printers

We'll be looking at two types of input/output processing in this chapter: I/O on the cassette **port** and **parallel** printer output. We'll also take a look at system I/O in general.

The TRS-80 reads and writes cassette tape primarily using software drivers rather than hardware logic. You can use these software drivers in ROM to read and write down to a bit at a time from cassette. Because the cassette port is easily addressable at an assembly-language level, you can also use it to generate "square wave" outputs that can be musical tones or other signals.

System parallel printers are addressed differently than the cassette port. We'll see how simple **printer drivers** may be coded.

Input/Output Programming

Many programmers are unnecessarily upset by input/output programming. Part of the reason for this is that I/O is done automatically in large computer systems; the programmer must make special supervisor calls to perform I/O through the operating system used on the system. This approach is necessary because the system is performing many tasks (many "job runs") simultaneously and is increasing the **throughput** by overlapping processing on one job with I/O on another. Another reason for the mysteries of I/O is some programmers feel that it requires an understanding of (ugh!) hardware.

In the TRS-80, we're operating in a different environment than a large multi-programming system (and believe it or not, a **much more** efficient environment on an individual basis). We can perform I/O ourselves without having to go through the operating system; of course, the provision is also there to have the operating system do our work for us. It's user's choice. As far as knowing "hardware," most programmers don't realize the simplicity of the logic involved in interfacing to most computer peripherals. We'll show you how easy it is in this chapter.

Z-80 and TRS-80 Input/Output

There are two basic types of I/O in any microcomputer system — **I/O mapped I/O** and **memory-mapped I/O.**

I/O Mapped I/O

I/O Mapped I/O uses the I/O instructions in the Z-80. There are four that we'll talk about here: IN A,(n); IN r,(C); OUT (n),A; and OUT (C),r. You can accomplish the same thing with all of them: transfer one byte of data between an external I/O device and a CPU register. The `IN` instructions read one byte from the external device into a CPU register, while the `OUT` instructions write one byte from a CPU register to an external device.

Hints and Kinks 10-1

Other I/O Instructions

The other I/O instructions are the ''block'' I/O instructions. They are somewhat similar in action to the block move instructions. Data can be transferred between memory and an I/O device in a block by setting up a block address in HL and a byte count of 1–256 in the B register; the C register holds the address of the I/O device.

Like the block moves, the I/O block instructions may go forward through the block (INIR,OTIR), backward through the block (INDR,OTDR), or ''semi-automatically'' (external loop back to INI, IND, OUTI, or OUTD). The ''automatic'' I/O block instructions cannot be used unless the I/O controller has been designed for such a transfer.

The format of these instructions is shown in Figure 10-1. All use an **I/O address**. This is an eight-bit address that is contained in the I/O instruction itself (IN A,(n);OUT (n),A) or is in the C register (IN r,(C);OUT (C),r). The I/O address can be 0 through 255.

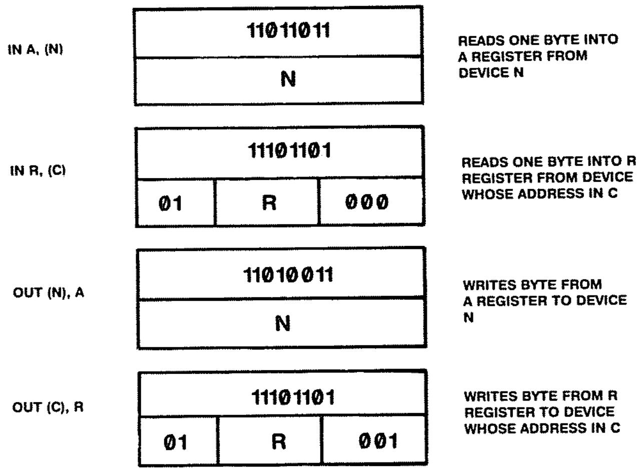


Figure 10-1. I/O Instruction Formats

When an I/O instruction is executed, it goes through a predefined sequence. The I/O address is first sent out along the system address lines A7-A0. Shortly afterwards, data is output along the data lines D7-D0 for the OUT, or input from the data lines D7-D0 for the IN.

External I/O devices are designed to expect this sequence. If the I/O device senses that an I/O instruction is being executed (there is another signal line that performs this function), it reads the address lines to determine if it's being addressed. If it is, it either reads in the byte of data from the data lines or sends back a byte of data to the data lines.

Hints and Kinks 10-2
I/O Mapped I/O Signals

The actual sequence for I/O mapped I/O goes like this: The usual I/O cycle is three cycles: T1, T2, and T3. The ''port address'' is put on address lines A0–A7 during T1. Next, at about T2, the TRS-80 OUT* or IN* signal (one or the other) goes to zero. Each is on a separate line. IN* signifies that an IN instruction is being performed, and OUT* denotes an OUT instruction.

If an input is being done and if an external devices's address is on the address bus, it responds by placing a byte of data on the data bus lines D7–D0. The byte is input to the CPU register at about T3.

If an output is being done and if an external device is being addressed, the device ''strobes in'' the data byte on D7–D0 in T2 or T3. The data was placed on the data lines sometime in T1.

Since there may be 256 separate **port addresses** (I/O addresses in the I/O instructions), there may be as many as 256 separate I/O devices, all looking for their address on the address lines so that they can perform their built in function of reading or writing one byte of data. In practice, there are probably only one or two devices hooked up to any microcomputer system.

Most TRS-80 Model I systems have only two devices. One is the cassette read/write logic (address 0FFH), and the other is the RS-232-C interface board (addresses 0E8H through 0EBH). Although these are somewhat integrated into the TRS-80 system, they are truly external devices viewed from the standpoint of the Z-80 microprocessor.

Model III systems also use cassette and RS-232-C port addresses, but use a number of other ports for disk and system operations.

Memory-Mapped I/O

The second type of I/O that we can have in the TRS-80 is memory-mapped I/O. Here, the external device is still looking for an address on the address lines, but all address lines A15-A0 are used. In this case, the external device does not look for a signal that says "I/O instruction being executed" together with its address, but simply looks for its address. As 16 address lines are being used, 65536 separate addresses could be employed.

> Hints and Kinks 10-3
> Memory-Mapped I/O Signals
>
> As in I/O mapped I/O, this sequence usually takes place in 3 T cycles. First of all, the CPU puts the device address onto the address bus lines A15-A0 during the T1 cycle. If the device is being addressed, it now looks at signals WR* and RD*.
>
> If an input is being done, signal RD* goes low (0) during T2. An external device responds to its address and RD* by placing a byte of data on data bus line D7-D0. If an output is being done, signal WR* goes low during T2. An external device responds to a WR* and address by ''strobing in'' the data on lines D7-D0. The data was placed on these lines by the CPU sometime in T1.
>
> Note that in this type of addressing, the WR* and RD* signals are generated by instructions such as LD (HL),A for an output (WR*) and LD (nn),A for an input (RD*). The I/O device looks identical to a memory location in this mode.

The catch in the above is that some of the addresses are also used for memory! In this type of I/O, there must be a decision by the system designer on how to divide up the 64K worth of addresses into memory and I/O addresses. This division in the TRS-80 is shown in Figure 10-2, which shows the memory mapping for the TRS-80.

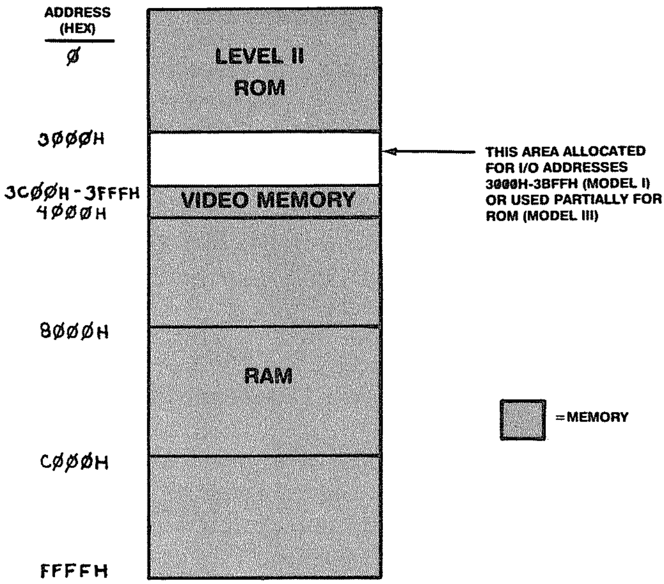


Figure 10-2. TRS-80 Memory Mapping For I/O

Of course, as we know, many of the addresses in the TRS-80 are devoted to video memory (addresses 3C00H through 3FFFH). In fact, this is very similar to regular RAM memory. Other memory-mapped addresses, however, are the system line printer (37E8H), and (in the Model I) the **disk controller chip** (37E0H, 37ECH through 37EFH), and the cassette latch in the expansion interface (37E4H).

To address any of these addresses on a read, we simply perform any **memory reference** instruction, such as LD A,(37E8H) or LD B,(HL). To address any of the addresses on a write, a store is done as in LD (37E8H),A or LD (HL),C. The memory reference instruction, of course, transfers one byte of data between the external (to the Z-80) I/O device and a CPU register.

Just about all I/O operations in the TRS-80 are performed one byte at a time by using either an I/O instruction in I/O mapped I/O, or by using a memory reference instruction in memory-mapped I/O. (Even disk operations are performed one byte at a time.)

Hints and Kinks 10-4
Other Types of I/O

We haven't talked about several other possible types of I/O in the TRS-80 for a good reason – they are not used in standard TRS-80 devices. However, let's mention them anyway, as the provision is there in the TRS-80 as far as bus signals.

The first of these is DMA, direct memory access. In this type of I/O, 8-bit I/O transfers are made directly to and from memory by the I/O device controller, bypassing CPU registers. During each transfer, the CPU is ''locked up'' and the device controller uses the bus lines in a ''cycle-stealing'' mode. This type of I/O is common for very high-speed devices where a software loop can not furnish the data at high enough rates.

The second type of I/O is the interrupt-driven I/O. This type of I/O is usually used with slow-speed devices. Each byte (for example, a keypush) generates an interrupt to the CPU, which causes an interrupt-processing routine to be entered. The interrupt-processing routine contains normal I/O instructions to read in or write out the data. The advantage of this type of I/O is that normal processing can be maintained until the interrupt occurs without any polling or overhead in testing the I/O device for the next byte.

Parallel Printer Operation

The system parallel printer uses memory-mapped I/O with an address of 37E8H. The expansion interface in the Model I or printer logic in the Model III **decodes** this address and passes data either to the printer or from the printer, as shown in Figure 10-3.

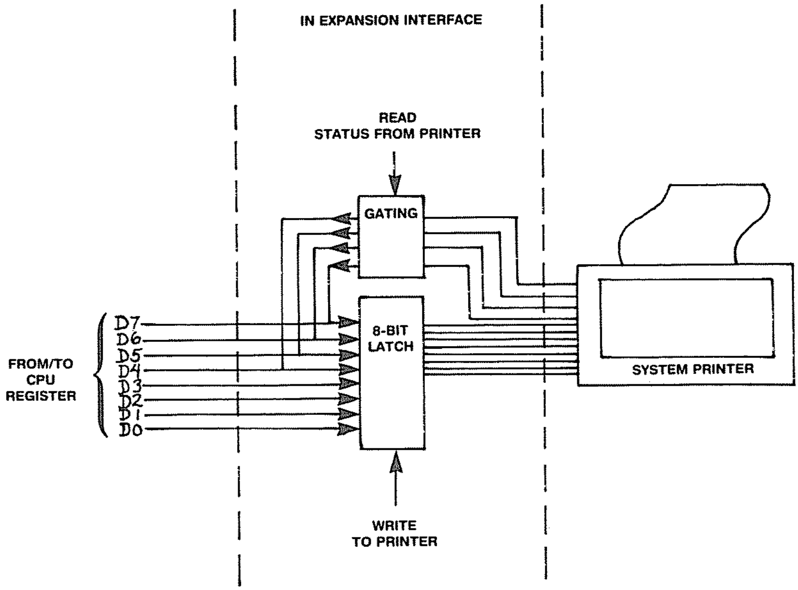


Figure 10-3. Parallel Printer Logic

Printers used on the TRS-80 use a somewhat standard **interfacing** specification called the **Centronics bus**. This specification defines the set of lines used to transfer the data and the signals for **handshaking**, timing, and signal levels.

The handshaking logic for parallel printers goes something like this. The CPU performs a memory-reference read instruction to read a byte of data from the printer **controller** logic. This byte is a **status** byte, as shown in Figure 10-4.

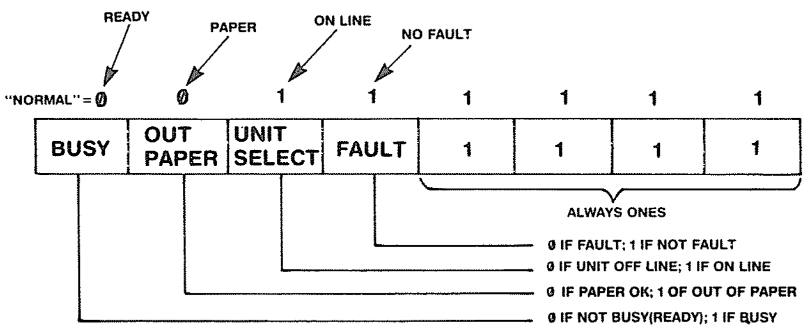


Figure 10-4. Printer Status Byte

The status byte normally contains several bits indicating "fault" conditions for the printer — printer out of paper, off-line, etc. It also contains a bit known as a **ready** bit.

The ready bit signifies that the printer is ready to accept the next byte of data to be printed. In an **unbuffered** printer, the printer is **busy** (not ready) during the time the character is being printed, and the ready bit is not set. After the character has been printed, the ready bit is set by the printer electronics. This situation is shown in Figure 10-5.

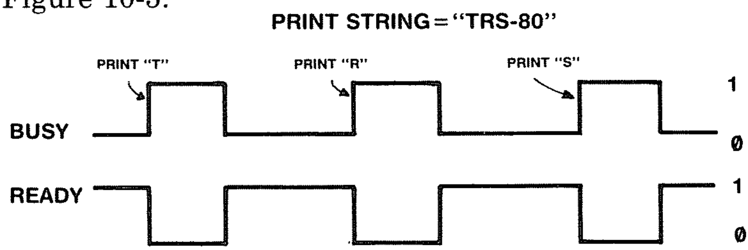


Figure 10-5. Ready/Busy Status

In a **buffered** line printer, the printer may receive the next byte of data to be printed while the printer is in the process of printing. The character is stored in a buffer memory within the printer. Of course, if the printer is printing at a rate of 100 characters per second and the program is outputting characters at rates of 50,000 characters per second, it doesn't take too long for the buffer to become filled up. In this case, the ready bit is reset until one or more characters have been read out of the buffer and printed. Almost all current line printers are of the buffered type.

Figure 10-6 shows a typical line printer **driver** program from the MORG program of Chapter 14. The status is first read from the printer controller logic by a LD A,(37E8H). The status bits are ANDed with 0F0H to mask out the bits from the other bit positions; they contain ones as they connect to nothing for a read from the printer. The other four bits are BUSY, OUTPAPER, UNIT SELECT, and FAULT. If the result is other than 30H, the printer is busy or has a fault condition, and a jump is made back to the read status instruction.

Hints and Kinks 10-5
Other Line Printer Characters

If the line printer is very complex, there may be a variety of ''control codes'' with special meanings that can be output. First of all, there are carriage returns (ODH) and line feeds (OAH). Most printers also have a ''top of form,'' which advances the paper to the next page (OCH).

Another common control code is the ''BEL'' code (07H) which literally rang a bell on early teleprinters; on modern line printers it usually sounds an electronic alarm.

Depending upon the printer, you may have codes for such things as underlining, setting vertical spacing, and setting horizontal print density.

On character-oriented printers – such as the Diablo, Qume, and NEC – you have a whole set of special ''escape sequences'' that control such things as horizontal and vertical spacing in fractions of an inch, vertical and horizontal tabs, and ribbon selection. These sequences are not single codes but are a series of characters, many times started by an ''escape'' character (1BH). If these printers are to be utilized to their fullest advantage, the printer software driver must make provision for outputting such sequences; this may increase the task of writing an assembly-language driver by a factor of ten or more!

```
               ;
               ; LINE PRINTER STATUS AND PRINT CHARACTER SUBROUTINE
               ;
83AF F5        LPSTAT  PUSH    AF              ;SAVE CHAR
83B0 3AE837    LPS010  LD      A,(37E8H)         ;GET STATUS
83B3 E6F0              AND     0F0H              ;MASK OUT GARBAGE BITS
83B5 FE30              CP      30H               ;TEST FOR BUSY
83B7 20F7              JR      NZ,LPS010         ;GO IF BUSY
83B9 F1                POP     AF              ;RESTORE CHAR
83BA 32E837            LD      (37E8H),A       ;OUTPUT
83BD C9                RET                     ;RETURN
```

Figure 10-6. Line Printer Driver Program

If the line printer is ready, the ASCII character is retrieved from the stack and output to address 37E8H by an LD instruction. All data output to the printer must be ASCII data, except for possible special control codes unique to the printer. (A BEL code of 07 would sound an alarm, or a code of 29 might select 20 characters per inch spacing, for example.)

Of course, the line printer driver shown above is the lowest-level subroutine for communication with a system printer. There may be others above it that control message output, format lines, and so forth. This simple **protocol** is typical of many peripheral devices such as line printers, paper tape readers, and card readers.

Cassette I/O

The cassette **controller** is contained in the CPU logic. Like the printer controller, it contains logic to decode its address and to transfer data between a CPU register and the I/O device. While the printer controller uses memory-mapped I/O addressing, however, the cassette logic uses I/O mapped I/O via IN and OUT instructions. The cassette logic is shown in Figure 10-7.

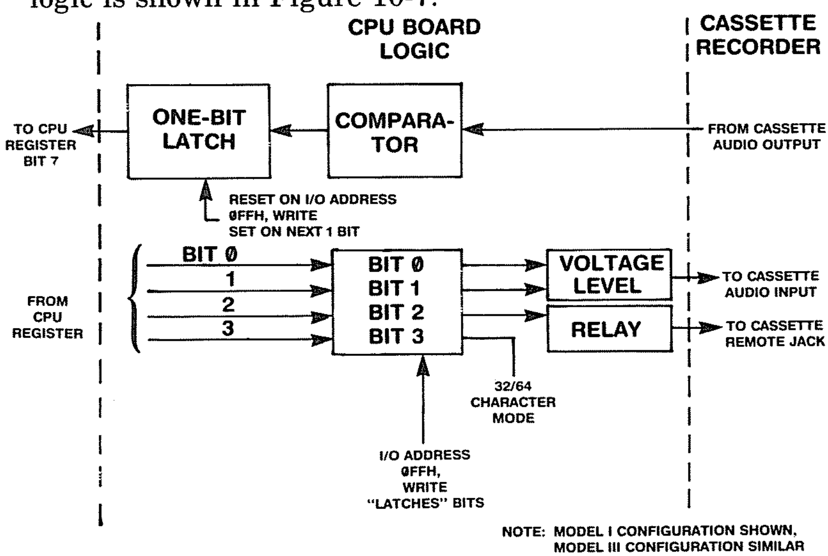


Figure 10-7. Cassette Logic

Cassette Output

When an OUT (0FFH),A or similar instruction is done, the four least significant bits in A (or another CPU register) are transfered to the cassette **latch** shown in Figure 10-7 for the Model I. The latch is really a four-bit memory cell that retains the data until rewritten. The logic for the Model III is similar, but not identical.

Bit 3 of the latch (Model I) controls the 32/64 character mode for the display. Bit 2 of the latch (Model I) controls the cassette relay that would normally be used to turn the cassette motor on and off. Bits 1 and 0 (Model I and III) generate a signal level to the cassette input for writing on cassette. Three signal levels are possible, as shown in Figure 10-8.

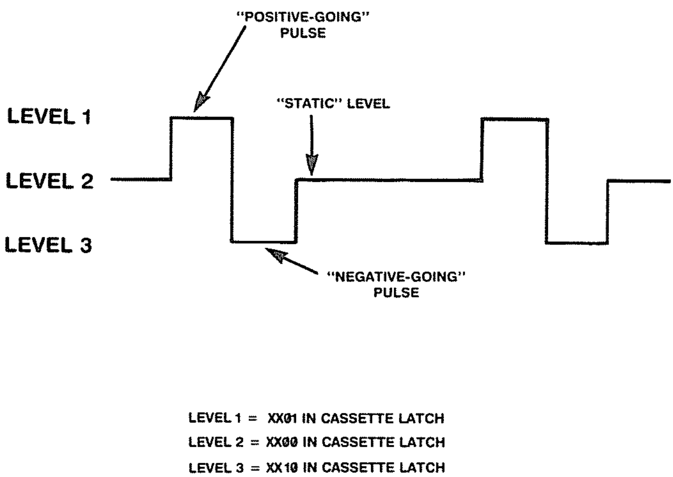


Figure 10-8. Cassette Output Levels

To write on cassette tape, a series of pulses are generated as shown in Figure 10-9. The separation between the pulses is 1 millisecond. The width of each pulse is 250 microseconds, divided into 125 microsecond segments.

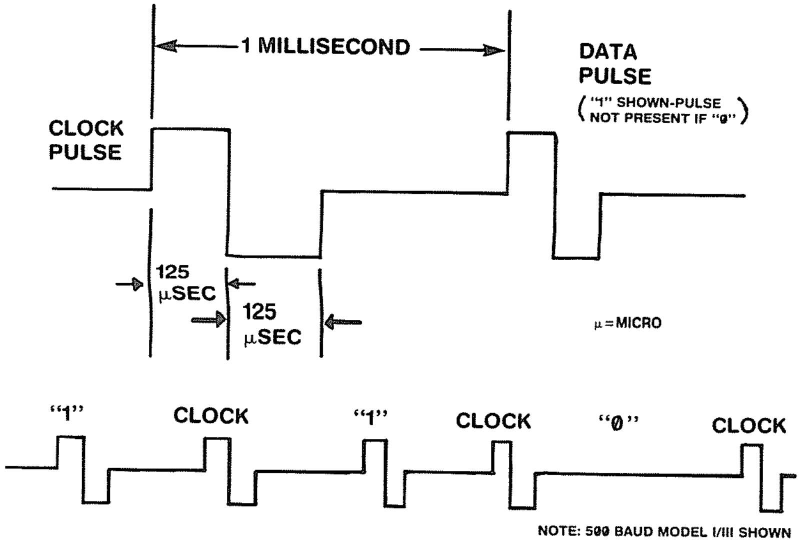


Figure 10-9. Cassette Pulse Formats

There are two pulses for every data bit. Each set of two is divided into a **clocking pulse** and a **data** pulse. There is always a clocking pulse. If there is a data pulse at the 1 millisecond time, a "1" bit is generated. If there is **no data** pulse, a "0" bit is produced, as shown in the figure.

Prior to the generation of pulses, of course, the cassette must have been turned on by a 0000X100 output, which turns on the cassette relay; the X represents the current state of 32/64 character mode (Model I), which must be retained. The positive portion of the cassette is produced by outputting a binary 0000X101, and the negative portion by a binary 0000X110. The reference level is produced by an output of 0000X100.

The normal sequence for writing to cassette is to turn on the motor, write a string of 255 bytes of zeroes (2040 bits of zeroes), and then write a **sync** byte of 0A5H. This really amounts to turning on the motor and then writing zeroes for 2040*2 milliseconds = 4.08 seconds, following with a sync byte. The sync byte is detected in the software read cassette routine and marks the start of all data (and the end of the zero header).

Cassette Input

The cassette read logic consists of a one-bit **read latch** and comparator logic. The software read cassette routine reads one bit from the cassette at a time. This bit is read into bit 7 by performing an IN A,(0FFH) or similar instruction. If the bit is a zero, there is no pulse present from the cassette input; if the bit is a one, there is a pulse present.

The following discussion applies to 500-baud cassette rates in the Model I, but is very similar for Model III operations.

The normal read cassette operation begins with turning the cassette motor on. The cassette read latch is cleared by an IN (0FFH),A (A may have any data). Next a series of IN A,(0FFH) instructions is executed until a one bit is read. This is the start of the clocking pulse. Then a delay about 500 microseconds. This puts the cassette tape past the clock pulse. Then the cassette latch is reset by an IN (0FFH),A. Now another 850 microseconds delay occurs. We're now positioned past where the data pulse should have been. If a data bit had been present, the read latch would now be set. If there is a one after reading the latch by an IN A,(0FFH), the data bit was a one, otherwise the data bit was a zero. After this sequence, the entire process is repeated for the next data bit. Figure 10-10 shows the operation.

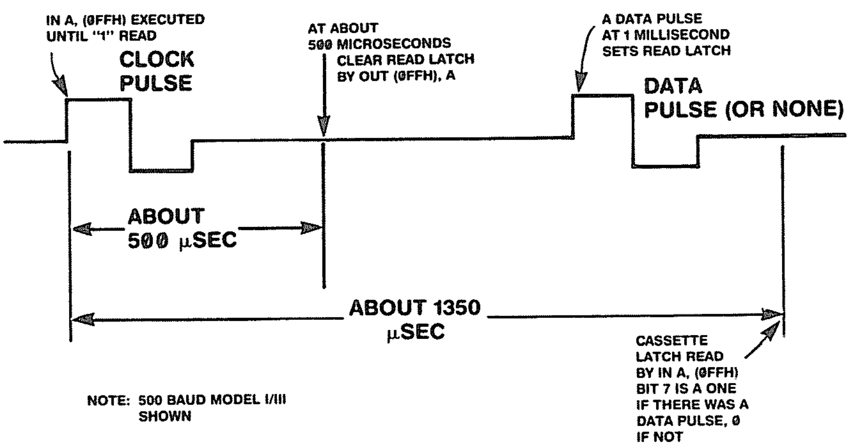


Figure 10-10. Cassette Read Timing

The first data bit to be read will be bit 7 of the 0A5H sync byte (or noise). After the entire 8 bits of the sync byte have been read and verified, the cassette software driver is "synced" to the data and can assemble the bytes of the cassette record from the individual reads.

It's entirely possible to write your own cassette I/O routines from the ground up, addressing the cassette write and read latches. However, there are ROM calls available in the EDTASM manual that you may use to perform the next level of byte-oriented operations. The calls are:

- Define drive and turn on motor
- Turn off motor
- Write leader and sync byte
- Write byte
- Read leader and sync byte
- Read byte

```
Hints and Kinks 10-6
High-Speed Tape Operations

Since the cassette tape is primarily software
driven, is it possible for you to speed up the
cassette data transfer rate? The answer is in
the qualifier ''primarily.'' It is certainly
possible to write assembly-language code to
write and read cassette at 1000 baud or
higher. However, since these frequencies have
an entirely different set of electrical
characteristics and the cassette electronics
are designed to work well with the standard
baud rates, a higher-speed cassette driver
would be a dicey thing at best. Best to invest
in that disk than in an experimental 19,200
baud cassette tape driver!
```

Define Drive, Motor On, Motor Off

A CALL to location 212H with the A register containing a 0 or 1 selects cassette 0 or 1 and turns on the motor. A CALL to location 01F8H deselects the cassette and turns the motor off.

Read Cassette

A CALL to location 0296H looks for the leader of 255 zeroes and the sync byte of 0A5H. It returns after a sync byte has been read. If no sync byte is read, it "hangs," looking for the elusive 0A5H. After this call has been made, a CALL to 0235H will read in the remaining bytes in the cassette record. A byte is returned in the A register.

Write Cassette

A CALL to location 0287H writes the leader and sync byte. A CALL to location 0264H with a data byte in A writes one byte of data.

The above routines may be used in any fashion to create your own tape data formats or to work with existing data formats. All tape operations will be at 500 baud (500 data bits per second) rates. For example, the size of records is normally less than 256 bytes. You may construct your own assembly-language routines to **block** more than one logical record into one physical record.

Figure 10-11 shows an assembly-language routine to write and read video display data in 1024-byte records by utilizing the ROM calls above. This is a relocatable program and is incorporated into the Level II BASIC program shown in Figure 10-12.

```
7F00              00100          ORG     7F00H
                  00110 ;**************************************************************
                  00120 ;*            CASSETTE/VIDEO DUMP/READ ROUTINE                *
                  00130 ;* DUMPS SCREEN TO 1024-BYTE CASSETTE RECORD. READS          *
                  00140 ;* BACK RECORD AND DISPLAYS ON SCREEN.                        *
                  00150 ;* ENTER AT "WRITE" TO DUMP, "READ" TO READ BACK             *
                  00160 ;**************************************************************
                  00170 ;
                  00180 ; SCREEN DUMP
7F00 F3           00185 WRITE   DI                       ;DISABLE RTC
7F01 AF           00190         XOR     A                ;SELECT CASSETTE 0
7F02 CD1202       00200         CALL    212H             ;TURN ON
7F05 CD8702       00210         CALL    287H             ;WRITE LEADER
7F08 21003C       00220         LD      HL,3C00H         ;SCREEN START
7F0B E5           00230 WRT010  PUSH    HL                 ;SAVE PNTR
7F0C 7E           00240         LD      A,(HL)             ;GET VIDEO MEMORY BYTE
7F0D CD6402       00250         CALL    264H               ;WRITE TO CASSETTE
7F10 E1           00260         POP     HL                 ;GET ADDRESS PNTR
7F11 23           00270         INC     HL                 ;BUMP PNTR
7F12 7C           00280         LD      A,H                ;GET MS BYTE
7F13 FE40         00290         CP      40H                ;TEST FOR 4000H
7F15 20F4         00300         JR      NZ,WRT010          ;GO IF NOT END
7F17 CDF801       00310 WRT020  CALL    1F8H             ;TURN OFF CASSETTE
7F1A C9           00320         RET                      ;RETURN
                  00330 ; READ CASSETTE RECORD
7F1B F3           00335 READ    DI                       ;DISABLE RTC
```

```
7F1C AF                  XOR     A            ;SELECT CASSETTE 0
7F1D CD1202              CALL    212H         ;TURN ON
7F20 CD9602              CALL    296H         ;READ LEADER
7F23 21003C              LD      HL,3C00H     ;SCREEN START
7F26 E5           REA010 PUSH    HL             ;SAVE PNTR
7F27 CD3502              CALL    235H           ;READ BYTE
7F2A E1                  POP     HL             ;GET ADDRESS PNTR
7F2B 77                  LD      (HL),A         ;STORE BYTE ON SCREEN
7F2C 23                  INC     HL             ;BUMP PNTR
7F2D 7C                  LD      A,H            ;GET MS BYTE
7F2E FE40                CP      40H            ;TEST FOR 4000H
7F30 20F4                JR      NZ,REA010      ;GO IF NOT END
7F32 18E3                JR      WRT020       ;END ACTION
0000                     END
00000 TOTAL ERRORS
```

Figure 10-11. Cassette/Video Program

```
50 CLEAR 100
100 'BASIC ROUTINE WITH EMBEDDED VIDEO/CASSETTE PROGRAM
200 A$=STRING$(52,"*")
300 B=VARPTR(A$)
400 B=PEEK(B+2)*256+PEEK(B+1)
500 FOR I=B TO B+51
600 READ A
700 IF I>32767 THEN POKE I-65536,A ELSE POKE I,A
800 NEXT I
900 INPUT "READ(R) OR WRITE(W)";A$
1000 C=B
1100 IF A$="W" GOTO 1300
1200 C=C+27
1300 POKE 16526,C-INT(C/256)*256
1400 POKE 16527,INT(C/256)
1500 INPUT "READY CASSETTE, HIT ENTER";A$
1600 X=USR(0)
1700 GOTO 900
2000 DATA 243,175,205,18,2,205,135,2,33,0,60,229,126,205,100,2
2100 DATA 225,35,124,254,64,32,244,205,248,1,201,243,175,205,18,2
2200 DATA 205,150,2,33,0,60,229,205,53,2,225,119,35,124,254,64
2300 DATA 32,244,24,227
```

Figure 10-12. Cassette/Video BASIC Driver

Cassette Music

As most of you know, the TRS-80 cassette latch is being used to generate musical tones ranging from Morse code to four-voice fugues. The music generated is a constant-level, square-wave tone produced by **toggling** the cassette output bits on and off.

In the simplest case, this involves turning on a positive pulse by 01 and then turning on a negative pulse by 10 bits, as shown in Figure 10-13. The CON subroutine from Chapter 14, for example, toggles the two bits by XORing the current configuration of the bits with 3, which alternates between writing 01 and 10 (see Figure 10-14).

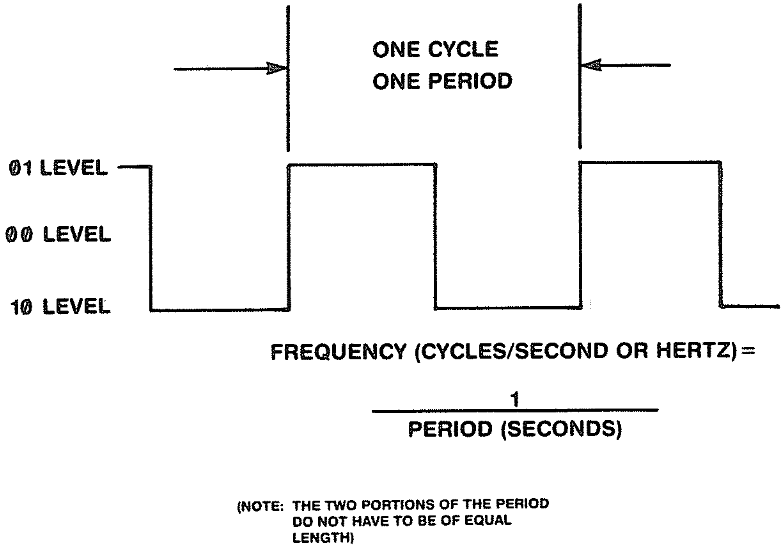


Figure 10-13. Cassette Square Wave

HL CONTAINS A "DURATION COUNT"

```
                ;
                ; ON HERE - GENERATE 500 HERTZ TONE
                ;
833D 3E01       CON     LD      A,01            ;ON
833F 2B                 DEC     HL              ;ADJUST FOR "JR C"
8340 01FFFF             LD      BC,-1           ;DECREMENT
8343 EE03       ON010   XOR     3                 ;TOGGLE
8345 D3FF               OUT     (0FFH),A          ;OUTPUT TO CASSETTE LATCH
8347 E5                 PUSH    HL                ;SAVE COUNT
8348 210100             LD      HL,1              ;FOR 1 MS
834B CDC085             CALL    DELAY             ;DELAY 1 MS
834E CD2984             CALL    INPUT             ;GET POSSIBLE CHARACTER
8351 E1                 POP     HL                ;GET COUNT
8352 09                 ADD     HL,BC             ;DECREMENT COUNT
8353 38EE               JR      C,ON010           ;GO IF NOT -1
8355 C9                 RET                     ;RETURN
```

Figure 10-14. CON Routine

The delay between the outputs is accomplished by calling the DELAY subroutine, which delays 1 millisecond. As the total **period** of the cycle is 2 milliseconds, a 500 hertz tone is produced for the Morse code dots and dashes. A duration count in HL is decremented down to zero to generate the tone for a given length of time.

In general, what range of tones can be generated by using this method of producing sounds? Producing low tones is no problem because we can delay as long as we wish between toggling the cassette latch output. (Of course, the **fidelity** of the output on either high- or low-frequency tones is another matter.)

The problem here is in producing high-frequency sounds. To do this, we must keep a "tight" loop in the code to toggle the cassette latch between positive and negative pulses. As we're talking about a general case "tone driver," we need some means to vary the period to produce different notes.

About the tightest loop that can be used is:

```
        LD      A,01        ;BIT CONFIGURATION
LOOP1   LD      B,C         ;GET DELAY COUNT
        OUT     (0FFH),A    ;TOGGLE ONE WAY
LOOP2   DJNZ    LOOP2       ;DELAY
        XOR     3           ;INVERT BITS
        JP      LOOP1       ;CONTINUE
```

This takes a delay count in C and puts it into B for each one-half cycle. Either a 01 or 10 is then output to the latch at 0FFH (the motor bit and mode select will be zeroes). The count in B is then decremented down to zero, the bit configuration is inverted, and the process is repeated.

The frequencies that you can produce by this code are easy to figure out. The instructions from LOOP1 through the JP LOOP1 take 2.3, 6.2, 7.3/4.5, 3.9, and 5.6 microseconds, respectively. (We took the T cycle times in the EDTASM manual and multiplied by .56 to get the times in microseconds.) The 7.3/4.5 microsecond time represents the DJNZ time for B<>0 and B=0. The total period for any count in C, is therefore:

PERIOD (microseconds)

```
  = (2.3+6.2+(CNT-1)*7.3+4.5+3.9+5.6)*2
  = 43 + 2*CNT
```

The maximum frequency would be for a minimum period when CNT=1, which would be a period of 45 microseconds, representing about 22,000 cycles per second (hertz). The minimum frequency would be for a CNT of 0 (256) and would be a period of 555 microseconds for a frequency of 1800 hertz.

There are several problems with this routine. First of all, we need more than just a continuous tone; we need some way of terminating! That means that we must maintain another count for duration. This count is also necessary to produce different durations for musical notes, if we want to implement a full-fledged music program.

Secondly, it appears that we need to keep a larger count in 16 bits for lower frequency notes. The lowest frequency here is 1800 hertz, which is much too high.

There's also another problem, which is not readily obvious. The frequency **resolution** may not be fine enough. The difference in frequencies for a CNT of 200 and 201 is about 10 hertz. This will probably become more pronounced as we add more overhead to the loop.

Hints and Kinks 10-7
Other Tone Parameters

A full-fledged music synthesizer should include some means to control the volume of the output for ''envelope generation'' or just plain dynamics. About the best we can do with the cassette output on the TRS-80 is to program two levels – one from the 10 level to the 01 level (low to high) and one from the 10 level to the 00 level (low to reference).

Another parameter that has some interesting effects, however, is the ''duty cycle'' of the square wave. We've been working with a 50%-on/50%-off square wave here. However, the harmonic content of square waves will vary with the proportion of on to off time, and you might want to experiment with this parameter as an input for the TONE routine (keeping the period constant).

All this means is that producing musical tones via the cassette port is a compromise (even at best) between range of notes and resolution.

The program in Figure 10-15 is an attempt to produce a wide range of musical notes with good resolution by using two separate routines within the one subroutine, one for high frequencies (HIFREQ) with low overhead and one for low frequencies (LOFREQ) with high overhead.

```
FF00                     ORG     0FF00H
                   ;*************************************************************
                   ;*                  TONE GENERATOR                           *
                   ;* GENERATES HIGH OR LOW FREQUENCY RANGE OF TONES. SYM-      *
                   ;* ETRICAL SQUARE WAVE THROUGH CASSETTE OUTPUT.              *
                   ;* ENTRY: (HL)=DURATION IN # CYCLES AT FREQUENCY             *
                   ;*        (BC)=FREQUENCY COUNT FROM EXTERNAL TABLE           *
                   ;*        (A)=0 IF UP TO 300 HERTZ, 1 IF ABOVE 300 HZ        *
                   ;*************************************************************
0002               DURA    DEFS    2                   ;DURATION POKED BY BASIC
0002               FREQ    DEFS    2                   ;FREQ CNT POKED BY BASIC
0001               FLAG    DEFS    1                   ;FLAG POKED BY BASIC
FF05 F3            TONE    DI                          ;DISABLE RTC
FF06 2A00FF                LD      HL,(0FF00H)         ;GET DURATION
FF09 ED4B02FF              LD      BC,(0FF02H)         ;GET FREQ COUNT
FF0D 3A04FF                LD      A,(0FF04H)          ;GET FLAG
FF10 B7                    OR      A                   ;TEST HIGH,LOW
FF11 2009                  JR      NZ,TON010           ;GO IF HIGH
FF13 E5                    PUSH    HL                  ;DURATION TO IX
FF14 DDE1                  POP     IX
FF16 C5                    PUSH    BC                  ;FREQ CNT TO HL
FF17 E1                    POP     HL
FF18 CD22FF                CALL    LOFREQ              ;GENERATE TONE
FF1B C9                    RET
FF1C 0600          TON010  LD      B,0                 ;CLEAR B OF BC
FF1E CD46FF                CALL    HIFREQ              ;GENERATE TONE
FF21 C9                    RET
                   ; LOW FREQUENCY TONE GENERATOR
FF22 2244FF        LOFREQ  LD      (FCNT),HL           ;STORE FREQ CNT(16)
FF25 11FFFF                LD      DE,-1               ;DECREMENT(10)
FF28 2A44FF        LOOP1   LD      HL,(FCNT)             ;GET FREQ CNT(16)
FF2B 3E01                  LD      A,1                   ;HIGH PULSE(7)
FF2D D3FF                  OUT     (0FFH),A              ;TURN ON(11)
FF2F 19            LOOP2   ADD     HL,DE                   ;DECREMENT F CNT(11)
FF30 DA2FFF                JP      C,LOOP2                 ;GO IF NOT -1(10)
FF33 2A44FF                LD      HL,(FCNT)             ;GET FREQ CNT((16)
FF36 3E02                  LD      A,2                   ;LOW PULSE(7)
FF38 D3FF                  OUT     (0FFH),A              ;TURN OFF(11)
FF3A 19            LOOP3   ADD     HL,DE                   ;DECREMENT F CNT(11)
FF3B DA3AFF                JP      C,LOOP3                 ;GO IF NOT -1(10)
FF3E DD19                  ADD     IX,DE                 ;DECREMENT DURATION(15)
FF40 DA28FF                JP      C,LOOP1               ;GO IF NOT -1(10)
FF43 C9                    RET                         ;RETURN(10)
0002               FCNT    DEFS    2                   ;TEMPORARY STORAGE
                   ; HIGH FREQUENCY TONE GENERATOR
FF46 11FFFF        HIFREQ  LD      DE,-1               ;DECREMENT(10)
FF49 41            LOOP4   LD      B,C                   ;GET FREQ CNT(4)
FF4A 3E01                  LD      A,1                   ;HIGH PULSE(7)
FF4C D3FF                  OUT     (0FFH),A              ;TURN ON(11)
FF4E 10FE          LOOP5   DJNZ    LOOP5                   ;GO IF NOT 0(8/13)
FF50 41                    LD      B,C                   ;GET FREQ CNT(4)
FF51 3E02                  LD      A,2                   ;LOW PULSE(7)
FF53 D3FF                  OUT     (0FFH),A              ;TURN OFF(11)
FF55 10FE          LOOP6   DJNZ    LOOP6                   ;GO IF NOT 0(8/13)
FF57 19                    ADD     HL,DE                 ;DECREMENT DURATION(11)
FF58 DA49FF                JP      C,LOOP4               ;GO IF NOT -1(10)
FF5B C9                    RET                         ;RETURN(10)
0000                       END
00000 TOTAL ERRORS
```

Figure 10-15. Cassette TONE Routine

Two values are used on entry. HL holds the "duration count," while BC holds the "frequency count." The frequency count FD represents the number of iterations through either LOFREQ or HIFREQ to produce a desired frequency. The frequency count FC for any tone less than 300 Hz. can be determined by

FC = (1/F-52.45 microsecs)/23.68 microsecs and for any tone greater than 300 hz. by
FC=(1/F-31 microsecs)/14.66 microsecs

These formulas are found by adding up the instruction times in the two routines (T cycle lengths are shown in parentheses).

The duration count DC is simply a count of the number of cycles for any frequency for a given duration in seconds. (For one second duration, this is F cycles).

```
DC=D*F-1
```

The `-1` represents the adjustment for the `JP C` termination instead of termination on zero.

Typical values for F(frequency), D(duration), FC(frequency count), and DC(duration count) are shown in Figure 10-16.

Frequency (Hz)	Duration (Secs)	Timing Count	Duration Count
100	.25	420.083	24
300	.25	225.261	74
500	.25	134.311	124
700	.25	95.3323	174
900	.25	73.6774	224
1100	.25	59.8971	274
100	.5	420.083	49
300	.5	225.261	149
500	.5	134.311	249
700	.5	95.3323	349
900	.5	73.6774	449
1100	.5	59.8971	549
100	.75	420.083	74
300	.75	225.261	224
500	.75	134.311	374
700	.75	95.3323	524
900	.75	73.6774	674
1100	.75	59.8971	824
100	1	420.083	99
300	1	225.261	299
500	1	134.311	499
700	1	95.3323	699
900	1	73.6774	899
1100	1	59.8971	1099

Figure 10-16. Timing Count Vs. Frequency

Because these counts would be hard to "number crunch" in assembly language, we've provided a BASIC driver to interface to the TONE routine (see Figure 10-17). The driver calculates FC and DC from a given frequency and duration and calls TONE with FF00H,1H=DC, FF02H,3H=FC, and FF04H=0 for a tone <300 hz. or 1 for a tone >=300 hz. It would be relatively easy for you to produce musical note frequencies and durations by adding to this driver.

```
100 'TEST DRIVER
200 D=.25
300 FOR F=20 TO 2000 STEP 10
400 GOSUB 10000
500 NEXT F
600 GOTO 600
10000 'TONE DRIVER. ENTER WITH F=FREQUENCY DESIRED, D=DURATION
10010 'IN SECONDS DESIRED.
10020 IF F<300 THEN FC=(1/F-52.43E-6)/23.68E-6 ELSE
FC=(1/F-31E-6)/14.66E-6
10030 DC=D*F-1
10040 POKE &HFF00,DC-INT(DC/256)*256
10050 POKE &HFF01,INT(DC/256)
10060 POKE &HFF02,FC-INT(FC/256)*256
10070 POKE &HFF03,INT(FC/256)
10080 IF F<300 THEN POKE &HFF04,0 ELSE POKE &HFF04,1
10090 DEFUSR0=&HFF05
10100 X=USR0(0)
10110 RETURN
```

Figure 10-17. BASIC TONE Driver

Chapter Eleven

Disk I/O in Assembly Language

We're going to look at another type of I/O device in this chapter — the floppy disk drive of the TRS-80. We'll first examine some of the physical and electrical characteristics of the disk, look at TRSDOS file structure, and then look at communication with disk data via assembly-language calls to TRSDOS.

Diskette and Disk Characteristics

A diskette is a circular piece of mylar coated with a ferro-magnetic material. As it comes from the manufacturer, it's unmagnetized with **no data of any type** on it. There are no inherent **tracks** or **sectors** permanently embedded in the magnetic medium.

The manufacturer usually **certifies** the diskette. This certification process involves writing and reading back data at high **bit densities** to verify that there aren't any gaps or flaws in the magnetic material that would cause loss of data.

There are two basic diskette formats, the **hard-sectored** and **soft-sectored**. The TRS-80 uses a soft-sectored type of diskette format. The hard sectored diskette has ten sector index holes, while the soft-sectored diskette has one sector index hole, as shown in Figure 11-1.

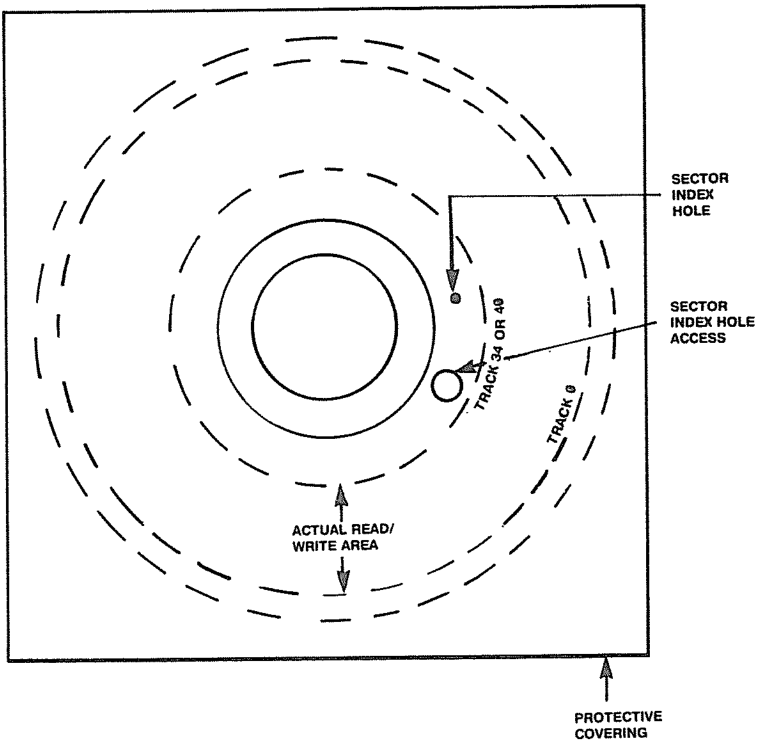


Figure 11-1. Diskette Structure

The purpose of the sector index hole is to act as a reference for the start of **sector 0**, the first sector of the ten sectors per **track** for the Model I or 18 sectors per track for the Model III. Each sector is filled with 256 bytes of data, making a total track's worth of data 2560 bytes (4608 bytes for the Model III). Each diskette is normally divided into 35 tracks for the Model I or 40 tracks for the Model II, numbered 0 through 34 (39). The entire diskette can therefore hold 2560*35 = 89600 bytes of data (184320 bytes for the Model III).

In addition to the data bytes in each sector, there's **header** and **trailer** data in the sector. This data contains the track #, sector number, and checksum for the sector. In addition, there is "filler" data preceding and trailing the ten sectors worth of data, as shown in Figure 11-2.

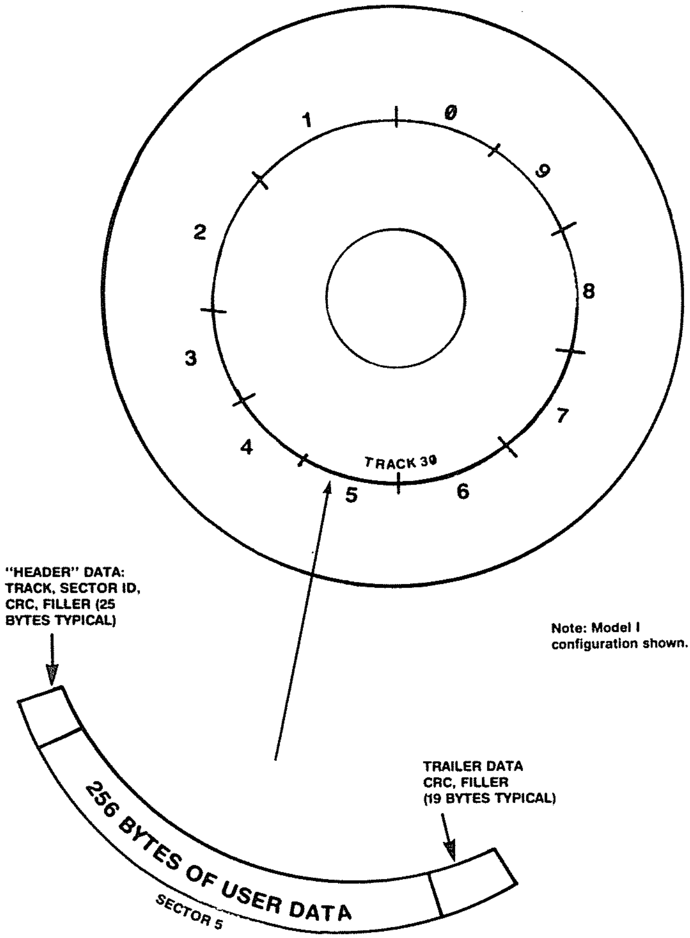


Figure 11-2. Disk Header/Trailer

All of this data is put on the disk by the **formatting program**. The formatting program is a simple program that you use to **format** the header, trailer, and filler data on the diskette in preparation for storing actual data in the data portion of each sector. You might visualize the action of the formatting program as putting a "skeleton" of the sector structure on disk. The formatting program simply stores 256 bytes of dummy data for the 256-byte data area in each sector.

Hints and Kinks 11-1
How is Formatting Done?

You don't really want to format your own disks, do you? All right, here goes . . . .

Formatting simply involves positioning the disk head to each of the 35 (Model I) or 40 (Model III) tracks and then issuing a ''write track'' command to the disk controller chip. The write track essentially tells the controller ''here comes the data.'' The data consists of ten sector segments and filler. The sector segments contain special characters that cause data address ''marks,'' ID address marks, and CRC check bytes to be written. The actual sequence is FE (ID address mark action), track #, sector #, F7 (CRC action), filler, data address mark, dummy user data (256 bytes of E5), F7 (CRC action), and filler, repeated 10 times.

The formatter program writes this data out one byte at a time until the entire track has been written.

Disk Drives

In order to locate a track, each 35- or 40-track disk drive steps or positions a read/write head one discrete increment for each track, as shown in Figure 11-3. There are no other track references that the drive can read to find the proper track; it has to step the head precisely for each increment to a new track.

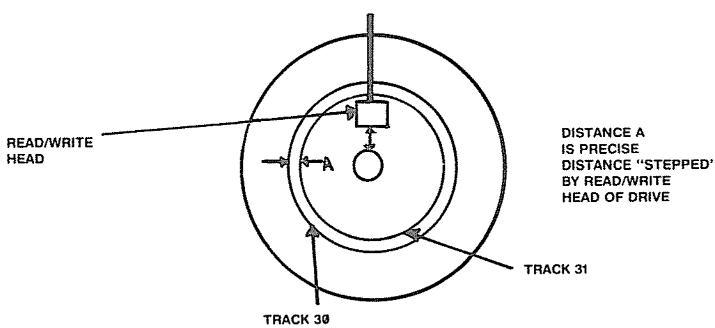


Figure 11-3. Disk Drive Operation

To perform this task, each drive has a stepper motor or other positioning scheme that positions the head precisely. There are three basic movements that each drive can perform: it can **restore** the head to a position over track zero, it can step inward one track; or it can step outward one track. The software controlling the drive usually does a restore operation initially, and then keeps track of where the head is as it steps in and out over the various tracks. At any time, however, it can always easily find track zero again by performing a restore.

The disk drive motor turns at about 300 revolutions per minute (5 revolutions per second). This means that data is passing under the read/write head of the drive at about 2560*5 bytes per second (12,800 bytes per second) for the Model I and about 4608*5 bytes per second (23,040 bytes per second) for the Model III. Since the data is recorded **serially** along the track, this amounts to a string of 102,400 (184,320) bits recorded along each track. (In these figures we're ignoring the header, trailer, and filler data, which actually increases the **data rate** by 20% or so.)

Hints and Kinks 11-2

Disk I/O Read/Write Timing

Considering that the data rate is about 12,800 bytes per second, that means a byte every 78 microseconds for the Model I. Since the data is being written out a byte at a time in a software timing loop, the loop itself must be fairly ''tight'' to ensure that it can get back with the next byte in time. If you miss that 78 microsecond ''window,'' you end up with the dreaded lost data condition. The ball game is lost, at least for that sector. A time of 78 microseconds represents, say, 15 instructions, so some efficiency in coding is called for here, especially if real-time-clock interrupt processing is occurring every 25 milliseconds or so! (RTC processing adds more instructions.) For the Model III, the timing constraints are even ''tighter'' – about a byte every 43 microseconds.

The disk drive is really a very **dumb** device. The signals passing from the disk drive to the expansion interface are shown in Figure 11-4. There aren't very many. There are two lines for WRITE DATA and READ DATA. Since the disk drive reads and writes data serially along the track, data is passed a bit at a time in one direction or the other. There's also a line that passes a STEP command to the drive; associated with the STEP line is a DIRECTION line which determines whether the track step will be in or out.

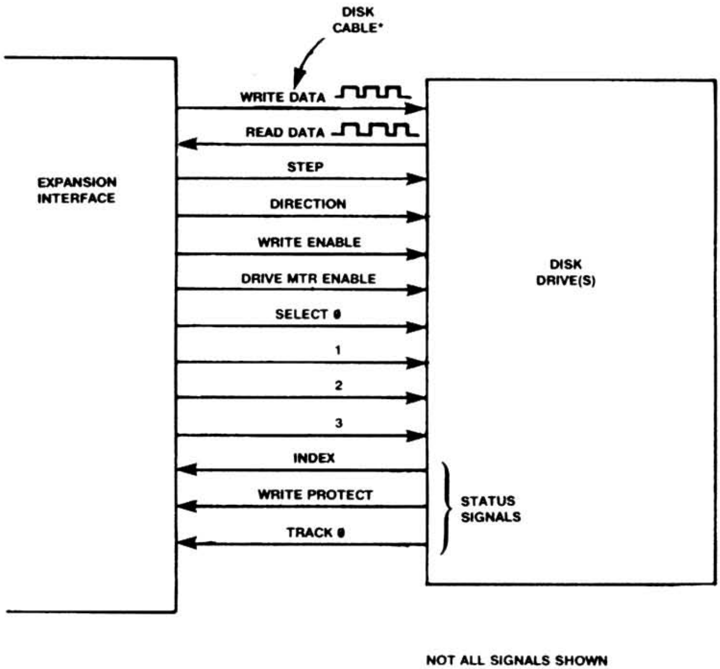


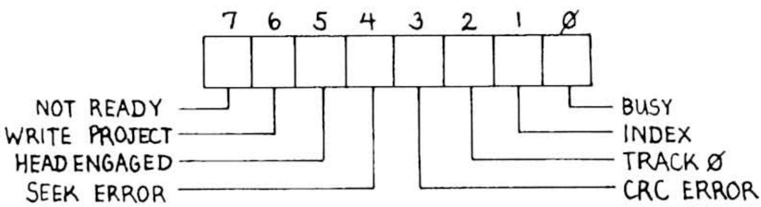
Figure 11-4. Disk Signals

WRITE ENABLE must be in force to allow writing data to the disk. You use DRIVE MTR ENABLE to turn on the disk motor for a read or write and SELECT to select the drive and enable the operation.

Signals coming back include the READ DATA line, and the **status** signals INDEX (true when index hole passes under a sensor), TRACK ZERO (true when the head is over track zero), and WRITE PROTECT (true when the write protect diskette notch is covered).

Hints and Kinks 11-3
Status Signals

Like other I/O devices, the disk and disk controller send back status so that the program knows about disk conditions and about the busy/read state for reads and writes. The status byte for a positioning command such as a restore command on the Model I would be



As a fun project, load the following BASIC program to test ''sector 0'' (index hole) status for a Model I disk drive:

```
100 POKE 14304,1
200 A=PEEK(14316)
300 IF (A AND 2)=2 PRINT "SECTOR 0"
400 PRINT A
500 GOTO 200
```

The Disk Controller

As the disk drive is such a dumb device, intelligence for disk drive operations must be incorporated elsewhere. Much of the intelligence is in the **disk controller chip** in the expansion interface of the TRS-80 Model I or the disk controller logic of the Model III.

The disk controller chip is a small microprocessor in itself. It handles all of the lower-level disk operations such as

- Converting 8-bit bytes into eight serial bits for writes

- Assembling serial data into 8-bit bytes for reads
- Restores (moving the head to track 0)
- Stepping the head in or out
- Seeks (finding a specified track)
- Reading and writing tracks, including formatting data
- Reading the "ID field" of a sector header

In doing all of these operations, the disk controller relieves the software from controlling the timing and other functions as in the cassette drivers.

The expansion interface of the Model I contains the disk controller chip and some additional circuitry for disk drive address decoding. A block diagram of this is shown in Figure 11-5. There are two general sets of addresses associated with the disk. Both are **memory-mapped addresses** that are 16-bit values. (Memory reference instructions are used in place of INs and OUTs.)

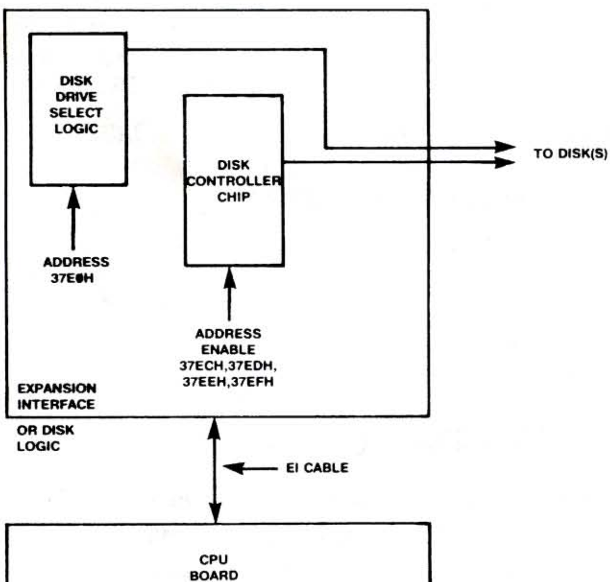


Figure 11-5. Disk Logic

One of the four possible disk drives on a TRS-80 Model I system is "selected" by loading a register with a 1, 2, 4, or 8 and performing a write to memory location 37E0H. This operation essentially turns the drive motor on for about three seconds and **enables** other operations to take place. A typical select of drive 0 would be

```
LD  A,1          ;FOR DRIVE 1
LD  (37E0H),A   ;SELECT DRIVE  OUT ~~F4H,A~~ 244
                                    F4H,A
```

The remaining addresses are 37ECH, 37EDH, 37EEH, and 37EFH.

These addresses are all associated with registers in the disk controller chip. The general actions for each address for a read or write are

ADDRESS	READ ACTION	WRITE ACTION
37ECH	Read status	Write command
37EDH	Read track	Write track
37EEH	Read sector	Write sector
37EFH	Read data	Write data

All communications from TRSDOS and other software are done with the disk controller chip by issuing a series of one-byte commands and by transferring one byte of data between a CPU register and the disk controller chip.

For the Model III, the procedure is very similar, except that the "memory-mapped" addresses have been changed to I/O ports of 0F0H, 0F1H, 0F2H, and 0F3H.

Hints and Kinks 11-4
Some Disk Controller Commands

Commands available to the program for the disk controller include eleven separate one-byte instructions. These include five commands to position the head: Restore puts the head over track 0; Seek finds a given track; Step, Step-In, and Step-Out all move the head either in or out one track.

Read and Write are used to start the process of writing one sector's worth of data. Prior to the read or write, the head must have been positioned over the proper track, and a sector register must have been loaded with the sector address. The read or write is followed by the transfer of 256 bytes in a software loop that looks for a status bit indicating that the controller can accept the next data byte.

Read Address reads the track/sector address (not often used). Read Track reads the entire formatted track; Write Track writes (formats) a track. Force Interrupt causes a software interrupt.

Unfortunately, a large number of commands can be issued to the controller chip with complicated actions and responses — far too many to adequately cover in one chapter. So we'll give you a typical sequence of operations, a flavor of how things are done in **disk driver** programs that make up the lowest level of TRSDOS and other programs that read and write data to the disk. Figure 11-6 shows the disk bootstrap program in Level II ROM. It first tests to see whether a disk drive is connected to the system, restores the disk head to track 0, and then reads in the 256 data bytes of sector 0, track 0. This sector contains a **bootstrap program** (BOOT/SYS) that reads in the remainder of TRSDOS.

```
0696                    ORG     696H
0696 3AEC37             LD      A,(37ECH)       ;GET DISK STATUS
0699 3C                 INC     A
069A FE02               CP      02
069C DA7500             JP      C,75H           ;GO IF NO DISK
069F 3E01               LD      A,1             ;FOR DRIVE 1
06A1 32E137             LD      (37E1H),A       ;SELECT DRIVE 1
06A4 21EC37             LD      HL,37ECH        ;ADDRESS COMMAND
06A7 11EF37             LD      DE,37EFH        ;DATA REGISTER ADDR
06AA 3603               LD      (HL),3          ;RESTORE COMMAND
06AC 010000             LD      BC,0000         ;DELAY 64K COUNTS
06AF CD6000             CALL    0060H
06B2 CB46       LOOP1   BIT     0,(HL)            ;TEST BUSY
06B4 20FC               JR      NZ,LOOP1          ;GO IF STILL BUSY
06B6 AF                 XOR     A               ;ZERO A
06B7 32EE37             LD      (37EEH),A       ;0 TO SECTOR REG
06BA 010042             LD      BC,4200H        ;MEMORY ADDRESS
06BD 3E8C               LD      A,8CH           ;READ COMMAND
06BF 77                 LD      (HL),A          ;READ SECTOR 0
06C0 CB4E       LOOP2   BIT     1,(HL)            ;TEST DRQ
06C2 28FC               JR      Z,LOOP2           ;GO IF NO DATA AVAIL
06C4 1A                 LD      A,(DE)            ;GET NEXT DATA BYTE
06C5 02                 LD      (BC),A            ;TRANSFER DATA
06C6 0C                 INC     C                 ;BUMP BUFFER PNTR
06C7 20F7               JR      NZ,LOOP2          ;GO IF NOT 256 BYTES
06C9 C30042             JP      4200H           ;TRANSFER TO LOADER
0000                    END
00000 TOTAL ERRORS
```

Note: Model I Bootstrap Shown.

LOOP TO READ 256 BYTES OF TRACK 0, SECTOR 0 INTO 4200H AREA

Figure 11-6. Disk Bootstrap

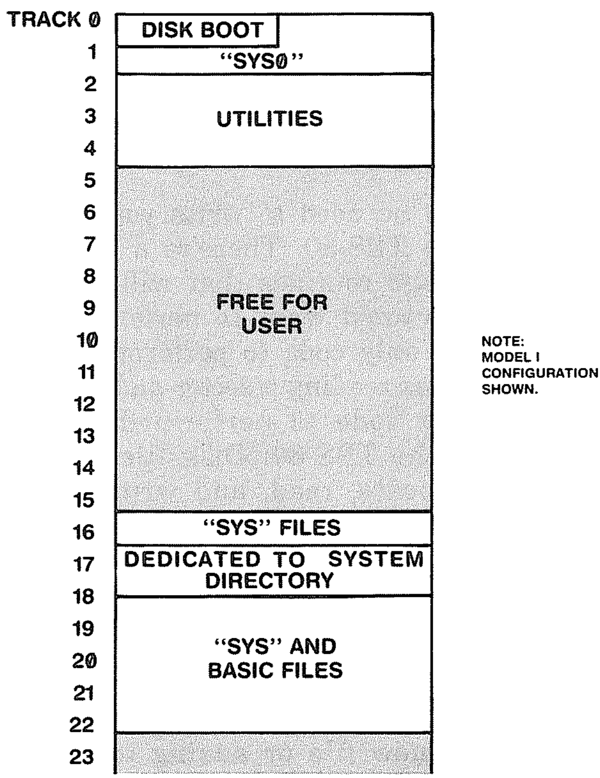
TRSDOS Disk Organization

Actually there's no need to write your own disk driver routines for the TRS-80. There're a number of TRSDOS assembly-language routines that will handle almost all disk operations you'd want to perform. These routines incorporate not only code to perform rudimentary disk operations such as reading a sector and restoring the head to track 0, but code to perform **disk file manage** operations on the TRS-80. Disk file manage refers to operations to locate, read, and write disk files. We'll discuss all of these routines in detail, but first let's look at TRSDOS disk organization so that we may better understand what is involved.

TRSDOS disk files are made up of from 1 to 32 **granules**. A granule is five (Model I) or three (Model III) sectors and is the minimum amount of space that TRSDOS allocates when establishing a new file or adding to an existing file.

TRSDOS uses **dynamic disc allocation** that allocates only enough (or slightly more) disk space to hold the current number of granules in a file. Deleted files cause the disk space used in the file to be released to the **pool** of disk granules (there is a **granule allocation table or** GAT that is essentially a directory of which granules are in use and which are in the pool of unused granules).

As you know from reading your TRSDOS manual, there may be up to 48 separate user files on each diskette. A file is simply a collection of **records** which are sets of any type of data in some organized fashion. In practice, a file may be spread over **non-contiguous** areas of the disk, but you can use the file manage functions of TRSDOS to retrieve all records associated with a file by reference to the **file directory** on disk track 17. A typical disk map for the Model I is shown in Figure 11-7.



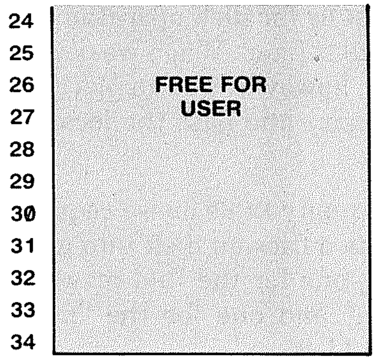


Figure 11-7. Typical Disk Map

A **physical record** on disk is equivalent to a disk sector. At the disk driver level, all disk files are read one sector at a time, bringing in 256 bytes of data. However, within a physical record there may be one or more **logical records**.

A logical record is the actual record used in data storage and retrieval by the program. It may be 1 to 255 bytes in length. Logical records are **blocked** into physical records (sectors) by the TRSDOS file manage routines. For example, if you used 64-byte logical records to hold names and phone numbers, four 64-byte records would be packed into the sector physical record. One of the main tasks of the TRSDOS file manage routines is to access the next logical record from the proper physical record.

TRSDOS works with a **buffer** in memory. A buffer is an area of memory set aside to hold the 256 bytes of a physical record sector. When we're working in Disk BASIC, these buffer areas are preassigned to fixed locations in the TRSDOS area; however, we can use our own areas, when we're using the TRSDOS routines, as we shall see.

Device Control Blocks

Device Control Blocks are "working storage" areas of memory dedicated to variables connected with a particular I/O driver. Level II BASIC uses several DCBs connected with keyboard, video display, and line printer operations; TRSDOS uses additional DCBs for disk operations. In both cases the location of the DCB is fixed. When we utilize the TRSDOS disk file manage calls, however, we can place the DCBs anywhere in memory we desire and pass the location of the DCB as a parameter.

We may have as many DCBs as we require. For example, if we are merging two files on disk into a third file, we could have three DCBs, one for the "old master file," one for the "transaction file," and one for the "new master file," as shown in Figure 11-8.

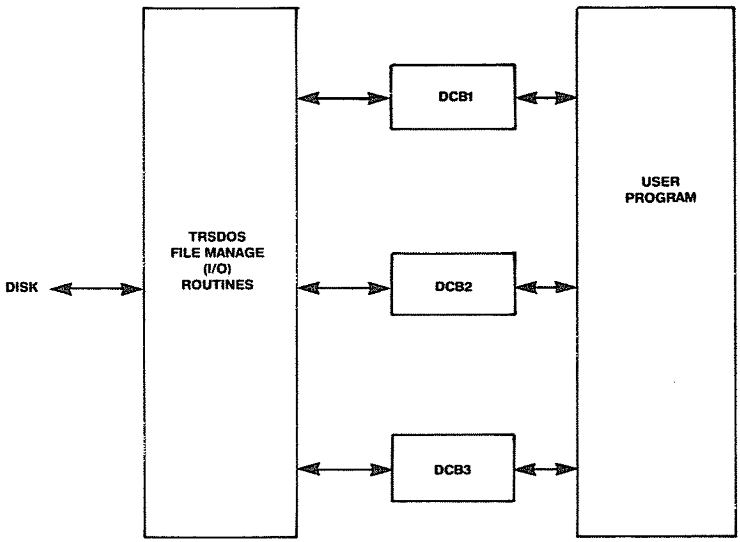


Figure 11-8. DCB Use

Each DCB has two lives. Before the file operations are started (OPENed) and after file operations are terminated (CLOSEd), the 32-byte (Model I) or 30-byte (Model III) DCB contains the file name in standard TRSDOS format, a carriage return, and blanks, as shown in Figure 11-9. During file operations, the TRSDOS file manage routines and the user communicate (pass parameters) by using variables in the DCB as shown.

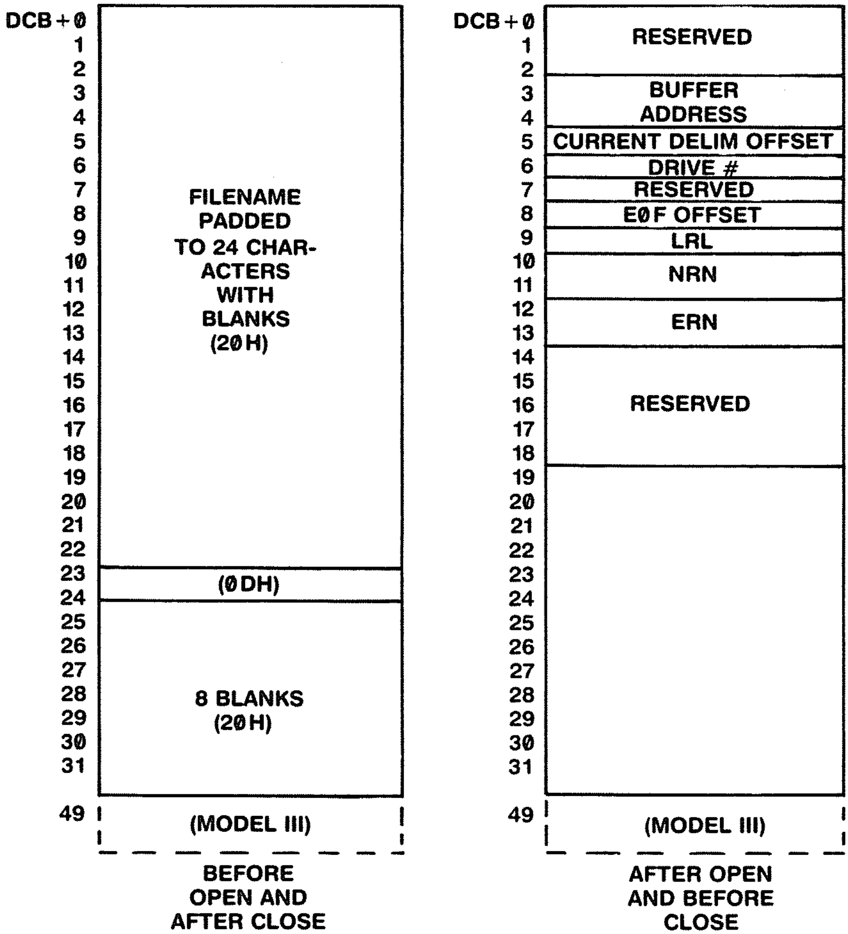


Figure 11-9. DCB Format

During the file operations, the DCB bytes are used to hold variables. The variables are put into the DCB by the TRSDOS routines but may be examined by your assembly-language code. Some of the variables are initialized from user-supplied data, such as the buffer address.

Bytes +3 and 4 hold the buffer address for disk reads and writes. The buffer may be anywhere in memory that's compatible with the configuration; without Disk BASIC, this means anywhere above 6FFFH (Model I) or 6000H (Model III). If Disk BASIC is being used, follow the same rules about assembly-language memory areas as you normally would (see Chapter 4).

Byte +5 holds a buffer displacement (or offset) of 1 to 255 to the end of the current record. In other words, it's an index to the last byte of the current record. If, for example, you were processing the second 64-byte logical record of a sector, this byte would contain 127. During normal processing, you would not employ this variable.

Byte +6 holds the file drive number of the disk drive being used in the operation. Valid numbers are 0 through 3. This constant is stored by TRSDOS from the file name of the DCB.

Byte +8 holds the buffer displacement (offset) of the last delimiter of the last physical record. In other words, when the last sector has been read in, this byte contains an index of 1 to 255 that points to the last byte of the file. Don't use this variable during normal processing.

Byte +9 holds the LRL, or logical record length. This variable may be 1 to 255 and is put in by TRSDOS from user-supplied data. If the LRL is 0, TRSDOS will assume the user is **blocking** his own records and will not find logical records for him (more about that later). This variable is constant for the entire file.

Bytes +10 and +11 hold the NRN or next record number in standard 16-bit format (ls byte followed by ms byte). This variable is set to 000H by TRSDOS after the file has been found initially (OPENed) or created (INITialized). It is incremented by one by TRSDOS after each read for **sequential files**, or you may change it for **random files.**

Bytes +12 and +13 contain the ending record number of the current file in standard 16-bit format. This represents the last logical record number of the file.

Byte +0, +1, +2, +3, +14, +15, +16, +17, and +18 are "reserved," which is a polite way of saying, "Don't tread on me!" They should not be altered by the user.

TRSDOS I/O Calls

Now that we know how the DCB is set up, let's look at the sequence and setup for TRSDOS I/O calls. There is an excellent write-up of the calls in the *TRSDOS/Disk BASIC Reference Manual*, so we won't repeat the formats here. There are eight calls for the Model I and seventeen calls for the Model III. We'll consider the "subset" of eight Model I calls here:

Call	Action
INIT	Initialize a new file
OPEN	"Open" an existing file
POSN	Positions a file to read or write a random record
READ	Reads one logical record from disk or buffer
WRITE	Writes one logical record into disk or buffer
VERF	Verifies a physical record
CLOSE	"Closes" an opened file
KILL	Closes a file and deletes it from directory

In general, all of these calls are made by CALLing a TRSDOS routine in the 44XXH area. In all calls, the DE register holds a pointer to the first byte of the DCB. HL and B or BC may also hold a parameter depending upon the call type. For all calls, after the call is made, a successful action returns with the Z flag set. If the Z flag is not set on return, an error has occurred and the A register contains an error code. Error codes for TRSDOS I/O calls, along with probable causes, are shown in the *TRSDOS/Disk BASIC Reference Manual.*

Reading an Existing I/O File

The normal sequence to read an existing file is this:

1. Put the file name in standard format into the DCB.
2. Make an OPEN call. This causes TRSDOS to search the directory and find the file.
3. If the file is a sequential file, perform a series of READs of the next logical record until the last record (ERN or ending record number) has been processed. If the file is a random file, perform the READs after a POSN call for

each READ. The POSN uses the desired NRN (next record number) in the DCB to find the required logical record.

4. CLOSE the file.

To see how this works in a simple case, enter the BASIC program shown in Figure 11-10. Save the program on disk by performing an ASCII save:

```
SAVE "TEST",A
```

```
100 'THIS IS LINE 1......................................
200 'THIS IS LINE 2......................................
300 'THIS IS LINE 3......................................
400 'THIS IS LINE 4......................................
500 'THIS IS LINE 5......................................
600 'THIS IS LINE 6......................................
```

Figure 11-10. BASIC ASCII Test File

This will write the file in ASCII format so that we can use the program in Figure 11-11 to read it in a record at a time and list it on the screen.

```
8000                      ORG     8000H
3C00           BUFFER     EQU     3C00H
               ; SAMPLE PROGRAM TO READ EXISTING FILE
               ;
               ; FIRST OPEN THE FILE
8000 21003C    READF      LD      HL,BUFFER        ;BUFFER LOCATION IN HL
8003 113B80               LD      DE,DCB1          ;DCB LOCATION
8006 0600                 LD      B,0              ;READ ONE SECTOR
8008 CD2444               CALL    4424H            ;MAKE OPEN CALL
800B 2808                 JR      Z,REA010         ;GO IF OK
800D F680      REA005     OR      80H              ;SETUP FOR ERROR MSG
800F CD0944               CALL    4409H            ;DISPLAY ERROR MESSAGE
8012 CD2D40               CALL    402DH            ;REBOOT
               ; NOW READ AND DISPLAY
8015 113B80    REA010     LD      DE,DCB1            ;DCB LOCATION
8018 CD3644               CALL    4436H              ;READ RECORD
801E 20F0                 JR      NZ,REA005          ;GO IF ERROR
801D 2A3E80               LD      HL,(DCB1+3)        ;GET BUFFER ADDRESS
8020 110001               LD      DE,256             ;INCREMENT
8023 19                   ADD     HL,DE              ;BUMP TO NEXT SCREEN SECTION
8024 223E80               LD      (DCB1+3),HL        ;STORE FOR NEXT READ
8027 DD213B80             LD      IX,DCB1            ;DCB ADDRESS
802B DD7E0A               LD      A,(IX+10)          ;GET NRN
802E DDBE0C               CP      (IX+12)            ;COMPARE TO LRN
8031 20E2                 JR      NZ,REA010          ;GO IF ERROR
8033 113B80               LD      DE,DCB1          ;DCB LOCATION
8036 CD2844               CALL    4428H            ;CLOSE FILE
8039 18FE      REA020     JR      REA020           ;JUMP HERE ON END
803B 54        DCB1       DEFM    'TEST
     45 53 54 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20
8052 20                   DEFM    '          '
     20 20 20 20 20 20 20
0000                      END
00000 TOTAL ERRORS
```

Figure 11-11. Read Test File Program

The program in Figure 11-11 is a simple read of an existing disk file. We did some things in it that are a little dangerous (like incrementing 256 bytes each time through the loop for the screen address and modifying the buffer address in the DCB), so try it only with the TEST file from Figure 11-10.

The DCB for the read is DCB1. Before we OPEN the file, the DCB contains TEST padded out to 23 characters with blanks, a carriage return, and eight terminating blanks. This corresponds to the DCB format required by the TRSDOS calls. We OPENed the TEST file by CALLing 4424H with the buffer address in HL, the DCB address in DE, and the logical record length in B. The logical record length in this case is 0, signifying that a logical record corresponds to a physical record, or sector.

If an OPEN error had occurred, Z would not be set on return, and we would have gone to the special "display TRSDOS error message" routine at 4409H and then rebooted.

Hints and Kinks 11-5

Error Code Routine

The error code routine is another TRSDOS call that converts the error code returned in the A register to a description message. The message is then displayed. It makes sense to use it, rather than printing out a nebulous ERROR 1786, SUB A-8 DURING HIGHEST HIGH TIDE.

After a successful open, the DCB contains variable data placed in it by the TRSDOS OPEN routine. The DCB appears as shown in Figure 11-12. Note that at this time TRSDOS knows the length of the TEST file and puts a 0002H in DCB+12,+13 (ERN or end record number). It has also initialized the NRN (next record number) to 0000H in preparation for a READ or WRITE.

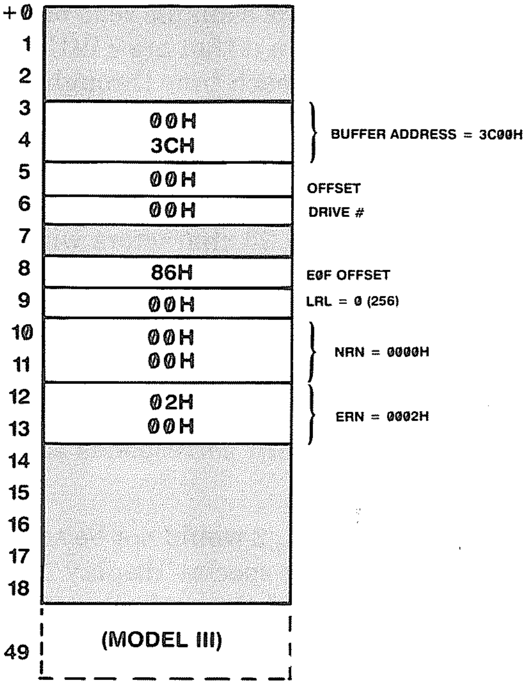


Figure 11-12. DCB After OPEN

The next set of code makes a call to READ a record. If the logical record length were not 0 (256 bytes), we would specify a "user record" area in HL. The READ call would then transfer the next logical record from the buffer into the user record area. This might involve a new disk read of the next sector, or would simply involve transferring the next logical record from the buffer.

Since we're working with logical records equal to physical records (sectors), however, we did not pass anything in HL.

After a successful READ, the DCB appears as shown in Figure 11-13. The NRN has been incremented to 1, and the data from the sector appears on the screen (buffer). The buffer location is now picked up from DCB + 3, + 4 and incremented by 256 to point to the next screen location. (Don't try this for more than four sectors!)

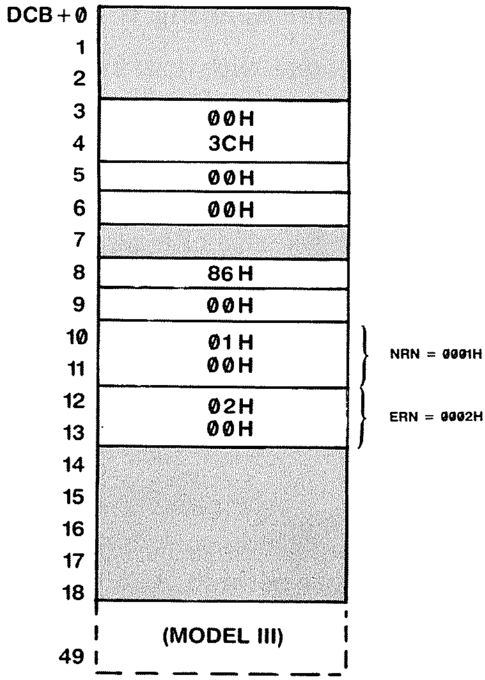


Figure 11-13. DCB After READ

Now we come to an important portion of code. The current record number is picked up from DCB+10 and compared to the ending record number in DCB+12. If they are not equal, another READ is done. If they are equal, you perform a CLOSE file by CALLing TRSDOS CLOSE at 4428H with DE containing the DCB address. For the TEST file, we make two passes through the read portion of the code and then CLOSE the file and loop at REA020.

This is in essence how a read of any existing sequential file can be accomplished. Many Radio Shack disk files have logical record lengths of 256 bytes (look at any DIR display), so the procedure for processing any existing BASIC files would be very similar to the above code.

Creating a New File and Reading It Back

Now that we've gotten our feet wet with disk I/O, let's try a plunge into deep water with a write of a new file and a following read. We'll use logical record lengths other than 256. The program shown in Figure 11-14 takes the contents of the screen and opens a new file called SCREEN. SCREEN has logical record length equal to the size of a screen line (64 bytes). There'll be 16 logical records in 4 physical records for the SCREEN file. After the file is created it can be read back and displayed by the second part of the program.

The program is divided into three parts — initializing a new file and writing it, clearing the screen, and reading the file back in.

Initializing

The first part calls INIT to create a new file. A 256-byte buffer is specified in HL, and the usual DCB is specified in DE. The DCB in this case has the name "SCREEN". **A logical record length of 64 is specified in B.** The code at WRT005 is an error routine to display any disk error on the screen.

After a successful INIT, the character string in the DCB is replaced with the date shown in Figure 11-15. Note that the NRC (next record number) is initialized to 0000H for the INIT, just as it was for the OPEN.

```
8000          00100                ORG      8000H
              00110 ; SAMPLE PROGRAM TO WRITE SCREEN TO DISC AND READ IT BACK
              00120 ; WRITE SCREEN
8000 217280   00130 WRTREA   LD       HL,BUFFER       ;BUFFER LOCATION
8003 117281   00140          LD       DE,WDCB         ;DCB LOCATION
8006 0640     00150          LD       B,64            ;64 BYTES PER LOG RECORD
8008 CD2044   00160          CALL     4420H           ;CREATE A NEW FILE
800B 2808     00170          JR       Z,WRT010        ;GO IF OK
800D F680     00180 WRT005   OR       80H             ;SETUP FOR ERROR MSG
800F CD0944   00190          CALL     4409H           ;WRITE ERROR MSG
8012 C32D40   00200          JP       402DH           ;REBOOT
8015 21C03B   00210 WRT010   LD       HL,3C00H-64     ;SCREEN START-256
8018 E5       00220          PUSH     HL              ;SAVE ADDRESS
8019 E1       00230 WRT020   POP      HL                ;GET LINE #
801A 114000   00240          LD       DE,64             ;INCREMENT
801D 19       00250          ADD      HL,DE             ;POINT TO NEXT LINE
801E 7C       00260          LD       A,H               ;GET MS BYTE
801F FE40     00270          CP       40H               ;TEST FOR LAST LINE
8021 280B     00280          JR       Z,WRT030          ;GO IF DONE
8023 E5       00290          PUSH     HL                ;SAVE UREC ADDRESS
8024 117281   00300          LD       DE,WDCB           ;WRITE DCB LOCATION
8027 CD3944   00310          CALL     4439H             ;WRITE UREC
802A 20E1     00320          JR       NZ,WRT005         ;GO IF ERROR
802C 18EB     00330          JR       WRT020            ;CONTINUE
802E 117281   00340 WRT030   LD       DE,WDCB         ;DCB LOCATION
8031 CD2844   00350          CALL     4428H           ;CLOSE FILE
8034 20D7     00360          JR       NZ,WRT005       ;GO IF ERROR
              00370 ; CLEAR SCREEN
8036 21003C   00380          LD       HL,3C00H        ;SCREEN START
8039 3E20     00390 WRT040   LD       A,' '             ;BLANK
803B 77       00400          LD       (HL),A            ;STORE BLANK
803C 23       00410          INC      HL                ;BUMP PNTR
803D 7C       00420          LD       A,H               ;GET MS BYTE
803E FE40     00430          CP       40H               ;TEST FOR END
8040 20F7     00440          JR       NZ,WRT040         ;GO IF NOT END
              00450 ; NOW READ BACK FILE
8042 217280   00460          LD       HL,BUFFER       ;BUFFER LOCATION
8045 117281   00470          LD       DE,WDCB         ;DCB LOCATION
8048 0640     00480          LD       B,64            ;64 BYTES PER LOG RECORD
804A CD2444   00490          CALL     4424H           ;OPEN SCREEN FILE
804D 20BE     00500          JR       NZ,WRT005       ;GO IF ERROR
804F 21C03B   00510 WRT050   LD       HL,3C00H-64     ;SCREEN START-256
8052 E5       00520          PUSH     HL              ;SAVE ADDRESS
8053 E1       00530 WRT060   POP      HL                ;GET LINE #
8054 114000   00540          LD       DE,64             ;INCREMENT
8057 19       00550          ADD      HL,DE             ;POINT TO NEXT LINE
8058 7C       00560          LD       A,H               ;GET MS BYTE
8059 FE40     00570          CP       40H               ;TEST FOR LAST LINE
805B 280B     00580          JR       Z,WRT070          ;GO IF DONE
805D E5       00590          PUSH     HL                ;SAVE UREC ADDRESS
805E 117281   00600          LD       DE,WDCB           ;DCB LOCATION
8061 CD3644   00610          CALL     4436H             ;READ UREC
8064 20A7     00620          JR       NZ,WRT005         ;GO IF ERROR
8066 18EB     00630          JR       WRT060            ;CONTINUE
8068 117281   00640 WRT070   LD       DE,WDCB         ;DCB ADDRESS
806B CD2844   00650          CALL     4428H           ;CLOSE FILE
806E 209D     00660          JR       NZ,WRT005       ;GO IF ERROR
8070 18FE     00670 WRT080   JR       WRT080          ;LOOP HERE
0100          00680 BUFFER   DEFS     256
8172 53       00690 WDCB     DEFM     'SCREEN                 '
     43 52 45 45 4E 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20
8189 0D       00700          DEFB     0DH
818A 20       00710          DEFM     '       '
     20 20 20 20 20 20 20
0000          00720          END
00000 TOTAL ERRORS
```

Figure 11-14. SCREEN Program

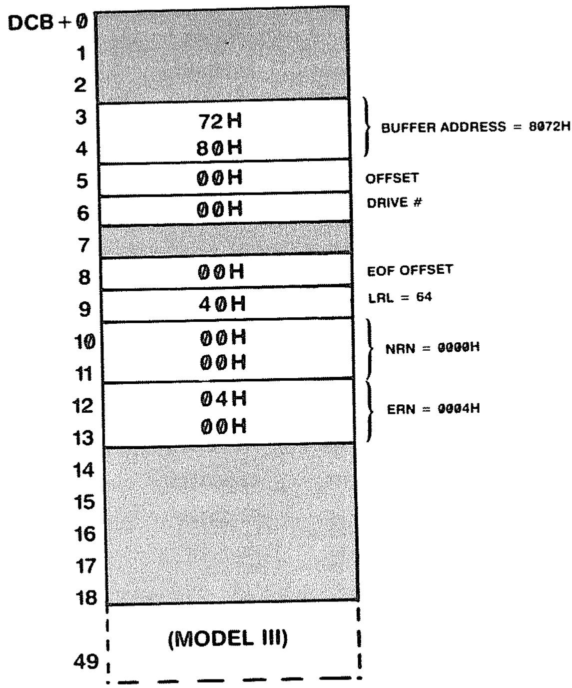


Figure 11-15. DCB After INIT

Writing

Now a series of disk writes is done. Each write specifies a UREC (user record area) in HL. This would normally be RAM memory, but in this case we're using the video screen as a user buffer. The user buffer must be the same length as the logical record length or greater. In fact each new call to WRITE uses the next screen line as the UREC. Each write causes 64 bytes to be moved from a screen line into the buffer area. After each fourth move, a disk write of the physical record (sector) is automatically performed. Typical DCB contents are shown in Figure 11-16.

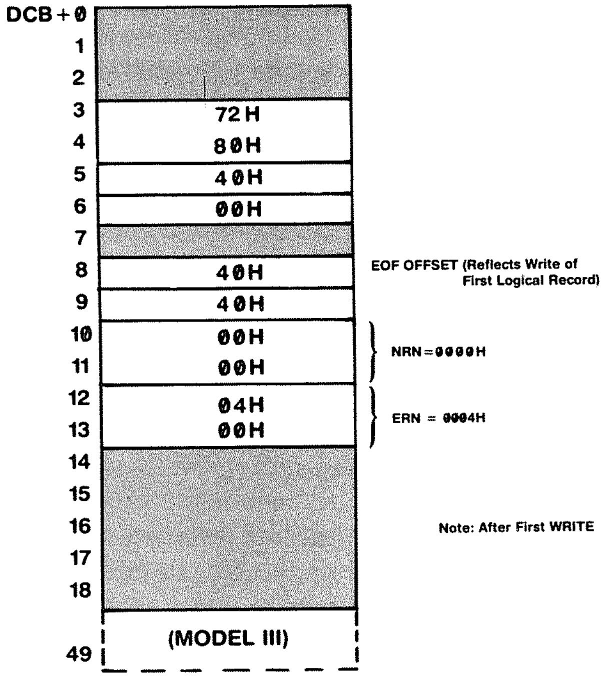


Figure 11-16. DCB After WRITE

Clearing

After 16 writes of the 16 screen lines, a call is made to CLOSE to close the SCREEN file. The CLOSE is especially important on a write as there may be logical records in the buffer that have not yet been written to disk. The CLOSE in this case **flushes** the buffer.

The screen is cleared with the next bit of code.

Now an OPEN is done with the same DCB used as in the INIT and WRITE. Note that the CLOSE **restored** the character data in the DCB, and that we can perform the OPEN without having to reinitialize the DCB with the file name. The OPEN specifies a logical record length of 64. In fact, we must know beforehand what the LRL will be in reading back a file. There is no mechanism for obtaining this from the TRSDOS calls.

Reading The File Back

After a successful OPEN, the NRN (next record number) in the DCB is set to 0000H in preparation for the READ (or WRITE — the TRSDOS routines don't know which will follow). A series of READs is then done in much the same fashion as the WRITEs. Each read specifies a UREC (user record area) in HL which is a video memory line.

After 16 reads, a CLOSE is done to terminate the read action. Note that in this case we knew how long the file was and simply read in 16 logical records. This is not very good programming practice with files; we really should have worked with the ERN (ending record number) and NRN (next record number) in conjunction with a discrete count of records in case there had been some programming or logic error in creating a file that had fewer records than expected. We hope you'll forgive us in this example!

The program will cause the following actions over several seconds: The screen contents (a DEBUG display is good) are written out to disk. The screen is cleared and filled with the same data from the SCREEN file. There are four distinct reads as each sector is read in, with four blocks of four lines being output rapidly to the screen after each read.

Killing a File

The KILL call is very similar to the CLOSE, except that it deletes the file name from the directory and releases the disk space used for the file to the common pool of granules. The action of the KILL can be seen by substituting a CALL 442CH in place of the CALL 4428H near WRT070 in Figure 11-14. Do a DIR command in TRSDOS after a CLOSE and after a KILL, and you will observe the KILL action.

We did not **verify** the physical records as we were writing them out in Figure 11-14. You have a choice of writing out each physical record without reading it back in for comparison (a "normal" WRITE) or writing out a record and reading it back in for comparison after each sector write. In my opinion, you should always verify. I'm assuming that all data being written out to disk is important to you. Although the VERIFY takes slightly longer because a second disk operation must be performed to read in the sector for comparison, it's just good programming practice to double-check.

Change the CALL 4439H after WRT020 to CALL 443CH to perform the VERIFY write. You might want to compare the time for both a WRITE sequence and a VERIFY sequence.

```
                    Hints and Kinks 11-6
              How Many Disk Errors Will There Be?

   (What is Truth? What is Beauty?) The floppy
   disk is an extremely reliable device for an
   electro-mechanical peripheral. If you choose
   not to VERIFY your writes, you'll probably not
   have an error in a diskette full of data.
   However, the overhead really is not that great
   when you consider the high data rates of the
   disk as compared to cassette operation . . . .
```

Using POSN

We created a sequential file in the code of Figure 11-14 by the process of writing a series of logical records in sequence. However, it's just as easy to work with random records.

In the case of a sequential file, the TRSDOS READ and WRITE routines keep pointers to the next logical record in the buffer in the DCB, along with incrementing the NRN (next record number). If we are to work with random records of a file, we must aid TRSDOS in locating the physical records by using the POSN call.

The POSN call is used to specify a logical record number in the BC register. TRSDOS then finds the proper logical record by either resetting the DCB pointers (if the logical record is in the current buffer) or reads in the sector containing the logical record and positions the pointers. This intermediate operation is necessary because there is a good chance that the random logical record is **not** in the current buffer.

Hints and Kinks 11-7

Adding to a File

The POSN command is used to find the last record of an existing file so that additional records may be appended. To do this, setup the BC register pair with a record number corresponding to ERN (ending record number) + 1. The POSN will automatically find the ''end of file.'' You can then do normal WRITEs if you are adding to a sequential file.

Figure 11-17 shows the use of POSN in reading the first and sixteenth records back from the SCREEN file. The code here should be used **after** the SCREEN file has been created by the program shown in Figure 11-14. Two READS are done after an OPEN. Each READ is preceded by a POSN call. The first POSN specifies logical record number 7, while the second call specifies logical record number 15.

There are many subtleties involved in working with disk files, but possibly this brief introduction has been helpful in getting you started. Try experimenting with using the disk calls as described in the TRSDOS manual. The TRSDOS I/O calls may be used to create some powerful assembly-language programs that use disk a lot more efficiently than BASIC code.

```
8000                       ORG     8000H
                   ; PROGRAM TO READ SCREEN FILE AND DEMONSTRATE POSN CALL
                   ; CLEAR SCREEN
8000 21003C                LD      HL,3C00H        ;SCREEN START
8003 3E20          WRT040  LD      A,' '            ;BLANK
8005 77                    LD      (HL),A           ;STORE BLANK
8006 23                    INC     HL               ;BUMP PNTR
8007 7C                    LD      A,H              ;GET MS BYTE
8008 FE40                  CP      40H              ;TEST FOR END
800A 20F7                  JR      NZ,WRT040        ;GO IF NOT END
                   ; READ BACK LOGICAL RECORDS 7 AND 15
800C 215D80                LD      HL,BUFFER       ;BUFFER LOCATION
800F 115D81                LD      DE,WDCB         ;DCB LOCATION
8012 0640                  LD      B,64            ;64 BYTES PER LOG RECORD
8014 CD2444                CALL    4424H           ;OPEN SCREEN FILE
8017 2808                  JR      Z,WRT050        ;GO IF NO ERROR
8019 F680          WRT045  OR      80H             ;SETUP FOR ERROR MSG
801B CD0944                CALL    4409H           ;WRITE ERROR MSG
801E C32D40                JP      402DH           ;REBOOT
8021 215D80        WRT050  LD      HL,BUFFER       ;BUFFER ADDRESS
8024 115D81                LD      DE,WDCB         ;DCB ADDRESS
8027 010700                LD      BC,7            ;LOGICAL RECORD 7
802A CD4244                CALL    4442H           ;POSN CALL
802D 20EA                  JR      NZ,WRT045       ;GO IF ERROR
802F 21003C                LD      HL,3C00H        ;UREC LOCATION
8032 115D81                LD      DE,WDCB         ;DCB LOCATION
8035 CD3644                CALL    4436H           ;READ UREC
8038 20DF                  JR      NZ,WRT045       ;GO IF ERROR
803A 215D80                LD      HL,BUFFER       ;BUFFER ADDRESS
803D 115D81                LD      DE,WDCB         ;DCB ADDRESS
8040 010F00                LD      BC,15           ;LOGICAL RECORD 15
8043 CD4244                CALL    4442H           ;POSN CALL
8046 20D1                  JR      NZ,WRT045       ;GO IF ERROR
8048 21C03F                LD      HL,3FC0H        ;LAST SCREEN LINE
804B 115D81                LD      DE,WDCB         ;DCB ADDRESS
804E CD3644                CALL    4436H           ;READ RECORD
8051 20C6                  JR      NZ,WRT045       ;GO IF ERROR
8053 115D81        WRT070  LD      DE,WDCB         ;DCB ADDRESS
8056 CD2844                CALL    4428H           ;CLOSE FILE
8059 20BE                  JR      NZ,WRT045       ;GO IF ERROR
805B 18FE          WRT080  JR      WRT080          ;LOOP HERE
0100               BUFFER  DEFS    256
815D 53            WDCB    DEFM    'SCREEN                '
     43 52 45 45 4E 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20
8174 0D                    DEFB    0DH
8175 20                    DEFM    '        '
     20 20 20 20 20 20 20
0000                       END
00000 TOTAL ERRORS
```

Figure 11-17. Use of POSN

SECTION III
Larger Assembly-Language Projects

Chapter Twelve
Assembly-Language Design, Coding, and Debugging

In this chapter, we'll follow a typical assembly-language programming project from beginning to end. It'll be a "medium-sized" project, one that might well take a professional programmer a month or more to complete in all phases. We'll describe the trials and tribulations of that programmer and illustrate the steps required in any large assembly-language job. (The punch line of this chapter: "For you see, my friends, that professional programmer was ME!")

This is a story about Cal Coder, a programmer/analyst at GIGO Software, Inc. GIGO is one of the smaller companies producing software packages for the Radio Shack TRS-80 microcomputers.

The Inception Phase

Cal had just walked into his office at GIGO when his phone rang. "Cal, this is Paul . . . . Can you drop in for a second? I think we might have something you'll be interested in."

Having served in the Sea Scouts, Cal knew a direct order couched in polite terms. He started toward his boss's office.

After the usual small talk, Paul explained, "The guys in Marketing have come up with this idea about a new program for the TRS-80 — the ads start tomorrow, and we've already sold a hundred copies. Can you get right on it? Here're the notes on it," he said, handing Cal a matchbook cover with an ad for "Earn Big Money in Your Spare Time Programming" on one side and several scribbled notes on the other.

Looking over the notes, Cal saw that there was a need for a Morse Code Generator program that would act as a Morse code instructor or playback prerecorded Morse code messages. Marketing had done some analysis in Butte, Montana that indicated potential sales of hundreds of copies. It was up to Cal to come up with a specification to further define the project.

Research

"Do you know anything about Morse code?" Cal asked Ted, his officemate and fellow-programmer.

"Is it anything like a Hamming code?" Ted replied, puzzled.

> **Hints and Kinks 12-1**
> **Hamming Code**
>
> Ted was making a pun at Cal's expense. A Hamming code is a special code frequently used in telemetry, but not often used in computer processing. Missing bits in data can be regenerated by analyzing the remaining bits. Obviously, the overall data transmission rate is reduced by inclusion of data bits for reconstruction.
>
> Few enough bits are ''dropped'' in computers to make a code such as this unnecessary in normal applications. Parity bits or other check bits do provide some verification of data, which is usually sufficient.

"Well, hams use it, but I don't think so," Cal replied. "I guess I'll have to do some research on it before I write that spec for the Morse Code Generator I'm working on."

"Why change your approach now?" Ted asked with a leer. Cal threw a carton of C90 cassettes at him.

Later, Cal had dug up most of what he needed to know about Morse code. It was basically a series of long and short pulses representing the letters of the alphabet, digits, punctuation marks, and some special characters. The number of pulses varied. A frequently used letter like "e" consisted of one short pulse, called a "dot" or "dit." A "t" consisted of one long pulse called a "dash" or "dah." Less frequently used letters, digits, and punctuation marks consisted of longer combinations of dots and dashes. A "p", for example, was represented by "dot dash dash dot," while a "5" was "dot dot dot dot dot."

Cal also located the specifications on the standard lengths for dots and dashes and on the spacing. A dot was one unit long, while a dash was three units long. The space between dots or dashes was one unit long, the space between characters was three units, and the space between words was five units long.

Cal verified his findings with some amateur radio friends and got more data. He supplemented this research with some articles in *BYTE*, *80-Microcomputing*, and other computer magazines, and after several days had done enough research to feel he knew more than enough to write the spec.

Hints and Kinks 12-2
Programming Periodicals

Here's a list of some current publications which would have general programming information:

73 Magazine	R
80 Microcomputing	T
ACM Computing Surveys	P
BYTE	H
Communications of the ACM	P
Computer Design	P
Computer Magazine (IEEE)	P
Creative Computing	H
Datamation	P
Dr. Dobb's Journal	H
Eighty US Journal	T
Ham Radio	R
Interface Age	H
Kilobaud Microcomputing	H
Personal Computing	H
Popular Electronics	E
QST	R
Radio Electronics	E

E=Electronics oriented, some computer topics
H=Hobbyist computer magazine
P=Professional magazine
R=Amateur radio, some computer topics
T=TRS-80 topics exclusively

The Preliminary Specification

The spec he produced is shown in Figure 12-1. It's a preliminary operational specification that does not show anything about the actual implementation of the program. As a matter of fact, in looking over the spec, we might ask ourselves whether it is possible to produce such a program. Certainly we know the TRS-80 can produce messages on the screen, can generate audio tones out of the cassette output, and read character strings from the keyboard. But can it generate Morse code characters at 60 words per minute? Is that too fast for even assembly language?

SPECIFICATION: GIGO SOFTWARE PRODUCTS MORSE CODE GENERATOR PROGRAM, MORSE

General Description

This assembly-language software package is a TRS-80 Model I program that will run in systems with 16K of RAM or greater. It will generate international Morse code through the cassette output. Code generated will be either random code characters for practice purposes or a user-defined string of Morse code characters. Speeds of operation are user-defined and may be from 3 to 60 words per minute.

Loading Procedure

The MORSE package is loaded from disk by entering the command, `MORSE`, while in TRSDOS command mode. The MORSE program will be loaded and execution will start immediately.

The MORSE package is loaded from cassette by entering the BASIC command mode. After the > prompt, the following sequence is entered to load and execute the MORSE program:

```
>CMD"T"   (Turn off real-time-clock in Disk BASIC)
>SYSTEM   (Enter Monitor mode)
*? MORSE  (Load cassette tape file MORSE)
*?/       (Start execution after successful load)
```

Operating Instructions

After loading, MORSE will clear the screen and position the cursor to the "home" position at the upper left corner of the screen. It will also establish a communication area on the bottom three lines of the screen as shown in Figure 1.

Pressing `CLEAR` at any time will "reset" MORSE and cause the following line to be displayed:

```
                 MORSE

CHAR=SEND CHARACTER      SHIFT Ø-9=SEND MSG N
SHIFT R=SEND RANDOM
SHIFT D=DEFINE MSG       SHIFT S=DEFINE SPEED
SHIFT P,N=PRINT OR NO
```

The above information is also displayed after loading.

Figure 12-1. Specification for MORSE

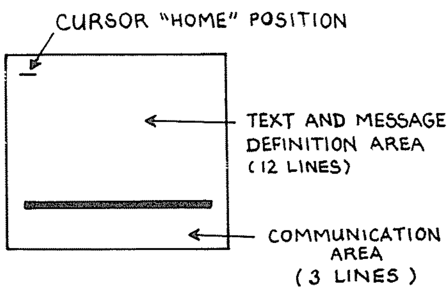


Figure 1. MORSE Display Characteristics

Normal operation: MORSE normally operates in the "send message" mode, where a single key stroke will send a character or one of 10 messages. Pressing a 0 through 9 key with a SHIFT will send a predefined message 0 through 9, while pressing the SHIFT R keys will send a string of random characters. If the message 0 through 9 has not been defined, no action will occur. Transmission of message 0 through 9 will continue at the current speed of operation until the end of the message is reached. At the end of the message, MORSE will await the next command. If a (R)andom string is being sent, transmission will continue indefinitely until the CLEAR key is pressed.

At the same time a character, message, or random characters are being sent, the message will be displayed on the screen and optionally printed on the system line printer (see "Line Printer Operation" below).

Define Message Mode: The define message mode is entered by entering SHIFT D. After the SHIFT D has been pressed, MORSE will respond by the line

```
DEFINE MESSAGE MODE.ENTER MESSAGE
# 0-9:
```

The user must then enter a valid message number 0 through 9. If a valid message number is not entered, the line

```
INVALID MESSAGE NUMBER. MUST BE 0 - 9
```

is displayed, and the user must reenter the number. The user may simply press CLEAR to exit the Define Message Number mode.

Once the message number has been entered, MORSE displays the line

```
ENTER MESSAGE. TERMINATE BY ENTER
```

The user can now enter the Morse code message to be stored as message number N. Valid characters for the message are the alphabetic characters A through Z, numeric digits 0 through 9, period, comma, question mark, slash (/), and several special characters. The special characters are a dash, for "break" (-....-); space, for word space; semicolon for "error" (........); and equals for "end message" (.-.-.). At the end of the message, the user presses the ENTER key. Invalid characters are ignored and are neither entered as part of the message, displayed, nor printed.

The size of the message is limited to 256 bytes per message. MORSE checks the amount of memory used and will generate the error message

```
MORE THAN 256 CHARACTERS. 256 ACCEPTED
```

if there is not enough space to store the current message. Only the first 256 characters of the current message will be stored if this condition occurs!

Deletion of Messages: Messages may be deleted by entering the Define Message mode and pressing the ENTER key for message text. This action releases any previously allocated memory area to the system message area.

Define Speed Mode: the Define Speed mode is entered by entering the keys SHIFT S during normal operation. MORSE responds with

```
SET SPEED MODE. ENTER SPEED 3 TO 60 WPM:
```

The user must now enter an appropriate speed for message transmission, followed by an ENTER. If a valid speed is entered, the MORSE speed is set to that value. If an invalid speed is entered, MORSE will display the message

```
INVALID SPEED. MUST BE 3 TO 60
```

The user must then reenter a correct speed.

The "default" speed of MORSE is 3 words per minute.

Line Printer Operation: Messages may be optionally printed on the system line printer by pressing the keys SHIFT P during normal operation. Pressing the keys SHIFT N disables line printer operation. The system line printer must be capable of responding at the desired speeds. A speed of 60 words per minute, for example, is about 300 characters per minute or 5 characters per second. If the printer has a long "carriage return" time without buffering data, some characters may be lost by certain types of slow printers.

Random Character Generation: During this mode, a sequence of pseudo-random (repeatable) characters are generated. Comparison of code practice copy and the sequence may be performed by examination of line printer or display output.

Audio Output: Audio output from MORSE is through the cassette "AUX" output which would normally connect to the system cassette recorder. Audio output may be recorded directly on blank cassette tape and replayed, or the AUX output may be connected to an external audio amplifier.

Amateur Radio Output: Amateur radio operators may use the audio output to key transmitters through appropriate circuitry of their own design. GIGO SOFTWARE, INC. can assume no liability in interfacing to such an application.

In producing the specification, Cal drew upon his experience with similar types of programs. He did some preliminary computations that verified speeds of 60 words per minute could be obtained with no problems. One of the formulas he found gave the words per minute speed as —

WPM = dots per second *2.4

Working from this, Cal deduced that the fastest **dot time** would be 50 milliseconds, or 50,000 microseconds, or about 10,000 instruction times. He concluded that this was slow for assembly language. However, this was a crucial stage. There were many other problems that may not be visible at this point, as we shall see.

Hints and Kinks 12-3
Instruction Times

Instruction times in the TRS-80 range from 4 T cycles to 23 T cycles. A T cycle (T state) is 1/4 of a clock cycle and is used in Zilog literature to define instruction speeds. The clock rate of the TRS-80 is about 1.77 megahertz, making a T cycle about .564 microseconds and instruction times about 2.255 microseconds to 51.9 microseconds (!). The nominal instruction time is about 9 T cycles or about 5 microseconds, since many instructions take 4,7,8,9, and 10 T cycles.

Program Design

Cal's next task was to produce a complete **implementation specification**. This would be a specification that would discuss technical considerations of implementing the program. This step is often ignored for short programs and sometimes ignored for major programs involving many man years of work, much to the dismay of some companies.

The implementation spec includes flow charts of all program code and may also include a text description of portions of the program. It may also define and specify **system tables**.

Cal was too experienced a programmer to simply start coding the program. He knew that even for small jobs a set of flow charts uncovered many **logic errors** and helped him visualize the **structure** of the program.

As for program structure, Cal usually used a combination of "top-down" and "bottom-up" design. The "top-down" approach dictated the program be designed from major functions downward, finally culminating in the lowest level modules or subroutines. The "bottom-up" approach used the opposite tack: the program was first coded at the lowest level of routines, and then progressively more complicated routines were designed.

Cal already had a preliminary spec that represented the "top-down" operation of the program. His job was now to implement the spec by defining program functional **modules** that would perform the logical functions of the program. A module is a collection of code performing a well-defined specific function with a set of inputs to produce a well-defined set of outputs. He knew that parts of the design would utilize old code, perhaps intact, such as a DELAY subroutine to delay a specified time. He also knew that other parts would be new code that he would have to write from scratch.

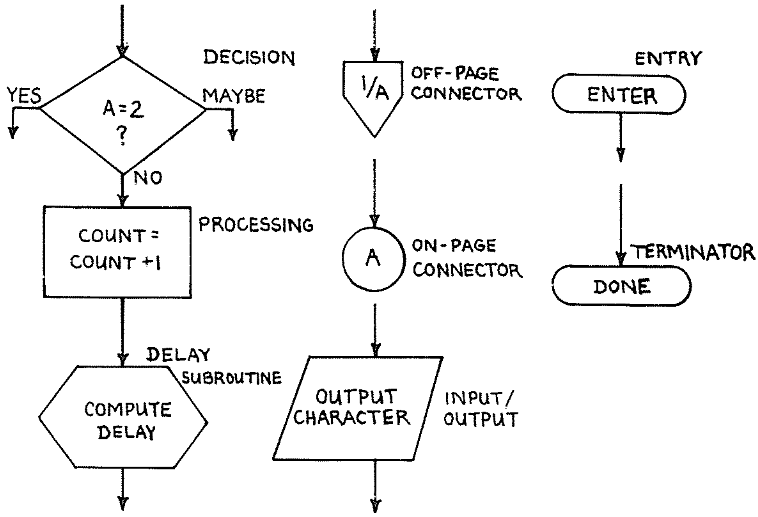
"Do you have your flow-chart template?" Cal asked Ted. "I've got to get this design done before I slip the schedule again."

"Sorry, I used mine to scrape up the TRSDOS diskette I left out in the sun yesterday. The template melted, too."

Hints and Kinks 12–4

''Standard'' Flow Chart Symbols

Here are some symbols commonly used in flowcharting:



Cal found a spare template and set to work. He did not simply produce page after page of flowcharts, but spent a great deal of time contemplating various aspects of the design, occasionally referring to reference materials or notes.

Several days later, Cal was done with the flow charts. A sample of what he produced is shown in Figure 12-2. Although there are several standard flowcharting symbols, Cal used those which his company had established as **their** standard. They were similar to those used by most other programmers in defining programming operations — a rectangular box for "processing," a diamond for a decision point, a "subroutine" symbol, on and off page **connectors**, etc. The important thing to Cal was that he produce a shorthand version of the program he was going to code in standard notation.

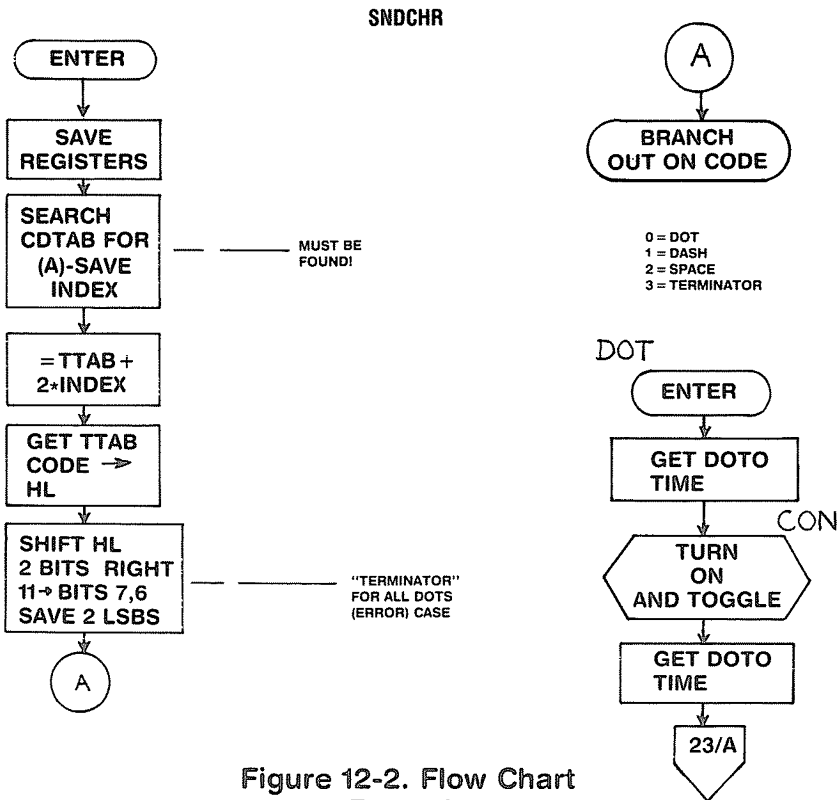


Figure 12-2. Flow Chart Example

"Would you take a look at these when you get a chance?" Cal asked Ted.

Cal's company had a policy of **design reviews** for major jobs. After a programmer had finished specifying and designing a large program, he would distribute copies of the implementation spec and flow charts to the other programmers in the department. Then, after each had had a chance to look them over, a general meeting would be held to review the design.

The meetings produced their share of nit-picking comments but also brought to light omissions or errors in most designs.

Since this was a smaller program, Cal was not obligated to hold a design review, but he did want to have his officemate review the flowcharts to catch any obvious errors.

"Oh, oh!" Ted exclaimed while looking over one page of the flowcharts.

"What's the matter?"

"I don't think you've thought this through! You've got the program picking up a character from the keyboard, converting it to Morse code, outputting the code through the cassette port, and then going back to pick up the next character."

"Right — what's wrong with that?"

"Well, that means that the next character typed can't be output 'til the last one is done!"

"So?"

"If you do that, the user has to wait instead of typing normally. He'd have to type a character, wait for the tone to stop, type the next . . . ."

"Oh," Cal groaned, "I see what you mean. I need a **buffered** input that'll pick up keys even during output of characters!"

Hints and Kinks 12-5

The Problem

If the input was not buffered, each character would have to be output before a new character was read from the keyboard. This doesn't seem annoying in theory, but in practice it would make for Morse code output that would have longer spaces than normal between characters and would inhibit the operator from typing freely.

"You didn't need this job, anyway!" Ted said with a grin and ducked as a copy of *TRS-80 Assembly-Language Programming* went flying by his head.

Cal did some revision and came up with a new set of flowcharts. After another review by Ted, he was ready to do the coding.

In the process of flowcharting, Cal defined several levels of modules. They are shown in Figure 12-3. The reader may want to refer to Figure 13-8 to see how they relate to the actual code of the program

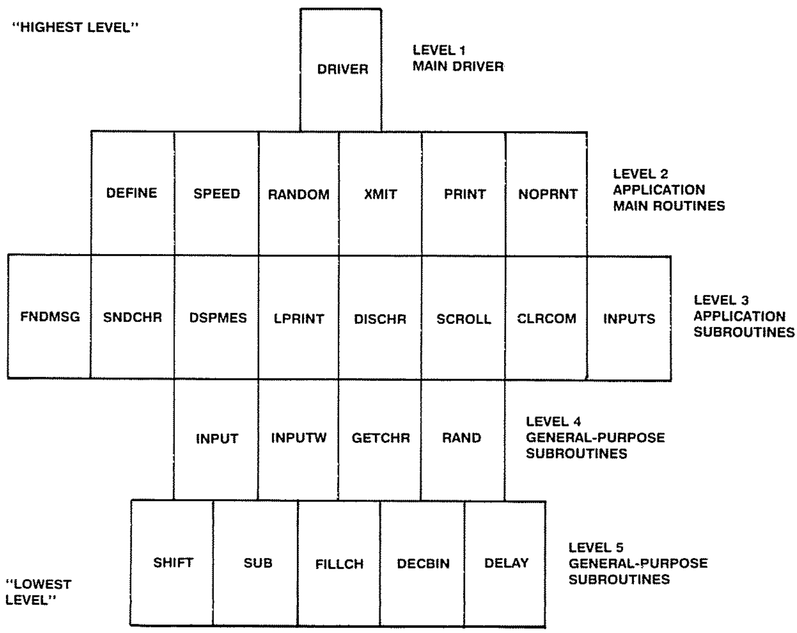


Figure 12-3. Typical Program Modules

Coding

Cal's flowcharts were such that he could have handed them to a junior programmer for coding. On larger jobs, there might be teams of programmers working on the project, each one coding his portion of the job after the implementation specification and flowcharting were done and reviewed. On this job, however, Cal did his own coding.

With the flowcharts and spec done, the coding went very rapidly. Cal didn't use regular coding sheets, although many of the programmers did. He wrote his code in pencil on quadrille pads since he would be entering the code on the system himself. The company maintained data entry operators that would take coding on coding sheets and enter it into the system for assembly, but Cal preferred to avoid the **local queues** on smaller jobs and enter the code himself.

Cal was very familiar with the Z-80 instruction formats, but remembered the problems he had had when he first started working with Z-80 code. It had taken him some time to get familiar enough with the instruction formats that he could automatically put down the correct form. Another problem had been the large number of instruction types available. Now, he rarely consulted the manual for instruction formats and operation.

One thing that Cal made certain of was to put a comment on virtually every line of the program. He knew they would be invaluable not only months later, but would help **days** later in interpreting some of the instructions.

Finally, the coding was done. Cal used his own TRS-80 in the office to enter the code and assembled the program. On the first assembly, he had several dozen assembly errors, most of them produced by a misspelled label. Correcting the errors, Cal reassembled, and after three edits and reassemblies got a "clean" assembly.

He compared the assembly listing with his original code in great detail, and then put the original code away in what he called his "coward's nook," a place he put material he thought he'd never use again but was afraid to throw away.

Desk Checking

Cal's next task was **desk checking**. In this step of producing the MORSE program, Cal went over the assembly listing in minute detail. He compared it with the flowcharts for **logical errors** and looked at each section of code for such things as registers destroyed in subroutines when they should not have been and errors in using the stack.

On this last point, Cal paid particular attention to detail. He made certain there was a POP for every PUSH and a RET for every CALL, or at least a proper adjustment of the stack pointer. More than once he had had his program "gobbled up" by a hungry stack.

On some of the code Cal "played computer" and used a paper and pencil to record the actual operation of a subroutine, just as the Z-80 would do. He drew a line with all Z-80 registers as a heading, assumed some initial entry data for the subroutine, and then laboriously went through instruction by instruction, decrementing registers, adding results, and counting loops. One of his sessions for the SCROLL subroutine is shown in Figure 12-4.

DE	HL	B	A	BC
0	3			
		7		
	6	6		
	12	5		
	24	4		
	48	3		
	96	2		
	192	1		
	384	0		
		3		
	381	2		
	378	1		
	375	0		
			7F	
			0	0

SEED			
0	0	375	

STACK
~~HL~~
~~DE~~
~~AF~~

Figure 12-4. Playing Computer

As a result of the desk checking, Cal edited and reassembled twice again. The first time picked up some fairly important errors he had made in logic, while the second time cleaned up some minor problems he had found on the desk check after the first reassembly. Cal finally felt confident enough in the code to try **on-line** debugging.

> Hints and Kinks 12-6
> How Much Desk Checking?
>
> How much desk checking is necessary? If you were debugging on a large system with many programmers and limited access, you'd want to do a great deal of desk checking. On your own TRS-80, however, do enough to keep from being forced to reassemble more than about once every four hours of debugging time. This usually means a couple of passes through all the code initially with fairly close scrutiny. Find a comfortable trade-off between time spent in reassembling and time spent desk checking.

"Ok, Ted, I'm going to run this turkey. Want to see it work the first time?"

Ted raised his eybrow and said, "I **almost** had a program run the first time. Of course, it was only three instructions long . . . "

Debugging

Actually, at this point, many of Cal's debugging problems were over. He had written a good implementation spec, flowcharted the program, had a one-man review, and had done comprehensive desk checking. The debugging task at this point would probably only be for minor oversights.

Cal loaded MORSE and then created a disk file version by the DUMP command. Loading it in again, he used Disk DEBUG to start execution at location 8000H. The screen filled with "@@@@@@@@@..." and the disk "rebooted."

"That wasn't quite the result I expected," muttered Cal as Ted turned back to his desk.

Cal debugged the program by using the "binary search" on errors we discussed in earlier chapters. By the end of the day, he had patched the program to the point where it ran through the initialization fairly successfully and even output a single character on the cassette audio. The next error he found, however, was one that was not easily patched, and he was forced to reedit and reassemble.

It took Cal several more days of debugging to get to the point where he could find no more **known errors**. At that point he turned to his next task in the production cycle.

Comprehensive Testing

GIGO, Inc. was a software house producing software for the TRS-80 and other computers. The company had long ago learned to perform comprehensive tests on their software, rather than releasing it prematurely. Of course, there was constant pressure from Sales for a volume of interesting new products, but the Programming Department manager was firm in his resolve to eliminate as many bugs as possible before releasing the product (he was also the son-in-law of the chairman of the board).

Cal spent several days drawing up a **test plan** that would exercise as many of the features of MORSE as possible. It included such things as a check to see that all characters would be properly output, that all speeds could be utilized, that all **limit** conditions (such as 256 characters per message) worked, and that there was an even distribution of random characters.

When he finished, he had a formalized test plan on paper. This was submitted to his manager, Paul, and kept in the program file. Cal then went down the test plan step by step and tested each item. He found several minor discrepancies and several **human factor** improvements, such as the length of time that error messages were displayed on the screen.

After the last test item had been cleared up, Cal reassembled the "last version" of the program, went through the test plan again and found no errors. He knew that it was entirely possible that lurking in the depths of the MORSE program were small bugs that might grow their way to the surface when a user in Chillicothe decided to transmit four error codes together while using the printer during a quarter moon. He also knew that it was virtually impossible to eliminate all bugs except by continued testing over great lengths of time. However, he was confident that most users would be very happy with the program.

With some trepidation, Cal walked into Paul's office and said, "Well, here it is, Paul — the final version of MORSE!"

"Great, Cal. By the way, I'm glad you're here. I just got in a request to develop a TRS-80 program to learn Tic-Tac-Toe. . . ."

Final Clean-Up

Cal's work on MORSE wasn't quite done. He revised the final specs on the program and filed the specs, listing, source code, and working program. He knew the importance of this because when he first joined the company, he had to take two existing programs and correct errors that had been discovered. The programmer who had written them, "No Comment" Garigan, had left the company to become a tour guide in Pismo Beach.

Garigan's code contained, as his nickname implied, virtually no comments on any lines. His flowcharts were incomplete, his design specs nonexistent. Since that time, Cal had made it a special point to be as thorough in documentation as possible. He knew it was entirely possible that one of those tiny bugs might become the "Monster That Ate the TRS-80" and that he himself might be forced to correct his own code.

No Resemblance to Programmers Living or Dead

Although the scenario above is ficticious, it is an attempt to show an idealized situation in program design and development. Most programs are developed under less formalized steps, and many delete such important steps as flowcharting and final testing. In developing your own assembly-language programs, you'll adopt the methods that work for you, but it may actually take **less** development time to go through the procedures defined above . . . so some programmer a year from now won't nickname **you** "No-Comment" Garigan!

In the following two chapters, we'll show the flowcharts and listings for two large assembly-language programs: one for Morse Code Generation and one for a Tic-Tac-Toe program that "learns." These should clearly represent some of the elements of **programming style** that we've discussed.

Chapter Thirteen
A Morse Code Generator Program

In this chapter we'll look at the design and implementation of a program to generate Morse code by audio tones sent to the cassette output port. The program may be used for code practice, or it may be used for amateur radio applications to record and playback messages and normal keyboard characters at speeds of 3 to 60 words per minute. One of the features of the program is that it is fully **buffered** — messages being typed on the keyboard may "overrun" the actual Morse code output and may be dozens of characters ahead.

This program was designed as a typical assembly-language application for this book to illustrate some of the concepts discussed Chapter 12 on large program design and implementation. Since we'll also discuss the code in the program, we'll be tying together many of the coding concepts discussed earlier.

General Specification

The general specification for the program is shown in Figure 12-1. (Only the name has been changed from "MORSE" to "MORG".) The program has basically three modes. The first mode is keyboard output. Characters typed on the keyboard will be output as audio-frequency Morse code characters from the cassette output (the plug that normally attaches to the cassette recorder "AUX" jack). A small, inexpensive amplifier can be used to play the resulting output through a small speaker. As characters are output, they are displayed on the screen and can be printed on the system line printer.

The second mode of operation is integrated with the first. You can generate predefined messages of up to 256 characters in length by a single keystroke. Up to ten messages can be defined and generated, and you can intersperse messages and normal text. As characters are output, they are displayed on the screen and optionally printed on the system line printer. A typical message might be defined as message 5 and read CQ CQ CQ DE WD6CTY WD6CTY WD6CTY K. Every time you pressed the "SHIFT, 5" keys, this message would be generated.

The third mode of operation functions independently of the other two. With Random mode, you can generate an endless string of pseudo-random characters for code practice. As the characters are output to the cassette port, they are displayed on the screen and can be printed on the system line printer. The speed of operation of any of these modes can be defined to be 3 to 60 words per minute.

Operation

After loading, MORG clears the screen, draws a line near the bottom to define a "communication area," and outputs a title message of

MORG

CHAR = SEND CHARACTER   SHIFT 0-9 = SEND MSG N
SHIFT R = SEND RANDOM   SHIFT D = DEFINE MSG
SHIFT S = DEFINE SPEED   SHIFT P,N = PRINT OR NO

You can now choose one of the options in the title message. To define the speed, you press SHIFT S and enter a speed value of 3 to 60 words per minute. After the speed is defined, the title message is again printed. To set simultaneous printing on the system line printer, you press SHIFT P. The line printer will now echo the Morse characters displayed on the screen. To disable the printer at any time, you press SHIFT N. Both actions terminate by display of the title message. To define a message, you press SHIFT D. The define message mode is now entered and will prompt you to input a message number of 0 through 9 and to enter the text of the messge. Any number of characters may be entered for the text up to 256. You terminate the message by an ENTER key press, bringing you back to the title message display.

To output characters and messages, you can type in characters from the keyboard. Each time you type a legitimate character, there is an output in Morse code from the cassette output at the currently defined rate of speed. Illegitimate keys are ignored. Each time you enter a SHIFT (followed by 0 through 9) the previously defined corresponding message will be output in its entirety. If no message has been defined, nothing will be output. As keyboard text or messages are output, the text is simultaneously displayed on the screen and output to the system line printer if Print has been set.

If you have chosen the Random mode by SHIFT R, a continuous stream of pseudo-random text will be output to the cassette port, simultaneously displayed on the screen, and (optionally) printed on the system line printer. The speed will be at the currently defined rate of speed. The Random mode simulates normal text by inclusion of spaces at regular intervals.

Hints and Kinks 13-1
Simulating Text

Text is simulated here by including spaces in the CTAB, the 128-byte table of random characters. There are 18 spaces out of 128 characters, making the frequency about a space every six characters. The program also checks to make certain that two spaces are not sent consecutively.

Legitimate characters are the alphabetic characters A through Z, digits 0 through 9, comma (,), period (.), slash (/), question mark (?), and three special characters. The three special characters are a dash for a "break" (—....—), an equals for end of transmission (.—.—.), and a semicolon for "error" (........). All other characters are ignored.

General Design

The general design considerations or research into the methods of Morse code generation can be broken down into the following areas:

- Characteristics of Morse code
- Generation of audio tones
- Keyboard read routines
- "Buffering"
- Conversion to Morse code characters
- Message storage and search

We'll discuss each of these areas, the alternative methods that could have been used, and the final method that was adopted in MORG.

Characteristics of Morse Code

Morse code consists of a series of **dots** and **dashes** to represent alphabetic, numeric, and special characters as shown in Figure 13-1. A dot is defined as a short burst of an audio tone or other signal, while a dash is a longer tone. The combinations of dots and dashes used are generally based on the frequency of the letter in normal text. An "e" for example is one dot, while a "q" is dash, dash, dot, dash. The special character question mark (?) is dot, dot, dash, dash, dot, dot.

A	.–	N	–.	0	–––––	/	–..–.
B	–...	O	–––	1	.––––		
C	–.–.	P	.––.	2	..–––		
D	–..	Q	––.–	3	...––		
E	.	R	.–.	4	....–		
F	..–.	S	...	5	.....		
G	––.	T	–	6	–....		
H	....	U	..–	7	––...		
I	..	V	...–	8	–––..		
J	.–––	W	.––	9	––––.		
K	–.–	X	–..–	.	.–.–.–		
L	.–..	Y	–.––	,	––..––		
M	––	Z	––..	?	..––..		

"T"	"R"	"S"	"8"	"0"
–	.–.	...	–––..	–––––

Figure 13-1. Morse Code Symbols

The relationship of dots, dashes, spaces between dots and dashes, and spaces between words is shown in Figure 13-2. A dot time is the basic unit, a dash time is three units, the time between dashes and dots is one unit; the time between characters is three units; and the time between words is five units. When an audio tone is used for the Morse code, the tone is on during the dot or dash time and off during the spaces as shown.

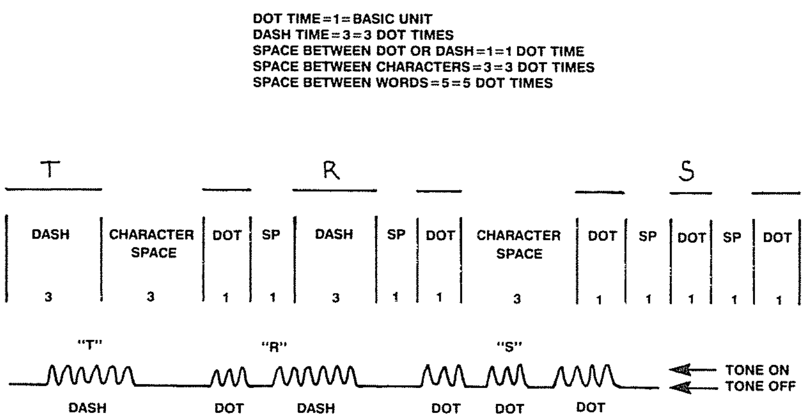


Figure 13-2. Morse Code Spacing and Audio

A slow speed for Morse code transmission is 5 words per minute. This is the speed used for the amateur radio "novice" code qualifying test. Average speeds are fifteen words per minute, while high-speed "brass pounders" send at rates of 35 words per minute or greater.

An average word is 5 characters, making the number of characters per minute 5*WPM (*W*ords *P*er *M*inute). Fifteen WPM is therefore about 75 characters per minute, or about 1.25 characters per second. The average number of dots and dashes in a character is hard to define. One formula relates the number of WPM to dots per second:

```
WPM=DPS*2.5
```

This would mean that 10 dots per second are equivalent to 25 WPM. We'll use this formula for our analysis in MORG.

In MORG, the code speed may vary between 3 and 60 WPM, allowing for a very slow user and a very fast one. The number of dots per second for 3 WPM is 1.2; the number of dots per second for 60 WPM is 24. Since a string of dots consists of a dot **on** time and a dot **off** time, this makes the dot duration:

Dot duration in milliseconds = 1200/WPM

For example, the duration of a dot at 3 WPM is 1200/3, or 400 milliseconds, while the duration of a dot at 60 WPM is 1200/60, or 20 milliseconds. On the surface, it appears that an assembly-language program should have no problem in achieving these speeds, since 20 milliseconds, the most stringent case, represents 4000 **instruction times** at 5 microseconds per instruction (5 microseconds*4000 = 20000 microseconds or 20 milliseconds).

The assembly-language program's main concern is to turn on an audio tone, leave it on for lengths of time ranging from 20 milliseconds (60 WPM dot) to 1.2 seconds (3 WPM dash), and turn it off.

Generation of Audio Tones

As we know from Chapter 10, you can easily use the TRS-80 to generate a wide range of audio tones. In this case, we need to generate only one frequency, unless we choose to make the pitch variable.

> Hints and Kinks 13–2
> Generating Tones
>
> You can accomplish tone generation here by ''toggling'' the cassette output latch on and off. You output alternating 01s and 10s to the cassette latch address 0FFH. The 01 causes a ''high'' level, and the 10 causes a ''low'' level. A zero reference level would be created by 00.

The audio tone will be on for a dot or dash on time and off for inter-character or inter-word spacing. Our only problem would be the shortest duration dot-time that will have to be handled. Does this represent a duration during which we can generate a tone or is the duration so short that we can't **toggle** the cassette latch on and off to produce a tone?

The shortest duration dot-time from above is 20 milliseconds. During that 20 milliseconds, we must toggle the cassette latch on and off several times. If we did it one time, on and off, the **period** of the tone produced would be 20 milliseconds.

As the **frequency** of such a tone is 1/period, or 50 cycles per second, there seems to be no problem in the tone. Fifty cycles per second is too low for both comfortable listening and amplification on an inexpensive amplifier. If we tried for 500 cycles per second (500 **hertz**), we would be toggling the cassette latch ten times on and off for the shortest duration dot, which would be fine.

Keyboard Read Routines

We know from Chapter 7 that we can hand-tailor a keyboard read routine to read any key and convert it to whatever character we want. In most cases here, we'll be doing a "straight" conversion into the ASCII code associated with the key. However, we will have to convert some keys into special codes to represent characters for "end of message," "error," and the like. Still, we should have no problem in the conversion.

Buffering Keyboard Input & Polling the Keyboard: Problems

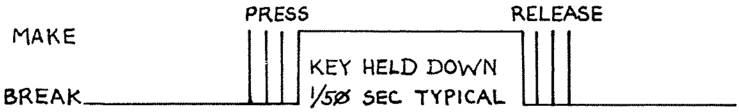
There **will** be a problem in the area of keyboard speed, however! The specification allows for **buffering** keyboard input. This means we can be typing at a faster rate than the data is being transmitted in Morse code audio. We'll have to store the characters in some sort of large buffer, depending upon how many characters ahead we wish to store.

This also means that as the character is being transmitted, we must **poll** the keyboard to see if the next character is being input. If it is, we must **debounce** the character and store it in the buffer. What other problems do we face at this point?

One major problem is this: If we use a time-delay-loop in order to generate the on/off times that toggle the cassette latch to produce an audio tone, we can't be going out during the generation of dots and dashes to **poll** the keyboard!

Hints and Kinks 13-3
Debouncing

Debouncing works like this: When a key is depressed, it bounces up and down in the space of fractions of milliseconds, making and breaking the contact. When a key is released, the same make and break occurs. Keys are held down for approximately 1/50th of a second. The make/break diagram looks roughly like this:



"Rollover" allows the next key to be detected when a previous key is still being held down. MORG does not have rollover, but looks for the next key 100 milliseconds or so from the last. This would allow for typing speeds up to 120 words per minute.

Debouncing detects the first "make" and then waits 100 milliseconds to bypass bounce on the press and release.

Scanning (or polling) the keyboard normally takes 50 to 100 milliseconds (1/20th to 1/10 second) because the character has to be debounced. If we wait this long to debounce a character that's being input, it'll extend the dot or dash time for high-speed transmission (for instance, a dot-time for 20 WPM is 60 milliseconds). If we delay 50 milliseconds to debounce, the dot-time changes to 110 milliseconds! This is clearly unacceptable.

What to do, what to do .... On the one hand, we need to be able to read the keyboard during transmission of characters to eliminate a tedious "wait until complete before reading keyboard" condition. On the other hand, it appears we can't debounce the character because it will affect the dot and dash times.

Is there any other way to **time-out** the bounce? If there were, we could rush out for a fraction of a millisecond or so during dots, dashes, or spaces and check to see if a key were pressed. If a key **were** pressed, we could quickly decode the key and store it in a buffer, start some sort of timer, and then go back to the dot/dash/space generation. We could then periodically check for an elapsed time of 50 milliseconds or so. If 50 milliseconds had elapsed, we could reset the timer and go back and repeat the process again.

A disk system would certainly let us perform the timing since it counts in increments of 25 milliseconds. But we may not have a **real-time clock** if we don't have a disk system. Is there some way to **emulate** the real-time-clock function?

One way would be to establish a counter inside our program. This software counter would count in increments of 1 millisecond. As we'll be using a subroutine to time the cycles of the audio tone, we could use the delay count for the delay subroutine to increment the counter by the number of milliseconds delayed. For a continuous stream of characters, this would give us a fairly accurate count of elapsed time.

However, we don't have a continuous stream of characters, do we? Sure, some times we've buffered a large number of characters, and they keep on being generated at the current speed. But what about the case where the user is doing a "hunt and peck" at the keyboard? If the counter is only incremented during the time delays for the audio tones, it certainly won't be a record of elapsed time.

Well, we'll be looping and waiting for that next character from the keyboard and that loop should be at a fairly constant speed. We can establish an increment of the software millisecond counter every "n" times through the loop. This should work for the rough elapsed time needed for debounce. After all, it doesn't matter too much whether we delay 50 milliseconds or 100 milliseconds for a maximum typing speed of 60 WPM (or five characters per second). Is this the best way to handle this problem? Probably not, but it's **one** way given the constraints of the system — no real time clock, no **interrupts**, and a need for buffering.

To minimize the time required to rush out and poll the keyboard, we'll need a fast keyboard scan routine, one that determines quickly if a character is there, and if not, returns to the calling routine. It'll also have to test for the debounce delay time from the software counter.

Buffering

What about the problem of **buffering**? How do we store the characters as they are read in and then transmit them?

To do this we'll need a buffer of a certain number of bytes. As the characters are read from the keyboard, they are stored in the next **slot** of the buffer. The cassette tone output will have to somehow test the contents of the buffer to see if there is any new character available for output.

If the operator is a very slow typist and the code speed is slow enough, this buffer will be filled with one character and then immediately emptied. However, if the operator is fast and the code speed is slow, the buffer may fill up rapidly with characters that are being input too fast to be handled.

What about the size of the buffer? If the program runs for any length of time, we'd need a very large buffer. A better way to do this would be to establish a **circular buffer** based on the worst-case input and output speed. We'll assume that the operator can't get more than 256 characters ahead before he gets confused! This should be adequate for most of us.

The circular buffer is used as shown in Figure 13-3. A new character goes into the next slot. A pointer to the next slot is then incremented by one. If we reach the end of the buffer, the pointer is set back to the beginning of the buffer. A second pointer points to the "next character available." If the two pointers are equal, all characters have been used. "Overrun" is possible if the number of unused characters exceeds 256.

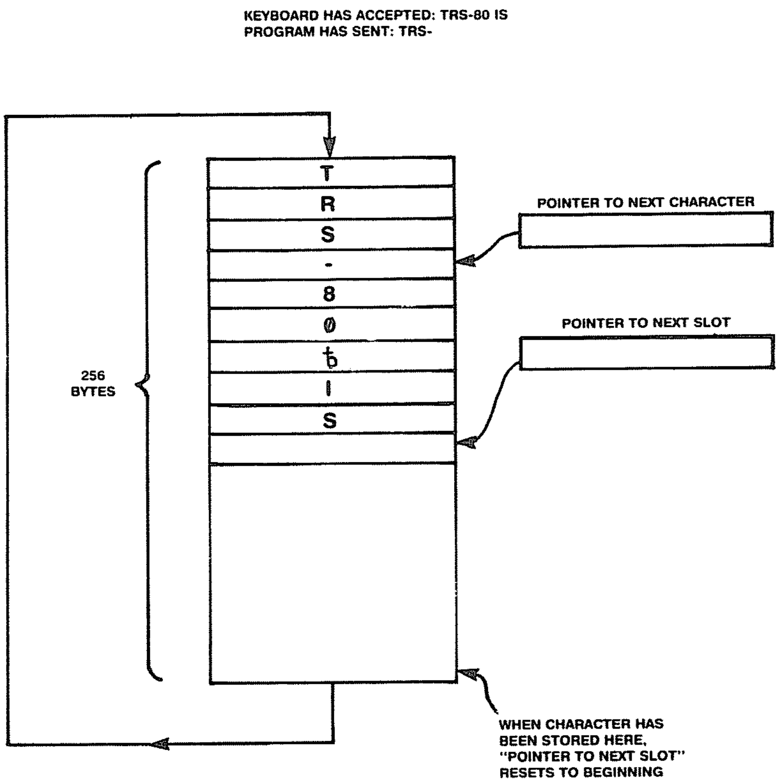


Figure 13-3. Circular Buffer

Conversion to Morse Code Characters

How will a given ASCII character be converted to a Morse code character? We have 26 alphabetic characters, ten digits, blank, and seven special characters to be concerned with. The Morse code combinations are completely unrelated to ASCII codes or to any other "neat" method of finding the proper combination of dots and dashes.

The implication is that instead of a formula or **algorithmic** means to produce the Morse code, we need to translate an ASCII character into a Morse code character by a **table lookup**. One Morse code configuration will be in the table for each legitimate ASCII character.

What about storage of the dots and dashes in the table? Each character we're concerned with is represented by six or fewer dots or dashes, except for the "error" code. It looks like we could use six codes for each character. One way to do this would be to store a fixed eight-byte entry for each character, with 0 representing a dot, 1 a dash, 2 a space, and -1 a terminator (or we could pack two codes in each byte).

After some thought, we came up with this: Each combination of dots and dashes is held in 16 bits, or two bytes. There are eight **fields** in the two bytes. Each field contains a two-bit code. A code of 00 represents a dot, a code of 01 represents a dash, a code of 10 is a dot space, and a code of 11 is a terminator. The 8 fields read from **right to left**. Each combination ends with a terminator field of 11, allowing up to seven dots, dashes, or dot spaces, plus a terminator.

This scheme is shown in Figure 13-4.

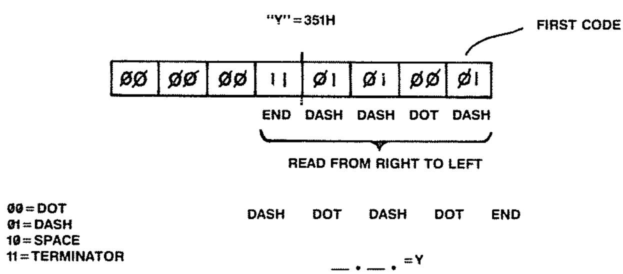


Figure 13-4. Dot, Dash, Space Coding

The special case of eight dots ("error code") is represented by two bytes of zeroes and is detected in the program.

Message Storage and Search

Messages are stored as ASCII character strings. A large buffer capable of holding 10*257+1 or 2,571 characters (and ten one-byte "headers") is established. A minus one is used as a terminator for the last message in the buffer. Unused spaces in the buffer are filled with minus one bytes.

All bytes in the buffer are either valid ASCII characters, minus ones, or the binary digits 0 through 9. A binary digit 0 through 9 marks the **header** of a message string. The length of the string is determined by the next occurrence of a digit, or a minus one.

Messages are put into the buffer as they are defined. A search is made for the header number to find a specified message. If a message is redefined, the old message is first deleted by "moving up" the remaining buffer into the space previously occupied by the message. This approach is shown in Figure 13-5.

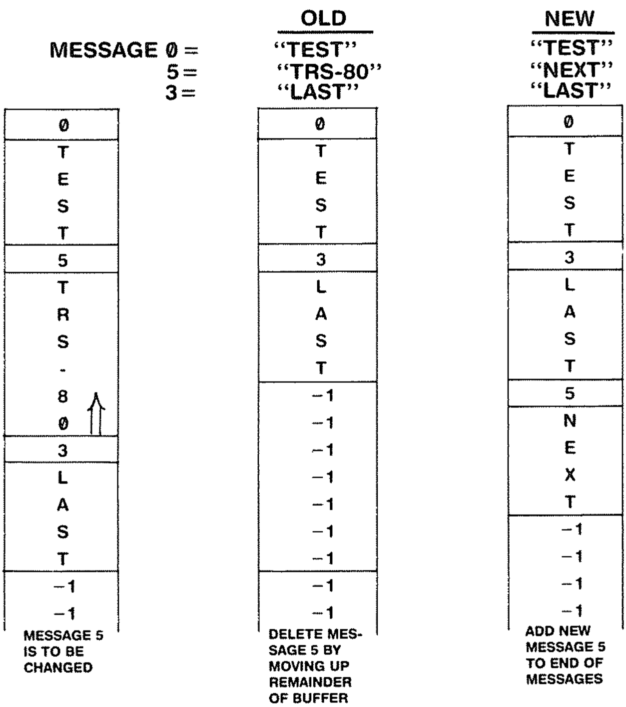


Figure 13-5. Message Storage

Implementation

Modules

MORG is implemented as a series of five levels of modules, shown in Figure 13-6. The top level of modules is the main **driver** of the program. The bottom level of modules are the most rudimentary subroutines of the program.

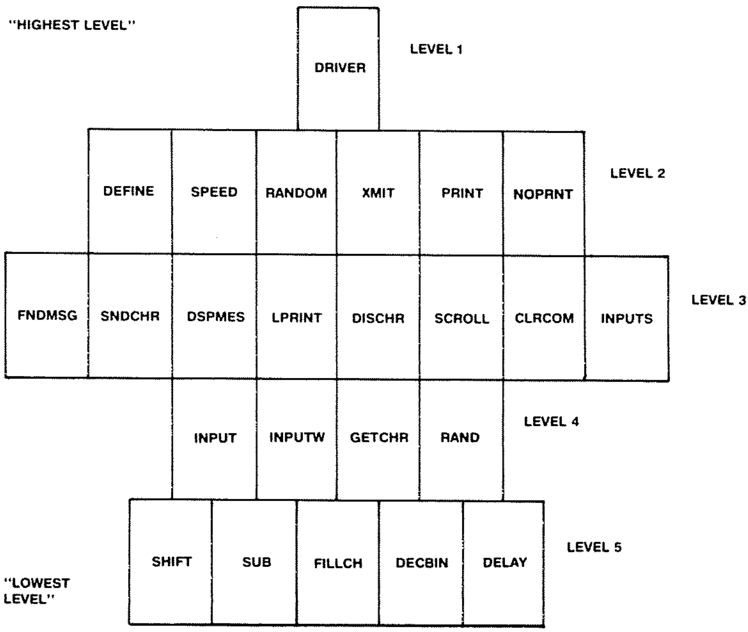


Figure 13-6. MORG Modules

The modules are generally dedicated to a particular well-defined function related to the MORG application, such as "finding a message number" or "clearing the communications area." Each module is generally in the form of a subroutine with one entry point and one exit point. The lowest level of subroutines are general-purpose subroutines, useful for many applications.

Figure 13-7 shows the modules and their interconnections. Each module generally calls another module by a CALL, with a set of parameters in the CPU registers. Higher level modules usually call lower-level modules, although some modules call other modules on the same level.

Hints and Kinks 13-4
Notes on Figure 13-7

The module interconnection diagram of Figure 13-7 reveals some interesting facts about the structure of MORG. The level one driver not only calls every level two routine but also does applications-related processing by calling level three routines, which are special purpose routines for the Morse code application. Level two routines also make heavy use of level three routines.

If we see only one connection (dot) along any horizontal connection to a module, it might indicate that the modules should be incorporated as ''in-line'' code. This occurs for SHIFT, SUB, and RAND.

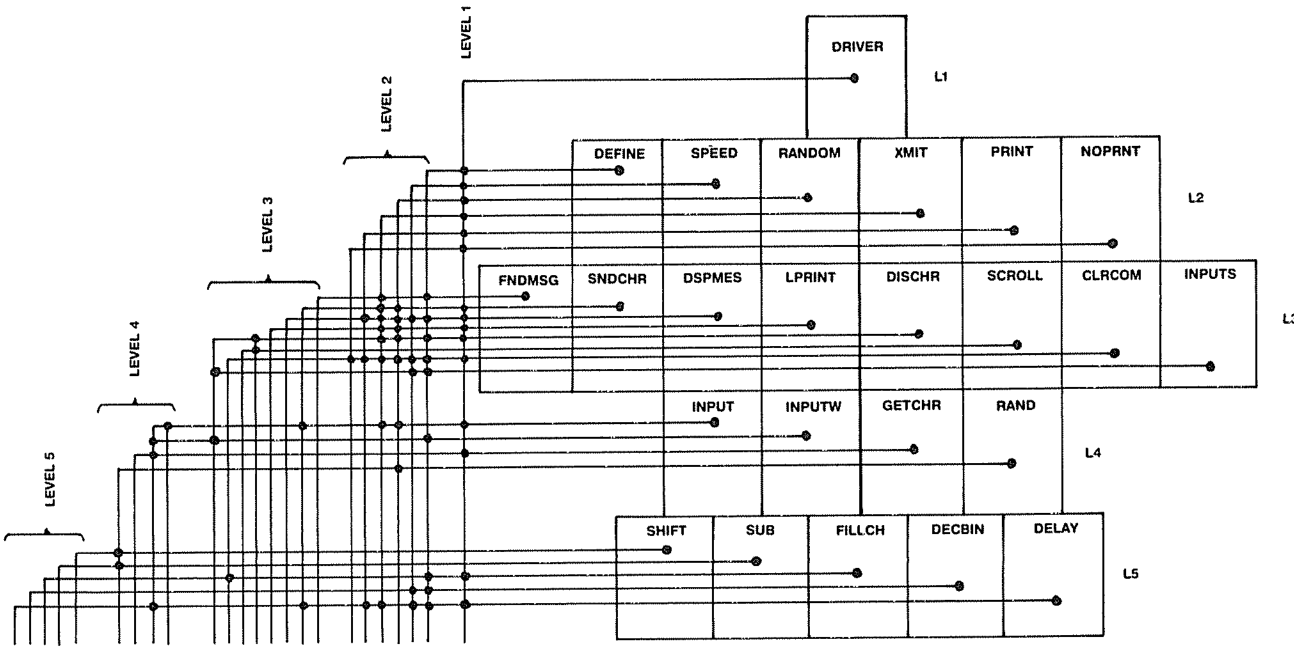


Figure 13-7. MORG Module Interconnections

To find the modules called by any particular module, follow the horizontal line from the module to the extreme left. When the line turns downward, read the lower level module called by referencing the connection dots. The arrangement of the modules is duplicated in the program listing, that is, higher-level modules appear first, followed by lower-level modules.

Tables, Buffers, and Variables

Refer to the MORG listing, Figure 13-8. Here is a set of variables, buffers, and tables the program uses to define system status, hold text, and facilitate conversion. The variables are held in the working storage area of the program, near the end. For the most part, their use is explained by the comments associated with each one.

```
8000                    ORG     8000H
                ;MORG-0820
                ;**************MORSE CODE GENERATOR PROGRAM**************
                ;
                ;
                ;********************SYSTEM EQUATES*********************
                ;
3C00            SCREEN  EQU     3C00H           ;START OF VIDEO DISPLAY
3C40            LINE1   EQU     SCREEN+64       ;SECOND LINE
3EC0            LINE11  EQU     SCREEN+704      ;TWELFTH LINE
3F00            LINE12  EQU     SCREEN+768      ;THIRTEENTH LINE
3F40            LINE13  EQU     SCREEN+832      ;FOUTEENTH LINE
3F80            LINE14  EQU     SCREEN+896      ;FIFTEENTH LINE
3FC0            LINE15  EQU     SCREEN+960      ;SIXTEENTH LINE
0002            ENTER   EQU     2               ;ENTER CHARACTER
0001            CLEAR   EQU     1               ;CLEAR CHARACTER
0064            DBDEL   EQU     100             ;DEBOUNCE DELAY IN MS
0065            DBDELP  EQU     DBDEL+1         ;DEBOUNCE DELAY+1 MS
000A            MLDEL   EQU     10              ;MAIN LOOP DELAY IN 1/10 MS
03C0            SPEEDF  EQU     960             ;FINAGLE FACTOR FOR SPEED
                ;
                ;********************MORSE EXECUTIVE********************
                ;
8000 F3         START   DI                      ;DISABLE INTERRUPTS
8001 315994             LD      SP,TOPS         ;SET STACK POINTER
8004 11EA88             LD      DE,MBUF         ;MESSAGE BUFFER ADDRESS
8007 010B0A             LD      BC,2571         ;2571 BYTES
800A 3EFF               LD      A,0FFH          ;-1 FOR FILL
800C CD8385             CALL    FILLCH          ;FILL MESSAGE BUFFER
800F 21D204             LD      HL,1234         ;INITIALIZE RANDOM # SEED
8012 22E185             LD      (SEED),HL
8015 212E16             LD      HL,5678
8018 22E385             LD      (SEED+2),HL
801B 3E20               LD      A,' '           ;BLANK CHARACTER
801D 11003C             LD      DE,SCREEN       ;START OF SCREEN
8020 010004             LD      BC,1024         ;# OF BYTES
8023 CD8385             CALL    FILLCH          ;CLEAR SCREEN
8026 3E8F               LD      A,08FH          ;ALL ON GRAPHICS CHAR
8028 11003F             LD      DE,LINE12       ;LINE 12
802B 014000             LD      BC,64           ;# OF BYTES
802E CD8385             CALL    FILLCH          ;DRAW LINE
8031 21003C             LD      HL,SCREEN       ;SCREEN START
8034 22E985             LD      (CURCUR),HL     ;INITIALIZE CURRENT CURSOR
                ; REENTER HERE FOR MOST FUNCTIONS
8037 216400     MOR015  LD      HL,DBDEL        ;DEBOUNCE DELAY IN MS
803A CDC085             CALL    DELAY           ;DELAY
803D 216500             LD      HL,DBDELP       ;MINIMUM DELAY +1
```

```
8040 22ED85          LD      (TSLC),HL       ;INITIALIZE FOR NEXT CHARACTER
8043 21F592          LD      HL,IBUF         ;INPUT BUFFER ADDRESS
8046 22F085          LD      (IBUFL),HL      ;INITIALIZE INPUT BUFFER PNTRS
8049 22F285          LD      (IBUFN),HL
804C 2AE985          LD      HL,(CURCUR)     ;GET CURRENT CURSOR
804F 22EB85          LD      (LSTCUR),HL     ;SAVE
8052 CDFB83          CALL    CLRCOM          ;CLEAR COMMUNICATIONS AREA
8055 21F485          LD      HL,MSG1         ;INITIAL MESSAGE
8058 01403F          LD      BC,LINE13       ;LINE 13
805B CD6D83          CALL    DSPMES          ;OUTPUT MESSAGE
805E 2AEB85          LD      HL,(LSTCUR)     ;GET OLD CURSOR
8061 22E985          LD      (CURCUR),HL     ;RESTORE
               ; REENTER HERE DURING TRANSMISSION OF CHARACTERS OR MSGS
8064 1600      MOR020  LD      D,0              ;INITIALIZE MINOR COUNT
8066 CD2984    MOR021  CALL    INPUT              ;GET RESPONSE
8069 CD1585            CALL    GETCHR             ;GET CHARACTER
806C 200F              JR      NZ,MOR022          ;GO IF PRESENT
806E 14                INC     D                  ;BUMP MINOR COUNT
806F 7A                LD      A,D                ;GET COUNT
8070 FE0A              CP      MLDEL              ;TEST FOR 1 MS
8072 20F2              JR      NZ,MOR021          ;GO IF NOT 1 MS
8074 2AED85            LD      HL,(TSLC)        ;GET TIME
8077 23                INC     HL               ;BUMP BY 1 MS
8078 22ED85            LD      (TSLC),HL        ;RESTORE
807B 18E7              JR      MOR020           ;CONTINUE
807D B0        MOR022  OR      B               ;MERGE SHIFT BIT
807E 21CC80            LD      HL,FTAB+FTABS-1 ;FUNCTION TABLE END ADD
8081 060F              LD      B,FTABS         ;FUNCTION TABLE SIZE
8083 BE        MOR025  CP      (HL)              ;TEST FOR FUNCTION
8084 2825              JR      Z,MOR030          ;GO IF FOUND
8086 2B                DEC     HL                ;POINT TO NEXT FUNCTION
8087 10FA              DJNZ    MOR025            ;CONTINUE
8089 E67F              AND     7FH             ;RESET BIT 7
               ; NOT FUNCTION HERE - TRANSMIT A SINGLE CHARACTER
808B 32DC85            LD      (TMP1),A        ;STORE IN TEMP BUFFER
808E 3EFF              LD      A,0FFH          ;-1
8090 32DD85            LD      (TMP1+1),A      ;STORE TERMINATOR
8093 21DC85            LD      HL,TMP1         ;ADDRESS OF "MESSAGE"
8096 7E        MOR027  LD      A,(HL)            ;GET NEXT CHARACTER
8097 FE20              CP      ' '               ;TEST FOR NON-ASCII
8099 FA6480            JP      M,MOR020          ;GO IF END OF MESSAGE
809C CDBE83            CALL    DISCHR            ;DISPLAY CHARACTER
809F CD8183            CALL    LPRINT            ;PRINT IF REQUIRED
80A2 CDE082            CALL    SNDCHR            ;SEND CHARACTER
80A5 23                INC     HL                ;POINT TO NEXT
80A6 CD2984            CALL    INPUT             ;TEST FOR INPUT
80A9 18EB              JR      MOR027            ;CONTINUE SENDING
80AB 48        MOR030  LD      C,B             ;INDEX+1 NOW IN C
80AC 0D                DEC     C               ;ADJUST FOR INDEX
80AD 0600              LD      B,0             ;INDEX NOW IN BC
80AF CB21              SLA     C               ;2*INDEX NOW IN BC
80B1 DD21CD80          LD      IX,BTAB         ;BRANCH TABLE
80B5 DD09              ADD     IX,BC           ;POINT TO BRANCH
80B7 DD6601            LD      H,(IX+1)        ;GET MSB OF ADDRESS
80BA DD6E00            LD      L,(IX)          ;GET LSB OF ADDRESS
80BD E9                JP      (HL)            ;BRANCH OUT
               ;
               ; FUNCTION TABLE
               ;
80BE C4        FTAB    DEFB    'D'+80H         ;DEFINE MESSAGE
80BF D3                DEFB    'S'+80H         ;DEFINE SPEED
80C0 D2                DEFB    'R'+80H         ;TRANSMIT RANDOM
80C1 B0                DEFB    '0'+80H         ;TRANSMIT MESSAGE 0
80C2 B1                DEFB    '1'+80H         ;                 1
80C3 B2                DEFB    '2'+80H         ;                 2
80C4 B3                DEFB    '3'+80H         ;                 3
80C5 B4                DEFB    '4'+80H         ;                 4
80C6 B5                DEFB    '5'+80H         ;                 5
80C7 B6                DEFB    '6'+80H         ;                 6
80C8 B7                DEFB    '7'+80H         ;                 7
80C9 B8                DEFB    '8'+80H         ;                 8
80CA B9                DEFB    '9'+80H         ;                 9
80CB D0                DEFB    'P'+80H         ;SET PRINT
80CC CE                DEFB    'N'+80H         ;RESET PRINT
000F           FTABS   EQU     $-FTAB          ;SIZE OF FUNCTION TABLE
               ;
               ; BRANCH TABLE
               ;
```

```
80CD EB80     BTAB    DEFW    DEFINE          ;DEFINE MESSAGE
80CF 9181             DEFW    SPEED           ;DEFINE SPEED
80D1 F081             DEFW    RANDOM          ;TRANSMIT RANDOM
80D3 3782             DEFW    XMIT            ;TRANSMIT MESSAGE 0
80D5 3782             DEFW    XMIT            ;                 1
80D7 3782             DEFW    XMIT            ;                 2
80D9 3782             DEFW    XMIT            ;                 3
80DB 3782             DEFW    XMIT            ;                 4
80DD 3782             DEFW    XMIT            ;                 5
80DF 3782             DEFW    XMIT            ;                 6
80E1 3782             DEFW    XMIT            ;                 7
80E3 3782             DEFW    XMIT            ;                 8
80E5 3782             DEFW    XMIT            ;                 9
80E7 8B82             DEFW    PRINT           ;SET PRINT
80E9 B182             DEFW    NOPRNT          ;RESET PRINT
              ;
              ;****************DEFINE MESSAGE N ROUTINE****************
              ;
80EB 2AE985   DEFINE  LD      HL,(CURCUR)     ;GET CURRENT CURSOR
80EE 22EB85           LD      (LSTCUR),HL     ;SAVE
80F1 CDFB83   DEF005  CALL    CLRCOM          ;CLEAR COMMUNICATIONS AREA
80F4 21F586           LD      HL,MSG5         ;DEFINE MESSAGE
80F7 01403F           LD      BC,LINE13       ;LINE 13
80FA CD6D83           CALL    DSPMES          ;DISPLAY DEFINE MESSAGE
80FD 0601             LD      B,1             ;1 CHARACTER
80FF CD0D84           CALL    INPUTS          ;GET CHARACTER
8102 C27F81           JP      NZ,DEF050       ;GO IF GT 1
8105 0601             LD      B,1             ;1 CHARACTER
8107 CD9085           CALL    DECBIN          ;CONVERT TO BINARY
810A C27F81           JP      NZ,DEF050       ;GO IF ERROR
810D 7D               LD      A,L             ;MSG # NOW IN A
810E 21C987           LD      HL,MSG11        ;INPUT MESSAGE
8111 01803F           LD      BC,LINE14       ;LINE 14
8114 CD6D83           CALL    DSPMES          ;DISPLAY MESSAGE
8117 2AEB85           LD      HL,(LSTCUR)     ;GET OLD CURSOR
811A 22E985           LD      (CURCUR),HL     ;RESTORE
811D F5               PUSH    AF              ;SAVE MESSAGE #
811E CDC782           CALL    FNDMSG          ;GET ADDRESS OF MESSAGE
8121 202A             JR      NZ,DEF035       ;GO IF NO CURRENT MSG FOR #
              ; CURRENT MSG FOR # IN MBUF - MUST DELETE
8123 E5               PUSH    HL              ;SAVE START
8124 D1               POP     DE              ;PUT IN DE
8125 23               INC     HL              ;BYPASS # TO TEXT
8126 7E       DEF025  LD      A,(HL)            ;GET CHARACTER
8127 FE20             CP      ' '               ;TEST FOR NON-ASCII OR -1
8129 FA2F81           JP      M,DEF030          ;GO IF NEXT MESSAGE
812C 23               INC     HL                ;BUMP POINTER
812D 18F7             JR      DEF025            ;CONTINUE
812F E5       DEF030  PUSH    HL              ;END
8130 C1               POP     BC
8131 E5               PUSH    HL              ;SAVE NEXT AREA
8132 21F592           LD      HL,ENDM         ;END OF MEMORY
8135 B7               OR      A               ;CLEAR CARRY
8136 ED42             SBC     HL,BC           ;# OF BYTES TO MOVE
8138 E5               PUSH    HL              ;TRANSFER TO BC
8139 C1               POP     BC              ;BYTE COUNT
813A E1               POP     HL              ;RESTORE SOURCE
813B EDB0             LDIR                    ;MOVE MESSAGE DOWN
813D 21F592           LD      HL,ENDM         ;END OF MEMORY
8140 B7               OR      A
8141 ED52             SBC     HL,DE           ;FIND # OF BYTES REMAINING
8143 E5               PUSH    HL              ;TRANSFER TO BC
8144 C1               POP     BC
8145 3EFF             LD      A,0FFH          ;-1
8147 CD8385           CALL    FILLCH          ;FILL REMAINING WITH -1S
814A CDC782           CALL    FNDMSG          ;FIND FIRST -1
              ; HL POINTS TO MESSAGE AREA, TOP OF STACK HOLDS MESSAGE #
814D F1       DEF035  POP     AF              ;GET MESSAGE #
814E 77               LD      (HL),A          ;STORE IN MESSAGE AREA
814F 23               INC     HL              ;POINT TO FIRST TEXT CHAR POS
8150 0600             LD      B,0             ;INITIALIZE COUNT OF CHARS
8152 CDF384   DEF040  CALL    INPUTW            ;GET NEXT CHARACTER
8155 FE02             CP      ENTER             ;TEST FOR ENTER CHAR
8157 CA3780           JP      Z,MOR015          ;GO IF ENTER
815A 77               LD      (HL),A            ;STORE IN MESSAGE AREA
815B 23               INC     HL                ;BUMP POINTER
815C CDBE83           CALL    DISCHR            ;DISPLAY
815F 10F1             DJNZ    DEF040            ;GO IF NOT 256 CHARS
8161 2AE985           LD      HL,(CURCUR)     ;GET CURRENT CURSOR
```

```
8164 22EB85               LD     (LSTCUR),HL      ;SAVE
8167 21EB87               LD     HL,MSG12         ;"256" CHARACTERS MESSAGE
816A 01C03F               LD     BC,LINE15
816D CD6D83               CALL   DSPMES           ;DISPLAY WARNING MESSAGE
8170 21D007               LD     HL,2000          ;2000 MILLISECONDS
8173 CDC085               CALL   DELAY            ;DELAY 2 SECS
8176 2AEB85               LD     HL,(LSTCUR)      ;GET OLD CURSOR
8179 22E985               LD     (CURCUR),HL      ;RESTORE
817C C33780               JP     MORO15           ;GET NEXT COMMAND
817F 21AA87       DEF050  LD     HL,MSG10         ;ERROR MESSAGE
8182 01803F               LD     BC,LINE14        ;LINE 14
8185 CD6D83               CALL   DSPMES           ;DISPLAY ERROR MESSAGE
8188 21D007               LD     HL,2000          ;2000 MILLISECONDS
818B CDC085               CALL   DELAY            ;DELAY 2 SECS
818E C3F180               JP     DEF005           ;TRY AGAIN
                  ;
                  ;*******************SET SPEED ROUTINE*******************
                  ;
8191 2AE985       SPEED   LD     HL,(CURCUR)      ;GET CURRENT CURSOR
8194 22EB85               LD     (LSTCUR),HL      ;SAVE
8197 CDFB83       SPE005  CALL   CLRCOM           ;CLEAR COMMUNICATION AREA
819A 21CB86               LD     HL,MSG4          ;SPEED MESSAGE
819D 01403F               LD     BC,LINE13        ;LINE 13
81A0 CD6D83               CALL   DSPMES           ;DISPLAY SPEED MESSAGE
81A3 0602                 LD     B,2              ;2 CHARS
81A5 CD0D84               CALL   INPUTS           ;GET CHARACTER STRING
81A8 2035                 JR     NZ,SPE020        ;GO IF GT 2 CHARACTERS
81AA CD9085               CALL   DECBIN           ;CONVERT TO BINARY
81AD 2030                 JR     NZ,SPE020        ;GO IF ERROR
81AF 7D                   LD     A,L              ;GET SPEED 0-99
81B0 FE03                 CP     3                ;TEST FOR 3 WPM
81B2 FADF81               JP     M,SPE020         ;GO IF LT 3 WPM
81B5 FE3D                 CP     61               ;TEST FOR 60 WPM
81B7 F2DF81               JP     P,SPE020         ;GO IF GT 60 WPM
81BA 21C003               LD     HL,SPEEDF        ;1200/WPM=DOTO TIME
81BD 4F                   LD     C,A              ;WPM LS BYTE
81BE 0600                 LD     B,0              ;NOW IN BC
81C0 11FFFF               LD     DE,-1            ;QUOTIENT
81C3 B7           SPE015  OR     A                  ;ZERO C
81C4 ED42                 SBC    HL,BC              ;DIVIDE BY SUCCESSIVE SUB
81C6 13                   INC    DE                 ;BUMP QUOTIENT
81C7 30FA                 JR     NC,SPE015          ;GO IF NOT NEGATIVE
81C9 ED53E585             LD     (DOTO),DE        ;STORE DOT ON TIME
81CD 210000               LD     HL,0
81D0 19                   ADD    HL,DE            ;FIND 3*DOTO
81D1 19                   ADD    HL,DE
81D2 19                   ADD    HL,DE
81D3 22E785               LD     (DASHO),HL       ;STORE DASH ON TIME
81D6 2AEB85               LD     HL,(LSTCUR)      ;GET OLD CURSOR
81D9 22E985               LD     (CURCUR),HL      ;RESTORE
81DC C33780               JP     MORO15           ;BACK TO DRIVER
81DF 212087       SPE020  LD     HL,MSG6          ;ERROR MESSAGE
81E2 01803F               LD     BC,LINE14        ;LINE 14
81E5 CD6D83               CALL   DSPMES           ;DISPLAY ERROR MESSAGE
81E8 21D007               LD     HL,2000          ;2000 MILLISCS
81EB CDC085               CALL   DELAY            ;DELAY 2 SECS
81EE 18A7                 JR     SPE005           ;TRY AGAIN
                  ;
                  ;***********TRANSMIT RANDOM CHARACTERS ROUTINE***********
                  ;
81F0 2AE985       RANDOM  LD     HL,(CURCUR)      ;GET CURRENT CURSOR
81F3 22EB85               LD     (LSTCUR),HL      ;SAVE
81F6 CDFB83               CALL   CLRCOM           ;CLEAR COMMUNICATION AREA
81F9 213F87               LD     HL,MSG7          ;RANDOM MESSAGE
81FC 01403F               LD     BC,LINE13        ;LINE 13 POSITION
81FF CD6D83               CALL   DSPMES           ;DISPLAY RANDOM MESSAGE
8202 ED5F                 LD     A,R              ;GET REFRESH COUNT (RANDOM)
8204 32E485               LD     (SEED+3),A       ;INITIAL SEED
8207 3E20                 LD     A,' '            ;BLANK
8209 32EF85               LD     (LASTR),A        ;INITIALIZE LAST RANDOM CHAR
820C 2AEB85               LD     HL,(LSTCUR)      ;GET OLD CURSOR
820F 22E985               LD     (CURCUR),HL      ;RESTORE
8212 CD4285       RAN010  CALL   RAND               ;GET RANDOM # 0-127
8215 211288               LD     HL,CTAB            ;CHARACTER TABLE
8218 09                   ADD    HL,BC              ;POINT TO CHARACTER
8219 7E                   LD     A,(HL)             ;GET ASCII CHARACTER
821A FE20                 CP     ' '                ;TEST FOR BLANK
821C 2008                 JR     NZ,RAN020          ;GO IF NOT BLANK
821E E5                   PUSH   HL                 ;SAVE HL
```

```
821F 21EF85              LD      HL,LASTR         ;ADDRESS OF LAST CHAR
8222 BE                  CP      (HL)             ;TEST AGAINST LAST
8223 E1                  POP     HL               ;RESTORE HL
8224 28EC                JR      Z,RAN010         ;DON'T SEND 2 BLANKS
8226 CDBE83      RAN020  CALL    DISCHR           ;DISPLAY CHARACTER
8229 CD8183              CALL    LPRINT           ;PRINT IF REQ'D
822C CDE082              CALL    SNDCHR           ;SEND CHAR
822F 32EF85              LD      (LASTR),A        ;SAVE FOR NEXT COMPARE
8232 CD2984              CALL    INPUT            ;TEST FOR CLEAR
8235 18DB                JR      RAN010           ;BACK TO NEXT CHAR
                 ;
                 ;***************TRANSMIT MESSAGE N ROUTINE***************
                 ;
8237 2AE985      XMIT    LD      HL,(CURCUR)     ;GET CURRENT CURSOR
823A 22EB85              LD      (LSTCUR),HL     ;SAVE
823D CDFB83              CALL    CLRCOM          ;CLEAR COMMUNICATION AREA
8240 216A87              LD      HL,MSG8         ;TRANSMIT MESSAGE
8243 C5                  PUSH    BC              ;SAVE MSG#+3*2
8244 01403F              LD      BC,LINE13       ;LINE 13 POSITION
8247 CD6D83              CALL    DSPMES          ;DISPLAY TRANSMIT MESSAGE
                 ; MESSAGE #*2 STILL IN BC
824A C1                  POP     BC              ;RETRIEVE MSG#+3*2
824B CB39                SRL     C               ;MESSAGE #+3 IN BC
824D 79                  LD      A,C             ;MESSAGE #+3 IN A
824E D603                SUB     3               ;MESSAGE # IN A
8250 CDC782              CALL    FNDMSG          ;GET ADDRESS OF MESSAGE
8253 201E                JR      NZ,XMT020       ;GO IF NONE
8255 23                  INC     HL              ;BYPASS #
8256 ED5BEB85            LD      DE,(LSTCUR)     ;GET OLD CURSOR
825A ED53E985            LD      (CURCUR),DE     ;RESTORE
825E 7E          XMT010  LD      A,(HL)           ;GET NEXT CHARACTER
825F FE20                CP      ' '              ;TEST FOR NON-ASCII
8261 FA3780              JP      M,MOR015         ;GO IF END OF MESSAGE
8264 CDBE83              CALL    DISCHR           ;DISPLAY CHARACTER
8267 CD8183              CALL    LPRINT           ;PRINT IF REQ'D
826A CDE082              CALL    SNDCHR           ;SEND CHAR
826D 23                  INC     HL               ;POINT TO NEXT
826E CD2984              CALL    INPUT            ;TEST FOR CLEAR
8271 18EB                JR      XMT010           ;CONTINUE SENDING
8273 219587      XMT020  LD      HL,MSG9         ;ERROR MESSAGE
8276 01803F              LD      BC,LINE14       ;LINE 14 POSITION
8279 CD6D83              CALL    DSPMES          ;DISPLAY ERROR MESSAGE
827C 21E803              LD      HL,1000         ;1000 MILLISECS
827F CDC085              CALL    DELAY           ;DELAY 1 SEC
8282 2AEB85              LD      HL,(LSTCUR)     ;GET OLD CURSOR
8285 22E985              LD      (CURCUR),HL     ;RESTORE
8288 C33780              JP      MOR015          ;BACK TO EXECUTIVE
                 ;
                 ;*******************SET PRINT ROUTINE*******************
                 ;
828B 2AE985      PRINT   LD      HL,(CURCUR)     ;GET CURRENT CURSOR
828E 22EB85              LD      (LSTCUR),HL     ;SAVE
8291 3E01                LD      A,1             ;PRINT FLAG ON
8293 32DE85              LD      (PRINTF),A      ;STORE
8296 CDFB83              CALL    CLRCOM          ;CLEAR COMMUNICATION AREA
8299 21B586              LD      HL,MSG2         ;PRINT MESSAGE
829C 01403F      PRI010  LD      BC,LINE13       ;LINE 13
829F CD6D83              CALL    DSPMES          ;DISPLAY PRINT MESSAGE
82A2 21E803              LD      HL,1000         ;1000 MILLISECONDS
82A5 CDC085              CALL    DELAY           ;DELAY 1 SEC
82A8 2AEB85              LD      HL,(LSTCUR)     ;GET OLD CURSOR
82AB 22E985              LD      (CURCUR),HL     ;RESTORE
82AE C33780              JP      MOR015          ;BACK TO DRIVER
                 ;
                 ; NO PRINT ROUTINE
                 ;
82B1 AF          NOPRNT  XOR     A               ;PRINT FLAG OFF
82B2 32DE85              LD      (PRINTF),A      ;STORE
82B5 32DF85              LD      (LPFTF),A       ;RESET FIRST TIME FLAG
82B8 2AE985              LD      HL,(CURCUR)     ;GET CURRENT CURSOR
82BB 22EB85              LD      (LSTCUR),HL     ;SAVE
82BE CDFB83              CALL    CLRCOM          ;CLEAR COMMUNICATIONS AREA
82C1 21BF86              LD      HL,MSG3         ;NO PRINT MESSAGE
82C4 C39C82              JP      PRI010          ;GO TO DISPLAY, DELAY   -
                 ;
                 ;******************FNDMSG SUBROUTINE*******************
                 ;*   FINDS MESSAGE BY SCANNING MBUF FOR 0-9. 0-9 IS    *
                 ;*   MESSAGE #. ASCII CHARACTERS ARE CHARACTERS TO      *
                 ;*   BE SENT. -1 IS PADDING AT END OF MESSAGE.          *
```

```
                   ;*   ENTRY: (A)=MESSAGE #                               *
                   ;*   EXIT: (HL)=POINTER TO MESSAGE IF Z OR              *
                   ;*              POINTER TO NEXT AVAILABLE IF NZ         *
                   ;*              RESET                                   *
                   ;*   ALL REGISTERS SAVED EXCEPT HL                      *
                   ;
82C7 C5            FNDMSG  PUSH    BC              ;SAVE BC
82C8 CDD782                CALL    FNDSR           ;SEARCH
82CB 2807                  JR      Z,FND020        ;GO IF FOUND
82CD F5                    PUSH    AF              ;SAVE NOT FND FLAG
82CE 3EFF                  LD      A,0FFH          ;FOR FIRST AVAILABLE
82D0 CDD782                CALL    FNDSR           ;SEARCH
82D3 F1                    POP     AF              ;GET FLAG
82D4 C1            FND020  POP     BC              ;RESTORE BC
82D5 2B                    DEC     HL              ;ADJUST HL
82D6 C9                    RET                     ;RETURN
82D7 21EA88        FNDSR   LD      HL,MBUF         ;START OF MSG BUFFER
82DA 010B0A                LD      BC,2571         ;SIZE OF MBUF
82DD EDB1                  CPIR                    ;COMPARE
82DF C9                    RET                     ;RETURN
                   ;
                   ;***************SEND CHARACTER SUBROUTINE***************
                   ;*   SENDS A SINGLE CHARACTER AT CURRENT SPEED BY OUT-  *
                   ;*   PUTTING A 500 HERTZ TONE TO CASSETTE.              *
                   ;*   ENTRY: (A)=ASCII CHARACTER                         *
                   ;*   ALL REGISTERS SAVED.                               *
                   ;
82E0 F5            SNDCHR  PUSH    AF              ;SAVE REGISTERS
82E1 C5                    PUSH    BC
82E2 E5                    PUSH    HL
82E3 DDE5                  PUSH    IX
82E5 213D88                LD      HL,CDTAB+CDTABS-1  ;ADDRESS OF CODE TAB END
82E8 012C00                LD      BC,CDTABS       ;SIZE OF CODE TABLE
82EB EDB9                  CPDR                    ;SEARCH CODE TABLE
                   ; MUST BE FOUND!
82ED C5                    PUSH    BC
82EE DDE1                  POP     IX              ;INDEX IN HL
82F0 DD29                  ADD     IX,IX           ;2*INDEX IN HL
82F2 019288                LD      BC,TTAB         ;TIMING TABLE ADDRESS
82F5 DD09                  ADD     IX,BC           ;POINT TO TIMING CHAR
82F7 DD6E00                LD      L,(IX)          ;GET TIMING CHAR
82FA DD6601                LD      H,(IX+1)
82FD E5                    PUSH    HL              ;SAVE HL
82FE E1            SND010  POP     HL                ;GET CURRENT HL
82FF 7D                    LD      A,L               ;GET NEXT 2 BITS
8300 CB3C                  SRL     H                 ;ALIGN FOR NEXT NIBBLE
8302 CB1D                  RR      L
8304 CB3C                  SRL     H
8306 CB1D                  RR      L
8308 CBFC                  SET     7,H               ;SUBTLETY HERE-TERMINATOR!
830A CBF4                  SET     6,H
830C E5                    PUSH    HL                ;SAVE CURRENT HL
830D E603                  AND     3
830F 2816                  JR      Z,DOT             ;GO IF DOT
8311 FE01                  CP      1                 ;DASH=1
8313 2820                  JR      Z,DASH            ;GO IF DASH
8315 FE02                  CP      2                 ;DOT SPACE=2
8317 2814                  JR      Z,SPACE           ;GO IF SPACE
8319 2AE585                LD      HL,(DOTO)       ;INTERCHAR SPACE=3
831C 29                    ADD     HL,HL           ;2*DOTO+PREVIOUS DOTO
831D CD5683                CALL    COFF            ;DELAY 2 DOT TIMES
8320 E1                    POP     HL              ;RESET STACK
8321 DDE1                  POP     IX              ;RESTORE REGISTERS
8323 E1                    POP     HL
8324 C1                    POP     BC
8325 F1                    POP     AF
8326 C9                    RET                     ;RETURN
                   ;
                   ; DOT HERE
                   ;
8327 2AE585        DOT     LD      HL,(DOTO)         ;DOT ON TIME
832A CD3D83                CALL    CON               ;TOGGLE CASSETTE OUT
832D 2AE585        SPACE   LD      HL,(DOTO)         ;DOT OFF TIME=DOT ON TIME
8330 CD5683                CALL    COFF              ;DELAY
8333 18C9                  JR      SND010            ;RETURN FOR NEXT NIBBLE
                   ;
                   ; DASH HERE
                   ;
8335 2AE785        DASH    LD      HL,(DASHO)        ;DASH ON TIME
```

```
8338 CD3D83   04510          CALL    CON               ;TOGGLE CASSETTE OUT
833B 18F0     04520          JR      SPACE             ;DASH OFF TIME=DOT OFF TIME
              04530 ;
              04540 ; ON HERE - GENERATE 500 HERTZ TONE
              04550 ;
833D 3E01     04560 CON      LD      A,01              ;ON
833F 2B       04570          DEC     HL                ;ADJUST FOR "JR C"
8340 01FFFF   04580          LD      BC,-1             ;DECREMENT
8343 EE03     04590 ON010    XOR     3                   ;TOGGLE
8345 D3FF     04600          OUT     (0FFH),A            ;OUTPUT TO CASSETTE LATCH
8347 E5       04610          PUSH    HL                  ;SAVE COUNT
8348 210100   04620          LD      HL,1                ;FOR 1 MS
834B CDC085   04630          CALL    DELAY               ;DELAY 1 MS
834E CD2984   04640          CALL    INPUT               ;GET POSSIBLE CHARACTER
8351 E1       04650          POP     HL                  ;GET COUNT
8352 09       04660          ADD     HL,BC               ;DECREMENT COUNT
8353 38EE     04670          JR      C,ON010             ;GO IF NOT -1
8355 C9       04680          RET                       ;RETURN
              04690 ;
              04700 ; OFF HERE
              04710 ;
8356 EE00     04720 COFF     XOR     0
8358 2B       04730          DEC     HL                ;ADJUST FOR "JR C"
8359 01FFFF   04740          LD      BC,-1             ;DECREMENT
835C D3FF     04750          OUT     (0FFH),A            ;OUTPUT TO CASSETTE LATCH
835E E5       04760 OFF010   PUSH    HL                  ;SAVE COUNT
835F 210100   04770          LD      HL,1                ;FOR 1 MS
8362 CDC085   04780          CALL    DELAY               ;DELAY 1 MS
8365 CD2984   04790          CALL    INPUT               ;GET POSSIBLE CHARACTER
8368 E1       04800          POP     HL                  ;GET COUNT
8369 09       04810          ADD     HL,BC               ;DECREMENT COUNT
836A 38F2     04820          JR      C,OFF010            ;GO IF NOT -1
836C C9       04830          RET                       ;RETURN
              04840 ;
              04850 ;**************DISPLAY MESSAGE AT LOCATION N*************
              04860 ;*   DISPLAYS MESSAGE AT GIVEN SCREEN POSITION. TER-     *
              04870 ;*   MINATES ON NULL (ZERO).                             *
              04880 ;*   ENTRY: (HL)=MESSAGE LOCATION                        *
              04890 ;*          (BC)=SCREEN POSITION                         *
              04900 ;*   ALL REGISTERS SAVED.                                *
              04910 ;
836D F5       04920 DSPMES   PUSH    AF                ;SAVE REGISTERS
836E C5       04930          PUSH    BC
836F E5       04940          PUSH    HL
8370 7E       04950 DSP005   LD      A,(HL)              ;GET MESSAGE CHAR
8371 B7       04960          OR      A                   ;TEST FOR 0
8372 2809     04970          JR      Z,DSP010            ;RETURN IF DONE
8374 02       04980          LD      (BC),A              ;STORE CHARACTER
8375 03       04990          INC     BC                  ;BUMP SCREEN POINTER
8376 23       05000          INC     HL                  ;BUMP MESSAGE POINTER
8377 ED43E985 05010          LD      (CURCUR),BC         ;SAVE POINTER
837B 18F3     05020          JR      DSP005              ;CONTINUE
837D E1       05030 DSP010   POP     HL                ;RESTORE REGISTERS
837E C1       05040          POP     BC
837F F1       05050          POP     AF
8380 C9       05060          RET                       ;RETURN
              05070 ;
              05080 ;*******************LPRINT SUBROUTINE*******************
              05090 ;*  OUTPUTS CHARACTER TO SYSTEM LINE PRINTER IF PRINT    *
              05100 ;*  FLAG IS SET.                                         *
              05110 ;*  ENTRY: (A)=ASCII CHARACTER                           *
              05120 ;*  ALL REGISTERS SAVED.                                 *
              05130 ;
8381 C5       05140 LPRINT   PUSH    BC                ;SAVE REGISTERS
8382 E5       05150          PUSH    HL
8383 47       05160          LD      B,A               ;SAVE CHARACTER
8384 3ADE85   05170          LD      A,(PRINTF)        ;GET PRINT FLAG
8387 B7       05180          OR      A                 ;TEST
8388 78       05190          LD      A,B               ;RESTORE A FOR POSSIBLE RTN
8389 2821     05200          JR      Z,LPR090          ;RETURN IF NOT SET
838B 3ADF85   05210          LD      A,(LPFTF)         ;GET LINE PRINTER 1ST TIME
838E B7       05220          OR      A                 ;TEST
838F 2007     05230          JR      NZ,LPR010         ;GO IF NOT FIRST TIME
8391 32E085   05240          LD      (CHARCT),A        ;INITIALIZE CHAR COUNT
8394 3C       05250          INC     A                 ;1 TO A
8395 32DF85   05260          LD      (LPFTF),A         ;SET FIRST TIME FLAG
```

```
8398 3AE085   LPR010  LD      A,(CHARCT)      ;GET CHARACTER COUNT
839B E61F             AND     1FH             ;GET 0-31 COUNT
839D 2005             JR      NZ,LPR020       ;GO IF NOT 32ND
839F 3E0D             LD      A,0DH           ;CARRIAGE RETURN
83A1 CDAF83           CALL    LPSTAT          ;TEST STATUS AND OUTPUT
83A4 78       LPR020  LD      A,B             ;RESTORE CHARACTER
83A5 CDAF83           CALL    LPSTAT          ;TEST BUSY AND OUTPUT
83A8 21E085           LD      HL,CHARCT       ;CHARACTER COUNT
83AB 34               INC     (HL)            ;BUMP CHARACTER COUNT
83AC E1       LPR090  POP     HL              ;RESTORE REGISTERS
83AD C1               POP     BC
83AE C9               RET                     ;RETURN
              ;
              ; LINE PRINTER STATUS AND PRINT CHARACTER SUBROUTINE
              ;
83AF F5       LPSTAT  PUSH    AF              ;SAVE CHAR
83B0 3AE837   LPS010  LD      A,(37E8H)         ;GET STATUS
83B3 E6F0             AND     0F0H              ;MASK OUT GARBAGE BITS
83B5 FE30             CP      30H               ;TEST FOR BUSY
83B7 20F7             JR      NZ,LPS010         ;GO IF BUSY
83B9 F1               POP     AF              ;RESTORE CHAR
83BA 32E837           LD      (37E8H),A       ;OUTPUT
83BD C9               RET                     ;RETURN
              ;
              ;**************DISPLAY CHARACTER SUBROUTINE**************
              ;*    OUTPUTS ONE CHARACTER TO CURRENT CURSOR POSITION    *
              ;*    ON SCREEN. MOVES CURSOR TO NEXT POSITION UNLESS     *
              ;*    LAST CHARACTER POSITION OF LINE 11. IF LATTER,      *
              ;*    SCROLLS UP FIRST.                                   *
              ;*    ENTRY: (CURCUR)=CURRENT CURSOR POSITION             *
              ;*           (A)=CHARACTER TO BE OUTPUT                   *
              ;*    ALL REGISTERS SAVED.                                *
              ;
83BE C5       DISCHR  PUSH    BC              ;SAVE REGISTERS
83BF E5               PUSH    HL
83C0 2AE985           LD      HL,(CURCUR)     ;GET CHARACTER POSITION
83C3 77               LD      (HL),A          ;STORE CHARACTER
83C4 01FF3E           LD      BC,3C00H+767    ;LAST CP OF LINE 11
83C7 23               INC     HL              ;BUMP CURSOR
83C8 22E985           LD      (CURCUR),HL     ;STORE
83CB B7               OR      A               ;RESET CARRY
83CC ED42             SBC     HL,BC           ;TEST FOR LAST
83CE 2003             JR      NZ,DIS010       ;RETURN IF NO SCROLL
83D0 CDD683           CALL    SCROLL          ;SCROLL UP
83D3 E1       DIS010  POP     HL              ;RESTORE REGISTERS
83D4 C1               POP     BC
83D5 C9               RET                     ;RETURN
              ;
              ;***************SCROLL SCREEN SUBROUTINE****************
              ;*    SCROLLS LINES 1-11 UP TO LINES 0-10. FILLS LINE 11 *
              ;*    WITH BLANKS.                                       *
              ;*    ENTRY: NO PARAMETERS                               *
              ;*    ALL REGISTERS SAVED.                               *
              ;
83D6 F5       SCROLL  PUSH    AF              ;SAVE REGISTERS
83D7 C5               PUSH    BC
83D8 D5               PUSH    DE
83D9 E5               PUSH    HL
83DA 11003C           LD      DE,SCREEN       ;START OF SCREEN
83DD 21403C           LD      HL,LINE1        ;LINE 1
83E0 010003           LD      BC,1024-256     ;# TO MOVE
83E3 EDB0             LDIR                    ;MOVE EM
83E5 11C03E           LD      DE,LINE11       ;START OF LINE 11
83E8 3E20             LD      A,' '           ;SPACE
83EA 014000           LD      BC,64           ;# TO FILL
83ED CD8385           CALL    FILLCH          ;FILL LINE
83F0 21C03E           LD      HL,LINE11       ;START OF LINE 11
83F3 22E985           LD      (CURCUR),HL     ;RESET
83F6 E1               POP     HL              ;RESTORE REGISTERS
83F7 D1               POP     DE
83F8 C1               POP     BC
83F9 F1               POP     AF
83FA C9               RET                     ;RETURN
              ;
              ;**********CLEAR COMMUNICATION AREA SUBROUTINE***********
              ;*    CLEARS SYSTEM COMMUNICATION AREA                   *
              ;*    ENTRY: NO PARAMETERS                               *
              ;*    ALL REGISTERS SAVED.                               *
              ;
```

```
83FB F5              CLRCOM  PUSH   AF                 ;SAVE REGISTERS
83FC C5                      PUSH   BC
83FD D5                      PUSH   DE
83FE 3E20                    LD     A,' '              ;BLANK
8400 11403F                  LD     DE,LINE13          ;START OF TEXT AREA
8403 01C000                  LD     BC,192             ;3 LINES WORTH
8406 CD8385                  CALL   FILLCH             :FILL WITH BLANKS
8409 D1                      POP    DE                 ;RESTORE REGISTERS
840A C1                      POP    BC
840B F1                      POP    AF
840C C9                      RET                       ;RETURN
                     ;
                     ;***************INPUT STRING SUBROUTINE*****************
                     ;*   INPUTS STRING OF CHARACTERS AT CURRENT COMMUNICA-  *
                     ;*   TION AREA. TERMINATED BY ENTER.                    *
                     ;*   ENTRY: (B)=MAXIMUM NUMBER                          *
                     ;*   (CURCUR)=CURRENT CURSOR POSITION                   *
                     ;*   EXIT:  (B)=ACTUAL NUMBER INPUT                     *
                     ;*          (HL)=FIRST CHARACTER LOCATION               *
                     ;*          NZ IF GT MAXIMUM NUMBER                     *
                     ;*          Z IF LE MAXIMUM NUMBER                      *
                     ;*   ALL REGISTERS SAVED EXCEPT HL,BC,A                 *
                     ;
840D 2AE985          INPUTS  LD     HL,(CURCUR)        ;CURRENT CURSOR POSITION
8410 E5                      PUSH   HL                 ;SAVE
8411 04                      INC    B                  ;BUMP MAXIMUM
8412 0E00                    LD     C,0                ;INITIALIZE COUNT OF CHARS
8414 C5              INS010  PUSH   BC                 ;SAVE COUNTS
8415 CDF384                  CALL   INPUTW               ;GET CHARACTER
8418 C1                      POP    BC                   ;RESTORE COUNTS
8419 FE02                    CP     ENTER                ;TEST FOR DONE
841B 2809                    JR     Z,INS030             ;GO IF ENTER
841D 0C                      INC    C                    ;BUMP CHARACTER COUNT
841E CDBE83                  CALL   DISCHR               ;DISPLAY
8421 10F1                    DJNZ   INS010               ;GO IF NOT MAXIMUM
8423 3EFF                    LD     A,0FFH             ;-1 TO A
8425 B7                      OR     A                  ;RESET Z FLAG
8426 E1              INS030  POP    HL                 ;RETRIEVE START
8427 41                      LD     B,C                ;GET CHARACTER COUNT
8428 C9                      RET                       ;RETURN
                     ;
                     ;***************KEYBOARD INPUT SUBROUTINE***************
                     ;*   IF DEBOUNCE DELAY LESS THAN ELAPSED TIME, SCANS    *
                     ;*   KEYBOARD AND STORES POSSIBLE INPUT CHARACTER IN    *
                     ;*   CIRCULAR INPUT BUFFER.                             *
                     ;*   ENTRY: NO PARAMETERS                               *
                     ;*   EXIT: NO PARAMETERS                                *
                     ;*   ALL REGISTERS SAVED                                *
                     ;*   CLEAR CHARACTER CAUSES RESTART AT MOR015H, SP RESET*
                     ;
8429 F5              INPUT   PUSH   AF                 ;SAVE REGISTERS
842A 3A7F38                  LD     A,(387FH)          ;ALL IN ONE SWELL FOOP
842D B7                      OR     A                  ;TEST FOR ANY KEY
842E CAB984                  JP     Z,INP065           ;GO IF NONE
8431 C5                      PUSH   BC
8432 E5                      PUSH   HL
8433 2AED85                  LD     HL,(TSLC)          ;GET TIME SINCE LAST CHARACTER
8436 016400                  LD     BC,DBDEL           ;MINIMUM DELAY
8439 B7                      OR     A                  ;RESET CARRY
843A ED42                    SBC    HL,BC              ;COMPARE
843C 3879                    JR     C,INP060           ;GO IF LT DEBOUNCE DELAY
843E 210138                  LD     HL,3801H           ;ROW 0 ADDRESS
8441 7E              INP010  LD     A,(HL)               ;GET ROW VALUE
8442 B7                      OR     A                    ;TEST FOR NON-ZERO
8443 2007                    JR     NZ,INP020            ;GO IF INPUT
8445 CB25                    SLA    L                    ;SHIFT ROW ADRESS
8447 F24184                  JP     P,INP010             ;MORE TO GO
844A 186B                    JR     INP060             ;RETURN
                     ; CONVERT ROW, COLUMN TO INDEX
844C 4F              INP020  LD     C,A                ;ROW VALUE
844D AF                      XOR    A                  ;ZERO A
844E CB3D            INP025  SRL    L                    ;SHIFT ADDRESS
8450 3804                    JR     C,INP035             ;GO IF DONE
8452 C608                    ADD    A,8                  ;ROW*8
8454 18F8                    JR     INP025               ;CONTINUE
8456 06FF            INP035  LD     B,0FFH             ;COLUMN COUNT
8458 04              INP040  INC    B                    ;BUMP COUNT
8459 CB39                    SRL    C                    ;SHIFT ROW VALUE
845B 30FB                    JR     NC,INP040            ;CONTINUE UNTIL 1 BIT
```

```
845D 80                    ADD     A,B             ;ROW*8+COL
845E 4F                    LD      C,A             ;TRANSFER TO C
            ; FIND TABLE ENTRY
845F 0600                  LD      B,0             ;NOW IN BC
8461 21BB84                LD      HL,KBTAB        ;ADDRESS OF LOOK UP TABLE
8464 09                    ADD     HL,BC           ;POINT TO VALUE
8465 7E                    LD      A,(HL)          ;GET VALUE
8466 B7                    OR      A               ;TEST CHARCTER
8467 284E                  JR      Z,INP060        ;GO IF NOT VALID
8469 F5                    PUSH    AF              ;SAVE CHARACTER
846A 3A8038                LD      A,(3880H)       ;GET SHIFT
846D 0F                    RRCA                    ;ALIGN TO BIT 7
846E 47                    LD      B,A             ;PUT IN B
846F F1                    POP     AF              ;RESTORE CHARACTER
8470 FE2F                  CP      '/'             ;TEST FOR SLASH
8472 2804                  JR      Z,INP052        ;GO IF SLASH
8474 FE2D                  CP      '-'             ;TEST FOR MINUS
8476 2006                  JR      NZ,INP055       ;NEITHER
8478 CB78          INP052  BIT     7,B             ;TEST FOR SHIFT
847A 2802                  JR      Z,INP055        ;GO IF LOWER CASE
847C CBE7                  SET     4,A             ;/ TO ? OR - TO =
847E FE01          INP055  CP      CLEAR           ;TEST FOR CLEAR
8480 2018                  JR      NZ,INP057       ;GO IF NOT CLEAR
8482 315994                LD      SP,TOPS         ;RESET STACK, DROP REGS
8485 2AE985                LD      HL,(CURCUR)     ;GET LAST CURSOR POSN
8488 01403F                LD      BC,LINE13       ;START OF COMM AREA
848B B7                    OR      A               ;CLEAR CARRY
848C ED42                  SBC     HL,BC           ;COMPARE
848E FA9784                JP      M,INP056        ;GO IF IN TEXT AREA
8491 2AEB85                LD      HL,(LSTCUR)     ;GET TEXT AREA PNTR
8494 22E985                LD      (CURCUR),HL     ;RESET CURSOR POSN
8497 C33780        INP056  JP      MORO15          ;RETURN TO EXEC
            ; STORE CHARACTER IN CIRCULAR INPUT BUFFER
849A 2AF285        INP057  LD      HL,(IBUFN)      ;GET POINTER TO NEXT
849D B0                    OR      B               ;MERGE IN SHIFT BIT
849E 77                    LD      (HL),A          ;STORE THIS CHARACTER
849F 23                    INC     HL              ;BUMP TO NEXT SLOT
84A0 22F285                LD      (IBUFN),HL      ;STORE
84A3 01F593                LD      BC,IBUFE        ;END OF INPUT BUFFER
84A6 B7                    OR      A               ;RESET CARRY
84A7 ED42                  SBC     HL,BC           ;TEST FOR END
84A9 2006                  JR      NZ,INP059       ;GO IF NOT END
84AB 21F592                LD      HL,IBUF         ;START OF I BUFFER
84AE 22F285                LD      (IBUFN),HL      ;BACK TO BEGINNING
84B1 210000        INP059  LD      HL,0            ;ZERO HL
84B4 22ED85                LD      (TSLC),HL       ;RESET TIMER
84B7 E1            INP060  POP     HL              ;RESTORE REGISTERS
84B8 C1                    POP     BC
84B9 F1            INP065  POP     AF
84BA C9                    RET                     ;RETURN
            ;
            ; KEYBOARD LOOK UP TABLE HERE FOR LOWER CASE
            ;
84BB 00            KBTAB   DEFB    0               ;ROW 0 - @
84BC 41                    DEFM    'ABCDEFG'
     42 43 44 45 46 47
84C3 48                    DEFM    'HIJKLMNO'      ;ROW 1
     49 4A 4B 4C 4D 4E 4F
84CB 50                    DEFM    'PQRSTUVW'      ;ROW 2
     51 52 53 54 55 56 57
84D3 58                    DEFM    'XYZ'           ;ROW 3
     59 5A
84D6 00                    DEFB    0
84D7 00                    DEFB    0
84D8 00                    DEFB    0
84D9 00                    DEFB    0
84DA 00                    DEFB    0
84DB 30                    DEFM    '01234567'      ;ROW 4
     31 32 33 34 35 36 37
84E3 38                    DEFM    '89'            ;ROW 5
     39
84E5 00                    DEFB    0               ;COLON
84E6 3B                    DEFB    ';'             ;"ERROR"
84E7 2C                    DEFM    ',-./'
     2D 2E 2F
84EB 02                    DEFB    2               ;ROW 6 - ENTER
84EC 01                    DEFB    1               ;CLEAR
84ED 00                    DEFB    0               ;BREAK
84EE 00                    DEFB    0               ;UP ARROW
```

```
84EF 00        07570          DEFB     0                  ;DOWN ARROW
84F0 00        07580          DEFB     0                  ;LFT ARROW
84F1 00        07590          DEFB     0                  ;RT ARROW
84F2 20        07600          DEFM     ' '

               07610 ;
               07620 ;**********INPUT CHARACTER AND WAIT SUBROUTINE***********
               07630 ;*    INPUTS SINGLE CHARACTER FROM KEYBOARD BY WAITING   *
               07640 ;*    UNTIL CHARACTER PRESENT.                           *
               07650 ;*    ENTRY: NO PARAMETERS                               *
               07660 ;*    EXIT:  (A)=CHARACTER                               *
               07670 ;*    ALL REGISTERS SAVED EXCEPT A                       *
               07680 ;
84F3 E5        07690 INPUTW   PUSH     HL                 ;SAVE REGISTERS
84F4 216500    07700          LD       HL,DBDELP          ;MINIMUM DELAY+1
84F7 CDC085    07710          CALL     DELAY              ;AVOID PREVIOUS CHAR
84FA 216500    07720          LD       HL,DBDELP          ;MINIMUM DELAY + 1
84FD 22ED85    07730          LD       (TSLC),HL          ;INITIALIZE
8500 21F592    07740          LD       HL,IBUF            ;START OF INPUT BUFFER
8503 22F285    07750          LD       (IBUFN),HL         ;RESET NEXT CHAR SLOT PNTR
8506 22F085    07760          LD       (IBUFL),HL         ;RESET LAST CHARACTER SLOT
8509 CD2984    07770 INW010   CALL     INPUT                ;SCAN
850C C5        07780          PUSH     BC
850D CD1585    07790          CALL     GETCHR               ;GET POSSIBLE CHARACTER
8510 C1        07800          POP      BC
8511 28F6      07810          JR       Z,INW010             ;GO IF NOTHING
8513 E1        07820          POP      HL                 ;RESTORE REGISTERS
8514 C9        07830          RET                         ;RETURN
               07840 ;
               07850 ;***************GET CHARACTER SUBROUTINE****************
               07860 ;*    GETS CHARACTER FROM INPUT BUFFER IF THERE IS ONE   *
               07870 ;*    ENTRY: NO PARAMETERS                               *
               07880 ;*    EXIT: (A)=ASCII CHARACTER OR ZEROES IF NONE        *
               07890 ;*          Z IF NONE, NZ IF CHARACTER                   *
               07900 ;*          (B)=80H IF SHIFT, 0 IF NO SHIFT              *
               07910 ;*    ALL REGISTERS SAVED EXCEPT A,BC                    *
               07920 ;
8515 E5        07930 GETCHR   PUSH     HL                 ;SAVE REGISTERS
8516 2AF085    07940          LD       HL,(IBUFL)         ;LAST CHARACTER
8519 ED4BF285  07950          LD       BC,(IBUFN)         ;NEXT SLOT
851D B7        07960          OR       A                  ;RESET CARRY
851E ED42      07970          SBC      HL,BC              ;TEST FOR END
8520 281E      07980          JR       Z,GET070           ;GO IF CAUGHT UP
8522 09        07990          ADD      HL,BC              ;RESTORE IBUFL
8523 7E        08000          LD       A,(HL)             ;GET NEXT CHARACTER
8524 E680      08010          AND      80H                ;MASK IN SHIFT
8526 47        08020          LD       B,A                ;B NOW HAS SHIFT BIT
8527 7E        08030          LD       A,(HL)             ;NEXT CHARACTER
8528 E67F      08040          AND      7FH                ;A NOW HAS CHARACTER
852A C5        08050          PUSH     BC                 ;SAVE SHIFT
852B F5        08060          PUSH     AF                 ;SAVE CHARACTER
852C 23        08070          INC      HL                 ;BUMP LAST PNTR
852D 22F085    08080          LD       (IBUFL),HL         ;SAVE
8530 01F593    08090          LD       BC,IBUFE           ;END OF I BUF
8533 B7        08100          OR       A                  ;RESET CARRY
8534 ED42      08110          SBC      HL,BC              ;TEST FOR END
8536 2006      08120          JR       NZ,GET065          ;GO IF NOT END
8538 21F592    08130          LD       HL,IBUF            ;START OF I BUF
853B 22F085    08140          LD       (IBUFL),HL         ;BACK TO BEGINNING
853E F1        08150 GET065   POP      AF                 ;RESTORE CHARACTER
853F C1        08160          POP      BC                 ;RESTORE SHIFT
8540 E1        08170 GET070   POP      HL                 ;RESTORE ENTRY REGISTER
8541 C9        08180          RET                         ;RETURN
               08190 ;
               08200 ;*B***************RANDOM NUMBER ROUTINE*****************
               08210 ;*    GENERATES A PSEUDO-RANDOM # FROM 0 TO 127.         *
               08220 ;*    ENTRY: NO PARAMETERS                               *
               08230 ;*    EXIT:  (BC)=RANDOM # 0-127                         *
               08240 ;*    ALL REGISTERS SAVED EXCEPT BC.                     *
               08250 ;
8542 F5        08260 RAND     PUSH     AF                 ;SAVE REGISTERS
8543 D5        08270          PUSH     DE
8544 E5        08280          PUSH     HL
8545 ED5BE185  08290          LD       DE,(SEED)          ;GET SEED
8549 2AE385    08300          LD       HL,(SEED+2)
854C 0607      08310          LD       B,7                ;COUNT FOR MULTIPLY BY 128
854E CD6B85    08320 RDM010   CALL     SHIFT                ;SHIFT ONE BIT LEFT
8551 10FB      08330          DJNZ     RDM010               ;SEED*128
```

```
8553 0603               LD      B,3             ;FOR SUBTRACT
8555 CD7185   RDM020    CALL    SUB               ;SUBTRACT ONE
8558 10FB               DJNZ    RDM020            ;SEED*128-3*SEED=SEED*125
855A ED53E185           LD      (SEED),DE       ;STORE NEW SEED
855E 22E385             LD      (SEED+2),HL
8561 3E7F               LD      A,7FH           ;MASK
8563 A2                 AND     D               ;GET 0-127
8564 4F                 LD      C,A             ;NOW IN C
8565 0600               LD      B,0             ;NOW IN BC
8567 E1                 POP     HL              ;RESTORE REGISTERS
8568 D1                 POP     DE
8569 F1                 POP     AF
856A C9                 RET                     ;SHIFT HL
              ;
              ;*******************SHIFT SUBROUTINE******************
              ;*   SHIFTS CONTENTS OF (DE,HL) ONE BIT LEFT           *
              ;*   ENTRY: NUMBER TO BE SHIFTED IN (DE,HL)            *
              ;*   EXIT:  (DE,HL) SHIFTED LEFT ONE BIT, LOGICAL      *
              ;*   ALL REGISTERS SAVED EXCEPT DE,HL.                 *
              ;
856B 29       SHIFT     ADD     HL,HL           ;SHIFT HL
856C EB                 EX      DE,HL           ;GET MS BYTE
856D ED6A               ADC     HL,HL           ;SHIFT MS 2 BYTES AND CARRY
856F EB                 EX      DE,HL           ;NOW ORIGINAL*2
8570 C9                 RET                     ;RETURN
              ;
              ;***************SUBTRACT SEED SUBROUTINE**************
              ;*   SUBTRACTS FOUR BYTES OF SEED FROM (DE,HL).        *
              ;*   ENTRY:  (SEED - SEED+3)=SEED #                    *
              ;*           (DE,HL)=FOUR-BYTE VALUE                   *
              ;*   EXIT:   (DE,HL)=RESULT OF SUBTRACT                *
              ;*   ALL REGISTERS SAVED EXCEPT DE,HL                  *
              ;
8571 C5       SUB       PUSH    BC              ;SAVE REGISTERS
8572 ED4BE385           LD      BC,(SEED+2)     ;GET LS BYTE
8576 B7                 OR      A               ;RESET CARRY
8577 ED42               SBC     HL,BC           ;SUBTRACT LS 2 BYTES
8579 EB                 EX      DE,HL           ;GET MS 2 BYTES
857A ED4BE185           LD      BC,(SEED)       ;GET MS 2 BYTES
857E ED42               SBC     HL,BC           ;SUBTRACT MS 2 BYTES AND CY
8580 EB                 EX      DE,HL           ;NOW ORIGINAL-SEED
8581 C1                 POP     BC              ;RESTORE REGISTERS
8582 C9                 RET                     ;RETURN
              ;
              ;***************FILL CHARACTER SUBROUTINE*************
              ;*   FILLS DESIGNATED AREA WITH GIVEN CHARACTER        *
              ;*   ENTRY:  (A)=CHARACTER                             *
              ;*           (DE)=AREA                                 *
              ;*           (BC)=NUMBER OF BYTES, 1-65525; 0 IS 65536 *
              ;*   ALL REGISTERS SAVED EXCEPT BC,DE                  *
              ;
8583 12       FILLCH    LD      (DE),A            ;FILL CHARACTER
8584 13                 INC     DE                ;BUMP POINTER
8585 0B                 DEC     BC                ;DECREMENT COUNT
8586 F5                 PUSH    AF                ;SAVE FILL CHAR
8587 78                 LD      A,B               ;TEST FOR ZERO
8588 B1                 OR      C
8589 2803               JR      Z,FIL010          ;GO IF DONE
858B F1                 POP     AF                ;RESTORE FILL CHAR
858C 18F5               JR      FILLCH            ;CONTINUE
858E F1       FIL010    POP     AF              ;RESTORE A
858F C9                 RET                     ;RETURN
              ;
              ;********DECIMAL TO BINARY CONVERSION SUBROUTINE********
              ;*   CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING  *
              ;*   DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535. *
              ;*   ENTRY: (HL)=BUFFER CONTAINING ASCII               *
              ;*          (B)=NUMBER OF CHARACTERS                   *
              ;*   EXIT:  (HL)=BINARY # 0-65535                      *
              ;*          NZ IF INVALID ASCII CHARACTER OTHERWISE Z  *
              ;*   ALL REGISTERS SAVED EXCEPT A,HL                   *
              ;
8590 C5       DECBIN    PUSH    BC              ;SAVE REGISTERS
8591 D5                 PUSH    DE
8592 DDE5               PUSH    IX
8594 DD210000           LD      IX,0            ;SET RESULT
8598 DD29     DEC040    ADD     IX,IX             ;INTERMEDIATE*2
859A DDE5               PUSH    IX
859C DD29               ADD     IX,IX             ;*4
```

```
859E DD29      09130           ADD     IX,IX           ;*8
85A0 D1        09140           POP     DE              ;*2
85A1 DD19      09150           ADD     IX,DE           ;*10
85A3 7E        09160           LD      A,(HL)          ;GET CHARACTER
85A4 D630      09170           SUB     30H             ;CONVERT
85A6 FAB685    09180           JP      M,DEC070        ;GO IF LT "0"
85A9 FE0A      09190           CP      10              ;TEST FOR GT "9"
85AB F2B685    09200           JP      P,DEC070        ;GO IF GT "9"
85AE 5F        09210           LD      E,A             ;NOW IN E
85AF 1600      09220           LD      D,0             ;NOW IN DE
85B1 DD19      09230           ADD     IX,DE           ;MERGE
85B3 23        09240           INC     HL
85B4 10E2      09250           DJNZ    DEC040          ;GO IF MORE
85B6 78        09260 DEC070    LD      A,B            ;COUNT TO A
85B7 B7        09270           OR      A              ;SET OR RESET Z FLAG
85B8 DDE5      09280           PUSH    IX             ;RESULT TO HL
85BA E1        09290           POP     HL
85BB DDE1      09300           POP     IX             ;RESTORE REGISTERS
85BD D1        09310           POP     DE
85BE C1        09320           POP     BC
85BF C9        09330           RET                    ;RETURN
               09340 ;
               09350 ;*****************DELAY SUBROUTINE********************
               09360 ;*   DELAYS 1 TO 65536 MILLISECONDS.                   *
               09370 ;*   ENTRY: (HL)=DELAY COUNT IN MILLISECONDS           *
               09380 ;*                   0=65536                           *
               09390 ;*   ALL REGISTERS SAVED.                              *
               09400 ;
85C0 C5        09410 DELAY     PUSH    BC             ;SAVE REGISTERS
85C1 D5        09420           PUSH    DE
85C2 E5        09430           PUSH    HL
85C3 11FFFF    09440           LD      DE,-1          ;-1 FOR DECREMENT
85C6 2B        09450           DEC     HL             ;ADJUST FOR "JP NC"
85C7 0682      09460 DEL010    LD      B,130            ;INNER LOOP TIMING
85C9 10FE      09470 DEL020    DJNZ    DEL020             ;LOOP FOR 1 MILLISEC
85CB E5        09480           PUSH    HL               ;SAVE COUNT
85CC 2AED85    09490           LD      HL,(TSLC)        ;GET TIMER COUNT
85CF 23        09500           INC     HL               ;BUMP
85D0 22ED85    09510           LD      (TSLC),HL        ;SAVE
85D3 E1        09520           POP     HL               ;RESTORE OUTER LOOP CNT
85D4 19        09530           ADD     HL,DE            ;DECREMENT OUTER LOOP CNT
85D5 DAC785    09540           JP      C,DEL010         ;CONTINUE
85D8 E1        09550           POP     HL             ;RESTORE REGISTERS
85D9 D1        09560           POP     DE
85DA C1        09570           POP     BC
85DB C9        09580           RET                    ;RETURN
               09590 ;
               09600 ;*******************WORKING STORAGE*********************
               09610 ;
85DC 0000      09620 TMP1      DEFW    0              ;TEMPORARY STORAGE
85DE 00        09630 PRINTF    DEFB    0              ;PRINTER FLAG:0=OFF,1=ON
85DF 00        09640 LPFTF     DEFB    0              ;LP 1ST TIME FLAG:0=1ST TIME
85E0 00        09650 CHARCT    DEFB    0              ;LP CHARACTER COUNTER
85E1 D204      09660 SEED      DEFW    1234           ;DEFAULT SEED
85E3 2E16      09670           DEFW    5678
85E5 9001      09680 DOTO      DEFW    400            ;DOT ON TIME (3 WPM DEFAULT)
85E7 B004      09690 DASHO     DEFW    1200           ;DASH ON TIME (3 WPM DEFLT)
85E9 003C      09700 CURCUR    DEFW    3C00H          ;CURRENT CURSOR POSITION
85EB 003C      09710 LSTCUR    DEFW    3C00H          ;LAST CURSOR POSITION
85ED 0000      09720 TSLC      DEFW    0              ;TIME IN MS SINCE LAST CHAR
85EF 20        09730 LASTR     DEFB    ' '            ;LAST RANDOM CHARACTER SENT
85F0 F592      09740 IBUFL     DEFW    IBUF           ;POINTER TO LAST IBUF SLOT
85F2 F592      09750 IBUFN     DEFW    IBUF           ;POINTER TO NEXT IBUF SLOT
               09760 ;
               09770 ;******************SYSTEM MESSAGES**********************
               09780 ;
85F4 20        09790 MSG1      DEFM    '                        ***MORG***
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 2A 2A 2A 4D 4F
     52 47 2A 2A 2A 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20
8634 43        09800           DEFM    'CHAR=SEND CHARACTER  SHIFT 0-9=SEND MSG N'
     48 41 52 3D 53 45 4E 44
     20 43 48 41 52 41 43 54
     45 52 20 20 53 48 49 46
```

```
     54 20 30 2D 39 3D 53 45
     4E 44 20 4D 53 47 20 4E

865D 20        09810          DEFM     ' SHIFT R=SEND RANDOM  SHIFT D=DEFINE MS'
     20 53 48 49 46 54 20 52
     3D 53 45 4E 44 20 52 41
     4E 44 4F 4D 20 20 53 48
     49 46 54 20 44 3D 44 45
     46 49 4E 45 20 4D 53
8685 47        09820          DEFM     'G   SHIFT S=DEFINE SPEED  SHIFT P,N=PRINT'
     20 20 20 53 48 49 46 54
     20 53 3D 44 45 46 49 4E
     45 20 53 50 45 45 44 20
     20 53 48 49 46 54 20 50
     2C 4E 3D 50 52 49 4E 54

86AE 20        09830          DEFM     ' OR NO'
     4F 52 20 4E 4F
86B4 00        09840          DEFB     0
86B5 50        09850 MSG2     DEFM     'PRINT SET'
     52 49 4E 54 20 53 45 54

86BE 00        09860          DEFB     0
86BF 50        09870 MSG3     DEFM     'PRINT RESET'
     52 49 4E 54 20 52 45 53
     45 54
86CA 00        09880          DEFB     0
86CB 53        09890 MSG4     DEFM     'SET SPEED MODE. ENTER SPEED 3 TO 60 WPM: '
     45 54 20 53 50 45 45 44
     20 4D 4F 44 45 2E 20 45
     4E 54 45 52 20 53 50 45
     45 44 20 33 20 54 4F 20
     36 30 20 57 50 4D 3A 20

86F4 00        09900          DEFB     0
86F5 44        09910 MSG5     DEFM     'DEFINE MESSAGE MODE. ENTER MESSAGE # 0-9: '
     45 46 49 4E 45 20 4D 45
     53 53 41 47 45 20 4D 4F
     44 45 2E 20 45 4E 54 45
     52 20 4D 45 53 53 41 47
     45 20 23 20 30 2D 39 3A
     20
871F 00        09920          DEFB     0
8720 49        09930 MSG6     DEFM     'INVALID SPEED. MUST BE 3 TO 60'
     4E 56 41 4C 49 44 20 53
     50 45 45 44 2E 20 4D 55
     53 54 20 42 45 20 33 20
     54 4F 20 36 30
873E 00        09940          DEFB     0
873F 52        09950 MSG7     DEFM     'RANDOM CHARACTER MODE. PRESS CLEAR TO STOP'
     41 4E 44 4F 4D 20 43 48
     41 52 41 43 54 45 52 20
     4D 4F 44 45 2E 20 50 52
     45 53 53 20 43 4C 45 41
     52 20 54 4F 20 53 54 4F
     50
8769 00        09960          DEFB     0
876A 54        09970 MSG8     DEFM     'TRANSMIT MESSAGE MODE. PRESS CLEAR TO STOP'
     52 41 4E 53 4D 49 54 20
     4D 45 53 53 41 47 45 20
     4D 4F 44 45 2E 20 50 52
     45 53 53 20 43 4C 45 41
     52 20 54 4F 20 53 54 4F
     50
8794 00        09980          DEFB     0
8795 4E        09990 MSG9     DEFM     'NO MESSAGE BY THAT #'
     4F 20 4D 45 53 53 41 47
     45 20 42 59 20 54 48 41
     54 20 23
87A9 00        10000          DEFB     0
87AA 49        10010 MSG10    DEFM     'INVALID MESSAGE #. MUST BE 0-9'
     4E 56 41 4C 49 44 20 4D
     45 53 53 41 47 45 20 23
     2E 20 4D 55 53 54 20 42
     45 20 30 2D 39
87C8 00        10020          DEFB     0
87C9 45        10030 MSG11    DEFM     'ENTER MESSAGE. TERMINATE BY ENTER'
     4E 54 45 52 20 4D 45 53
     53 41 47 45 2E 20 54 45
```

```
     52 4D 49 4E 41 54 45 20
     42 59 20 45 4E 54 45 52

87EA 00         10040           DEFB     0
87EB 4D         10050 MSG12     DEFM     'MORE THAN 256 CHARACTERS. 256 ACCEPTED'
     4F 52 45 20 54 48 41 4E
     20 32 35 36 20 43 48 41
     52 41 43 54 45 52 53 2E
     20 32 35 36 20 41 43 43
     45 50 54 45 44
8811 00         10060           DEFB     0
                10070 ;
                10080 ;******************CTAB CHARACTER TABLE*****************
                10090 ;*   TABLE OF CHARACTERS TO BE SENT IN RANDOM MODE.      *
                10100 ;*   DISTRIBUTION DOES NOT CORRESPOND TO THAT IN NOR-    *
                10110 ;*   MAL TEXT. SPACE CHARACTER NOMINALLY EVERY 5TH       *
                10120 ;*   CHARACTER.                                          *
                10130 ;
8812 41         10140 CTAB      DEFM     'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.,?/- =;'
     42 43 44 45 46 47 48 49
     4A 4B 4C 4D 4E 4F 50 51
     52 53 54 55 56 57 58 59
     5A 30 31 32 33 34 35 36
     37 38 39 2E 2C 3F 2F 2D
     20 3D 3B
883E 30         10150           DEFM     '0123456789.,?/- =;ABCDEFGHIJKLMNOPQRSTUVWXYZ'
     31 32 33 34 35 36 37 38
     39 2E 2C 3F 2F 2D 20 3D
     3B 41 42 43 44 45 46 47
     48 49 4A 4B 4C 4D 4E 4F
     50 51 52 53 54 55 56 57
     58 59 5A
886A 20         10160           DEFM     '                   '
     20 20 20 20 20 20 20 20
     20 20 20 20 20 20 20 20
     20
887C 41         10170           DEFM     'ABCDEFGHIJKLMNOPRSTUVY'
     42 43 44 45 46 47 48 49
     4A 4B 4C 4D 4E 4F 50 52
     53 54 55 56 59
                10180 ;
                10190 ;*******************CDTAB CODE TABLE*******************
                10200 ;*   TABLE OF VALID ASCII CHARACTERS TO BE TRANSMITTED. *
                10210 ;*   INDEX TO CHARACTER USED TO OBTAIN TIMING CODES     *
                10220 ;*   FROM TTAB.                                         *
                10230 ;
8812            10240 CDTAB     EQU      CTAB              ;SAME DATA
002C            10250 CDTABS    EQU      44                ;SIZE OF DATA
                10260 ;
                10270 ;*******************TTAB TIMING TABLE******************
                10280 ;*   TABLE OF TIMING CODES FOR CHARACTERS IN CDTAB.     *
                10290 ;*   POSITION IN THIS TABLE CORRESPONDS TO POSITION IN  *
                10300 ;*   CDTAB. EACH ENTRY IS 2 BYTES LONG.                 *
                10310 ;*****************************************************
                10320 ;
8892 3400       10330 TTAB      DEFW     34H               ;A
8894 0103       10340           DEFW     301H              ;B
8896 1103       10350           DEFW     311H              ;C
8898 C100       10360           DEFW     0C1H              ;D
889A 0C00       10370           DEFW     0CH               ;E
889C 1003       10380           DEFW     310H              ;F
889E C500       10390           DEFW     0C5H              ;G
88A0 0003       10400           DEFW     300H              ;H
88A2 3000       10410           DEFW     30H               ;I
88A4 5403       10420           DEFW     354H              ;J
88A6 D100       10430           DEFW     0D1H              ;K
88A8 0403       10440           DEFW     304H              ;L
88AA 3500       10450           DEFW     35H               ;M
88AC 3100       10460           DEFW     31H               ;N
88AE D500       10470           DEFW     0D5H              ;O
88B0 1403       10480           DEFW     314H              ;P
88B2 4503       10490           DEFW     345H              ;Q
88B4 C400       10500           DEFW     0C4H              ;R
88B6 C000       10510           DEFW     0C0H              ;S
88B8 0D00       10520           DEFW     0DH               ;T
88BA D000       10530           DEFW     0D0H              ;U
88BC 4003       10540           DEFW     340H              ;V
88BE D400       10550           DEFW     0D4H              ;W
88C0 4103       10560           DEFW     341H              ;X
```

```
88C2 5103            DEFW    351H          ;Y
88C4 0503            DEFW    305H          ;Z
88C6 550D            DEFW    0D55H         ;0
88C8 540D            DEFW    0D54H         ;1
88CA 500D            DEFW    0D50H         ;2
88CC 400D            DEFW    0D40H         ;3
88CE 000D            DEFW    0D00H         ;4
88D0 000C            DEFW    0C00H         ;5
88D2 010C            DEFW    0C01H         ;6
88D4 050C            DEFW    0C05H         ;7
88D6 150C            DEFW    0C15H         ;8
88D8 550C            DEFW    0C55H         ;9
88DA 4434            DEFW    3444H         ;.
88DC 0535            DEFW    3505H         ;,
88DE 5030            DEFW    3050H         ;?
88E0 410C            DEFW    0C41H         ;/
88E2 0134            DEFW    3401H         ;-(-....-)
88E4 AA03            DEFW    3AAH          ;BLANK
88E6 440C            DEFW    0C44H         ;=(.-.-.)
88E8 0000            DEFW    0             ;ERROR(........)
               ;
               ;********************MESSAGE BUFFER*********************
               ;*   ARBITRARILY SET AT 2560 BYTES (256 BYTES PER MSG   *
               ;*   PLUS MSG# PLUS 1 TERMINATOR).                      *
               ;
88EA           MBUF     EQU     $
92F5           ENDM     EQU     $+2571
               ;
               ;*********************INPUT BUFFER**********************
               ;*   CIRCULAR INPUT BUFFER OF 256 BYTES                 *
               ;
92F5           IBUF     EQU     ENDM
93F5           IBUFE    EQU     IBUF+256
9459           TOPS     EQU     IBUFE+100      ;TOP OF STACK
8000                    END     START
00000 TOTAL ERRORS
```

BTAB	80CD	CDTAB	8812	CDTABS	002C	CHARCT	85E0	CLEAR	0001
CLRCOM	83FB	COFF	8356	CON	833D	CTAB	8812	CURCUR	85E9
DASH	8335	DASHO	85E7	DBDEL	0064	DBDELP	0065	DEC040	8598
DEC070	85B6	DECBIN	8590	DEF005	80F1	DEF025	8126	DEF030	812F
DEF035	814D	DEF040	8152	DEF050	817F	DEFINE	80EB	DEL010	85C7
DEL020	85C9	DELAY	85C0	DIS010	83D3	DISCHR	83BE	DOT	8327
DOTO	85E5	DSP005	8370	DSP010	837D	DSPMES	836D	ENDM	92F5
ENTER	0002	FIL010	858E	FILLCH	8583	FND020	82D4	FNDMSG	82C7
FNDSR	82D7	FTAB	80BE	FTABS	000F	GET065	853E	GET070	8540
GETCHR	8515	IBUF	92F5	IBUFE	93F5	IBUFL	85F0	IBUFN	85F2
INP010	8441	INP020	844C	INP025	844E	INP035	8456	INP040	8458
INP052	8478	INP055	847E	INP056	8497	INP057	849A	INP059	84B1
INP060	84B7	INP065	84B9	INPUT	8429	INPUTS	840D	INPUTW	84F3
INS010	8414	INS030	8426	INW010	8509	KBTAB	84BB	LASTR	85EF
LINE1	3C40	LINE11	3EC0	LINE12	3F00	LINE13	3F40	LINE14	3F80
LINE15	3FC0	LPFTF	85DF	LPR010	8398	LPR020	83A4	LPR090	83AC
LPRINT	8381	LPS010	83B0	LPSTAT	83AF	LSTCUR	85EB	MBUF	88EA
MLDEL	000A	MOR015	8037	MOR020	8064	MOR021	8066	MOR022	807D
MOR025	8083	MOR027	8096	MOR030	80AB	MSG1	85F4	MSG10	87AA
MSG11	87C9	MSG12	87EB	MSG2	86B5	MSG3	86BF	MSG4	86CB
MSG5	86F5	MSG6	8720	MSG7	873F	MSG8	876A	MSG9	8795
NOPRNT	82B1	OFF010	835E	ON010	8343	PRI010	829C	PRINT	828B
PRINTF	85DE	RAN010	8212	RAN020	8226	RAND	8542	RANDOM	81F0
RDM010	854E	RDM020	8555	SCREEN	3C00	SCROLL	83D6	SEED	85E1
SHIFT	856B	SND010	82FE	SNDCHR	82E0	SPACE	832D	SPE005	8197
SPE015	81C3	SPE020	81DF	SPEED	8191	SPEEDF	03C0	START	8000
SUB	8571	TMP1	85DC	TOPS	9459	TSLC	85ED	TTAB	8892
XMIT	8237	XMT010	825E	XMT020	8273				

Figure 13-8. MORG Listing

The variables are followed by system messages and each message is terminated by a zero.

The CTAB table (Character Table) is a table of 128 ASCII characters distributed among all permissible characters. This table is used in the Random mode to generate a random character. A pseudo-random number of 0 through 127 is used as an index to pick up the corresponding character from the table.

The CDTAB (CoDe Table) is a table of 44 characters that represent all valid characters generated by the system. Since these are contained in the first 44 characters of CTAB, CDTAB is equated to CTAB.

TTAB (Timing Table) is a table of 44 entries associated with CDTAB. Each entry is two bytes (one word) long and represents the coded eight-field representation of the Morse code character. CDTAB and TTAB are used to convert a valid character to its dot and dash equivalent.

The next object is MBUF (Memory Buffer), a 2571-byte table that holds all defined messages. Since each message can consist of 256 characters plus a one-byte header of the message number, MBUF is 257*10 bytes long plus one byte. The last byte ensures that there'll always be a -1 terminator for a full buffer.

IBUF (Input Buffer) is a circular buffer that holds the current characters read from the keyboard. You can use it to implement the buffered input while characters are being generated to the cassette output. TOPS is the "top of stack," 101 bytes up from the end of the IBUF area.

The memory map for MORG is shown in Figure 13-9.

Program Description

We'll use a "bottom-up" approach to describe the program modules. We'll start at the lowest-level modules and work our way up to a description of how the upper two levels utilize the other modules to implement the Morse code program functions.

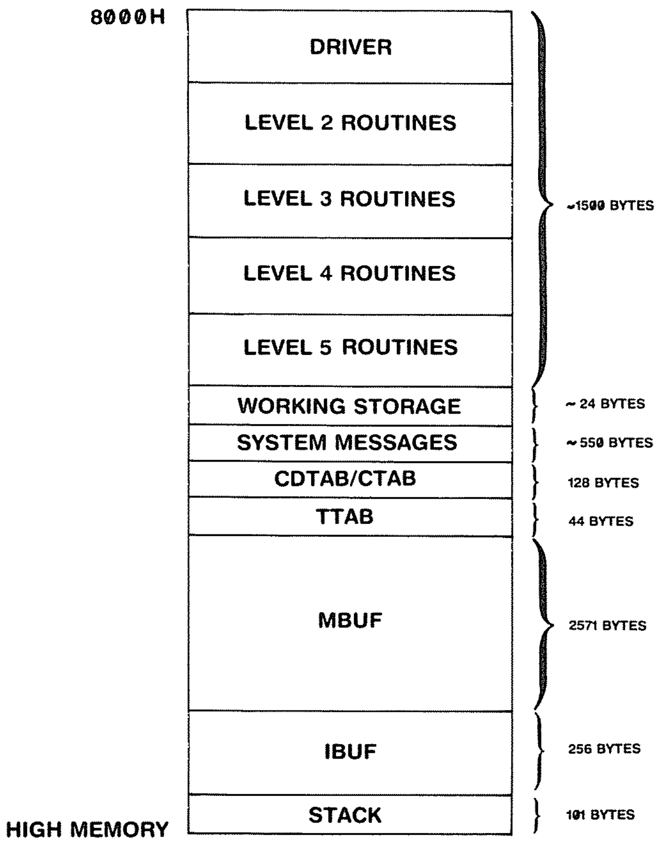


Figure 13-9. MORG Memory Map

Delay Subroutine (DELAY)

The DELAY subroutine delays from 1 to 65536 milliseconds, depending upon the input count in the HL register pair. In addition to the delay, it increments the software "elapsed time" variable, TSLC, by the timing count. TSLC holds the rough elapsed time in milliseconds and is used only for timing the debounce delay.

DELAY works by decrementing the delay count in HL by -1 in DE. Note that the decrement is done by an ADD HL,DE, which produces a carry if the contents of HL are positive or zero.

Decimal-to-Binary Conversion Subroutine (DECBIN)

This subroutine takes a string of ASCII characters representing decimal digits and converts it into a binary number of 0 through 65535. Entry is made with HL pointing to the buffer containing the leftmost character of the string and B containing the number of characters in the string.

The result is in HL on exit. If an invalid character is found, DECBIN stops conversion and returns with the Z flag reset (NZ). An invalid character is defined as one that's not 30H(0) through 39H(9).

DECBIN performs the conversion by taking an ASCII character, subtracting 30H to change it into binary 0 through 9, testing the result for validity, and adding the 0 through 9 to a partial result in IX.

The loop at DEC040 is entered once for each character of the string. Each time the loop is entered, the partial result is multiplied by 10 with a shift-and-add technique.

Fill Character Subroutine (FILLCH)

The Fill Character subroutine fills a specified area with a given fill character. It operates with a simple loop that tests for a decrement of BC down to zero.

Subtract Seed Subroutine (SUB)

The Subtract Seed subroutine is a four-byte multiple-precision subtract that subtracts the contents of DE,HL, (treated as a four-byte integer value) from the "SEED" variable, a four-byte value in SEED through SEED+3. The result goes back to SEED. Two subtracts are done, the second using a possible carry from the lower-order subtract.

Shift Subroutine (SHIFT)

The SHIFT subroutine shifts the contents of DE,HL, treated as a four-byte integer, one bit position left in logical fashion. SHIFT is used to multiply by two and is called by the RAND routine.

Random Number Subroutine (RAND)

RAND is a subroutine to generate a pseudo-random number. The algorithm for the routine is this: Starting with a "seed" value, a new pseudo-random number is generated by multiplying by an odd power of 5, **modulus** 64K. "Modulus" simply means that the result is divided by 64K, the quotient is discarded, and the remainder is saved. This process is done automatically as the result is held in a four-byte integer variable in SEED through SEED+3; the maximum value here is 65535.

The odd power of five chosen here is 125 (5 to the third power). The old "seed" is put into DE,HL from SEED. The SHIFT subroutine is then called 7 times to multiply the value by 128. The original value in SEED is then subtracted three times from DE,HL by calling the SUB subroutine three times. The result is the old seed*125. This value is put back into variable SEED for the next generation.

Hints and Kinks 13-5
Random Number Notes

The algorithm used here is

R(N+1) = 125*R(N) mod 2^{32}

The mod 2^{32} operation is automatically performed by working in 32 bits.

You can generate about 2^{30} or 1 billion numbers without repeats in this approach. I haven't verified this, having quit after scanning only about 1 million cases.

A 7-bit **mask** of the high-order byte of the new random number is performed to get a value of 0 through 127. This is returned in BC. Of course, B will always be 0 on return, but it's convenient to have the number in a register pair for further processing.

Get Character Subroutine (GETCHR)

The Get Character subroutine gets the next character, if any, from the circular input buffer IBUF. During normal operation IBUF is being filled with characters as they are read in from the keyboard. The characters keep filling up IBUF as fast as they're input. Also during normal operation, characters are being output from IBUF to the Morse Code Send Character (SNDCHR) routine, and for display on the screen.

We use this routine to handle the task of seeing if another character is ready for output and display. The routine uses two pointers, IBUFL and IBUFN. IBUFL holds the address of the next character to be output. IBUFN holds the address of the next "slot" in IBUF. If these two pointers are equal, the GETCHR subroutine has caught up with the character storage.

If the pointers are not equal, GETCHR uses IBUFL to get the next character. It then bumps IBUFL by one to point to a possible next character. A **limit** condition occurs when IBUFL points to the last character position of IBUF+1. If this is true IBUFL is reset to the start of IBUF.

On exit, A contains the ASCII character found in IBUF or zeroes if there was no character available. The Z flag is either set if there was no character or reset if there was a character.

Input Character and Wait (INPUTW)

The Input Character and Wait subroutine is called to pick up a single character for user input. It is not used during normal operation, only for response to system queries such as code speed and message number. In these cases, no buffering need be used.

INPUTW calls INPUT to input the next character. INPUT is the **fast** input routine used for all character input. Before calling INPUT, INPUTW "dummies up" the "Time Since Last Character" variable TSCL to make it look like the debounce time has elapsed. It does this each time a new character is required. This approach makes it possible to use INPUT for both "wait until next character" and "scan fast and return" functions. INPUTW does not exit until a character has been input to the keyboard.

Keyboard Input Subroutine (INPUT)

This subroutine is the high-speed keyboard input subroutine that permits **buffering** of input characters during transmission of code characters. INPUT has three basic functions: fast scan, conversion, and storage.

When INPUT is called it loads the A register with input/output address 387FH. This address enables all "row" addresses of the keyboard. If you are pressing **any** key, there will be a one bit in A after this LD. In effect, this is an OR of all rows into eight column bits. If you aren't pressing any key, INPUT immediately returns. This sequence takes six instructions.

If there is a one bit in A after the fast scan, INPUT checks the elapsed time by testing variable TSLC. If the debounce time has not elapsed (DBDEL), INPUT returns since the key press may represent the same key input on the last read.

If the debounce time has elapsed, INPUT goes into the conversion processing. It now scans each row by reading in I/O address 3801H, 02H, ... If a one bit is found after any read, the conversion routine at INP020 is entered. At this point HL contains the row address. The column bit is converted to a number from 0 through 7 and added to 8 times the row value to get an index of 0 through 56.

The index value is added to the address of the start of KBTAB. KBTAB is a table of 56 characters representing the keyboard configuration. KBTAB represents the "unshifted characters" from the keyboard. The character represented by the keypress is read from KBTAB and put into A. If this character is a zero, it is an "ignore" character and a return is made.

The SHIFT key is now read and stored in B. (NOTE: On Model III, this is the LEFT SHIFT key.) If the character read is either slash (/) or minus (-), the SHIFT key is tested; if SHIFT is being pressed, the slash or minus is converted to a question mark (?) or equals (=) by setting bit 4 of the character.

A test is then made for the CLEAR character. The CLEAR key can be pressed to reset operation of the program. If CLEAR is being pressed, the program is restarted at MOR015 after some cleanup of the current cursor position (CURCUR).

If CLEAR is not being pressed, the code at INP057 is entered. This code stores the character in IBUF by using IBUFN, the variable that points to the next IBUF character "slot." After the store, IBUFN is incremented by one. A test is then made to see that IBUFN has not gone beyond the end of IBUF. If it has (IBUFN=IBUFE), IBUFN is reset to the beginning of IBUF. Since a character has just been read, the "time since last character" variable TSLC is reset to zero, and the subroutine is exited.

Input String Subroutine (INPUTS)

The Input String Subroutine is used to input a user response to a system question such as that for code speed. It calls INPUTW, Input Character and Wait. As each character is input, it's displayed on the screen by a CALL to DISCHR. INPUTS detects the end of the input string by testing for an ENTER and returns with a character count in B. The character count will be used in converting the (decimal) string to binary in DECBIN.

Clear Communications Area (CLRCOM)

The Clear Communications subroutine calls FILLCH to clear the last three lines on the screen (the communications area). Blanks are filled.

Scroll Subroutine (SCROLL)

SCROLL is used to scroll up the first 12 lines of the screen. This is the "text" area used to display text as Morse code characters are entered and transmitted. Lines 1 through 11 are moved up into lines 0 through 10 by an LDIR instruction. The last line of the text area, line 11, is then filled with blanks by a call to FILLCH.

Display Character Subroutine (DISCHR)

The Display Character Subroutine stores a character on the screen at the current cursor position. It is used for all screen output, both in the communications and text area. The current cursor position is always held in variable CURCUR as a screen address. After the character is stored, CURCUR is incremented by one. If it now points to the last position of line 11, a CALL to SCROLL is made.

Line Printer Subroutine (LPRINT)

LPRINT is the line printer driver subroutine. If the PRINTF variable is non-zero, the "P" command has been given for simultaneous line printer output. In this case, the character is output to the system line printer. On the first output to the line printer (LPFTF=0) and after 32 characters have been output to the line printer, the subroutine automatically outputs a carriage return for a new line. This eliminates overprinting in some system line printers.

LPRINT uses an "internal" subroutine LPSTAT to read line printer status.

Hints and Kinks 13-6
Line Printer Output

LPSTAT is a typical line printer driver for parallel line printers. The handshaking between the computer and line printer is probably the simplest of any peripheral. A line printer is ''ready'' when it has finished printing a character and ready to accept the next. The inverse of ready is ''busy.'' In an unbuffered line printer, the busy time duration in seconds is 1/N, where N is the number of characters per second.

A status loop checks the busy status (LD A,(37E8H)) until the line printer is ready to accept the next character, at which time the CPU outputs the character (LD (37E8H),A) causing the line printer to again become busy. The busy flag is set automatically by the line printer electronics during printing and reset after printing.

Buffered line printers can accept characters during printing as long as the buffer does not become full.

Display Message at Location N (DSPMES)

DSPMES is called for system messages, such as those prompting the user to enter code speed. On entry, HL points to the start of a message string. This string is assumed to be terminated by a zero (null) character. DSPMES picks up a character at a time and stores it on the screen by using the contents of BC as a pointer. HL and BC are incremented after each store. When a null is detected, DSPMES returns. Variable CURCUR, Current Cursor, is adjusted for each store.

Send Character Subroutine (SNDCHR)

This subroutine is the heart of MORG as far as transmission of audio code characters. It's called with the A register containing the ASCII character to be transmitted.

SNDCHR first searches for the character in CDTAB, the character code table. As SNDCHR can only be called with a valid character, the search must be successful. When the character is found in CDTAB, its index of 0 through 43 is multiplied by 2. The index now can be used to pick up the corresponding TTAB entry, which contains the coding of dots, dashes, spaces, and terminator. The character is picked up from TTAB by adding the index to TTAB and loading the character into H and L.

The code starting at SND010 and ending at the next return is the major "output" loop of SNDCHR. The contents of HL are shifted two bits right. The two bits shifted out are saved in A and tested. They are either 00 (dot), 01 (dash), 10 (space), or 11 (terminator). If the bits are a terminator, the subroutine is exited.

If the bits represent a dot, the DOT subroutine is called; if a dash, DASH is called.

DOT and DASH call two other subroutines at a lower level, CON and COFF. CON produces an audio tone of 500 hertz by outputting alternate 01 and 10 to the two least significant bits of the cassette latch at address 0FFH. DELAY is called to time out the 1 millisecond delay between an on and off condition. The output goes on for 1 millisecond, then off for one millisecond, producing a 500 hertz **square wave**.

COFF is similar to CON, except that it leaves the cassette latch off and performs only the delay.

DOT calls CON to produce the tone for one dot time (DOTO) and then calls COFF for one dot time. DASH calls CON to produce the tone for one dash-time (DASHO) and then calls COFF for one **dot** time.

If the two-bit code was a 10 in the main loop of SNDCHR, an inter-character space is called for. In this case, only COFF is called for two dot times, as one dot time has already been output for the last dot or dash. The total elapsed time is three dot times.

Find Message Subroutine (FNDMSG)

This subroutine is used to search the Memory Buffer MBUF for a given message number. It's possible that the message has not been defined.

A search is made by scanning for a given message number 0 through 9. This byte can only exist if a message has been stored in MBUF, as other bytes are ASCII characters, minus ones, or other digits.

If the message number is found, FNDMSG is exited with HL pointing to the message, otherwise HL points to the next available area for the message. A CPIR instruction is used for the search in a small subroutine called FNDSR.

Main Driver

The subroutines above are used by the main body of code to implement the functions of MORG. The main driver of MORG starts at START. All "restarts" (as, for example, a CLEAR) come back to MOR015.

The code from START to MOR015 initializes MBUF with minus ones, initializes the SEED, clears the screen, and draws the line separating the text area from the communications area. The restart code at MOR015 initializes the IBUF pointers and TSLC and outputs the initial message.

MOR020 is the start of the loop to output characters or messages. A call is made to INPUT to scan the keyboard and then to GETCHR to get a possible character. If no character is available, a "minor" count in D is incremented. When this count is equal to MLDEL, approximately one millisecond has been reached, and variable TSLC is bumped by one. This software counting is necessary as TSLC is used in the debounce delay in INPUT. If it were not done, the keyboard routine could be "locked up" waiting for the debounce delay.

If a character is present, it's either a character to be transmitted or a special function that uses the shift key and 0 through 9, T (Transmit Message), D (Define

Message), S (Define Speed), R (Random), P (Print), or N (No Print). If no shift is present, the character is displayed (DISCHR), printed (LPRINT), and transmitted (SNDCHR) and a return is made to MOR020.

If a special SHIFT key has been pressed, a table of allowable functions, FTAB, is searched for the key. If none is found, the keypress is ignored. If a match is found, the index from FTAB is used to pick up an address from **branch table** BTAB. The entries in this table match the entries for the character. A branch is then made to one of six routines that will implement the function required. Return will be made back to MOR015 at the completion of the function.

Function Routines

The Define Message Routine (DEFINE) is entered if SHIFT D is pressed. It calls CLRCOM to clear the communication area and then prompts the user to input a message number. A test is made of the input for a valid number and an error message is output if the number is not 0 through 9.

FNDMSG is called to check for a current message with that number. If a current message is found, it's deleted by moving up the remainder of MBUF on top of the existing message. The remainder of MBUF is then filled with minus one bytes.

The loop at DEF040 is then executed. This loop uses INPUTW to get a character, store it in MBUF, and display it by DISCHR. A check is made for messages exceeding 256 characters with an appropriate message output if this is the case. The main driver loop at MOR015 is reentered when an ENTER is input, ending the message.

The Set Speed Routine (SPEED) is entered if a SHIFT S is input. It clears the communication area and displays a prompt message for user speed input. INPUTS is used to get a character string, which is then converted to binary by DECBIN.

If the speed value is less than 3 or greater than 60, an error message is output, and the sequence is repeated. Otherwise, the speed value is used to divide system value SPEED (nominally 1200, but adjusted for system overhead) to obtain the dot time in milliseconds. This value is stored in DOTO. The dot time is multiplied by three to obtain the dash time. This value is stored in DASHO. The main driver is then reentered at MOR015.

The Transmit Random Characters Routine is entered when a SHIFT R is input. This routine clears the communications area. It then calls RAND to get a random number of 0 through 127. This value is then used to get a character from the 128-byte CTAB. If a blank character is picked up, a test is made to see if the last character was blank. If it was, a new character is generated to avoid sending two blanks in a row.

The character is then displayed (DISCHR), printed (LPRINT), and transmitted (SNDCHR). A test is then made for a CLEAR character by calling INPUT even though no input is taking place for characters. A CLEAR will restart at MOR015, otherwise RAN010 is executed.

The Transmit Message Routine (XMIT) is entered if a SHIFT 0 through 9 is entered. The communications area is cleared. At this point, the message number is still in the BC register pair. It is used to call FNDMSG to find the location of the message in MBUF. If the message number is not found, an error message is briefly output and the main driver is reentered at MOR015.

If the message is found, it is transmitted character by character until a non-ASCII character is detected (new message number header or minus one). DISCHR, LPRINT, and SNDCHR are called to display, print, and send each character.

The Print (PRINT) and No Print (NOPRNT) Routines are entered by SHIFT P and SHIFT N, respectively. The only action in each is to clear the communications area and display a short `PRINT SET` or `PRINT RESET` message. The main driver is then reentered at MOR015.

Using This Program

There are several alternatives in using the program. First of all, of course, you don't really have to use it at all except for reference. It illustrates many of the concepts we discussed earlier in the book.

If you care to actually run the program, you may key in the machine code by using Disk DEBUG or T-BUG. The program is assembled at 8000H and, of course, is non-relocatable except by reassembly. At first blush, this seems like a formidable task, as there are several thousand bytes. However, it can be done in slightly over an hour for a fast typist. **Checkpoint** occasionally by saving what you've done. Also, mind the locations! There is nothing worse than completing a large input of several thousand locations and finding out at the end that you've been one location off for 1500 bytes!

A third alternative is to key in the source. This **is** a formidable job, but does allow you to experiment with the program once you have captured the program on disk or tape.

Chapter Fourteen
Tic-Tac-Toe Learning Program

We're presenting another larger assembly-language project in this chapter — a program that plays tic-tac-toe. The difference between this and other programs that play the game, however, is that this one learns! It starts off playing all possibilities, but quickly learns which paths lead to winning games.

The program was assembled using the Disk Editor/Assembler, but can be easily modified to run on EDTASM by the simple format changes of adding colons after labels and using single arguments on data generation pseudo-ops.

General Specifications

I can sum up the general specs for the program in a paragraph or so: Write a program that plays tic-tac-toe with a human. The program always makes the first move. The program must start out by playing any possible sequence and must not use any predetermined logic to decide where to play, except for the obvious capabilities of being able to block a human's winning row, column, or diagonal and of trying to complete its own winning row, column, or diagonal.

As the program plays games, it should record the win, loss, and draw records of the games. More importantly, it should "learn" which paths produced winning games and somehow reinforce those paths. It should also learn which paths caused it to lose and avoid those sequences.

The program should, of course, draw a tic-tac-toe grid, put in Xs and Os, prompt the human, display the win/loss/draw record, and do all the usual things in interacting with the user.

Operation

Here again, as with the Morse Code Generator program, we can define the operation of the program before actually designing it — we can define the external or outward operation of the system. Whether or not we can actually implement such a program is still a question at this point.

The program should draw a tic-tac-toe grid on the screen as shown in Figure 14-1. The computer's symbol will be an X and the human's an O. Initially, and at each new game, the title `TIC-TAC-TOE` will be displayed at the top of the screen.

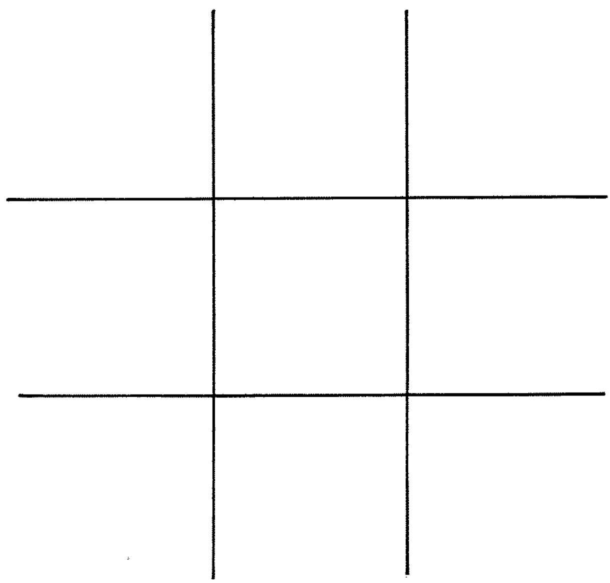


Figure 14-1. Tic-Tac-Toe Grid

The computer moves first, placing an X in one of the nine positions, and then prompts the user with a `YOUR MOVE` message. The human then responds by entering a digit of 0 through 8, depending upon the position in which he wants to place an O. After the O is placed, the computer plays again and prompts the user. This dialogue continues until either the human or the computer wins.

If the human makes an invalid choice of square (one that is already occupied by an X or O), the message TRY AGAIN is displayed, and the human must try a new move.

At the end of the game, the program displays either YOU WIN!, I WIN, or DRAW and then displays the message ONE MORE? If the human wants another game, he can press any key from 0 through 8 to start a new game.

At the end of each game, a "history" of the past 128 games is displayed on the bottom three lines of the screen as shown in Figure 14-2. There are 2 lines of 64 characters on the bottom of the screen. As each game is played, a W, L, or D is displayed in the next position for win, lose, or draw. When the 128 positions are filled up, the history "slides" to the left to display the last game and previous 127 games.

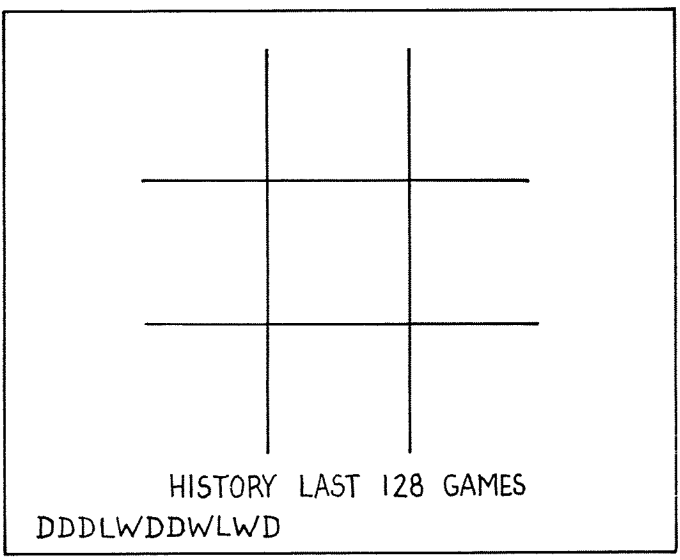


Figure 14-2. Tic-Tac-Toe History

Messages are displayed in "large format" at the top of the screen, except for the history message, which is standard alphanumeric size.

All of the above seems perfectly achievable for an assembly-language or BASIC program, with the exception of the internal workings of the "learning" capability. We'll discuss this major point next.

General Design

The general design of this program can be divided into several areas of research:

- Nature of tic-tac-toe
- Alternatives to learning
- Algorithms

Nature of Tic-Tac-Toe

Just to start from "square one," let's review the rules of tic-tac-toe. Tic-tac-toe is played on the grid shown in Figure 14-1. There are nine squares in the grid. Two opponents play, one using Xs and one using Os (an old name for the game is "naughts and crosses").

Each player plays in turn, putting his X (or O) into a vacant square. The first player to successfully put all Xs or Os into a row, column, or diagonal wins the game. If neither player can complete a row, column, or diagonal, the game is a draw. A sample game is shown in Figure 14-3.

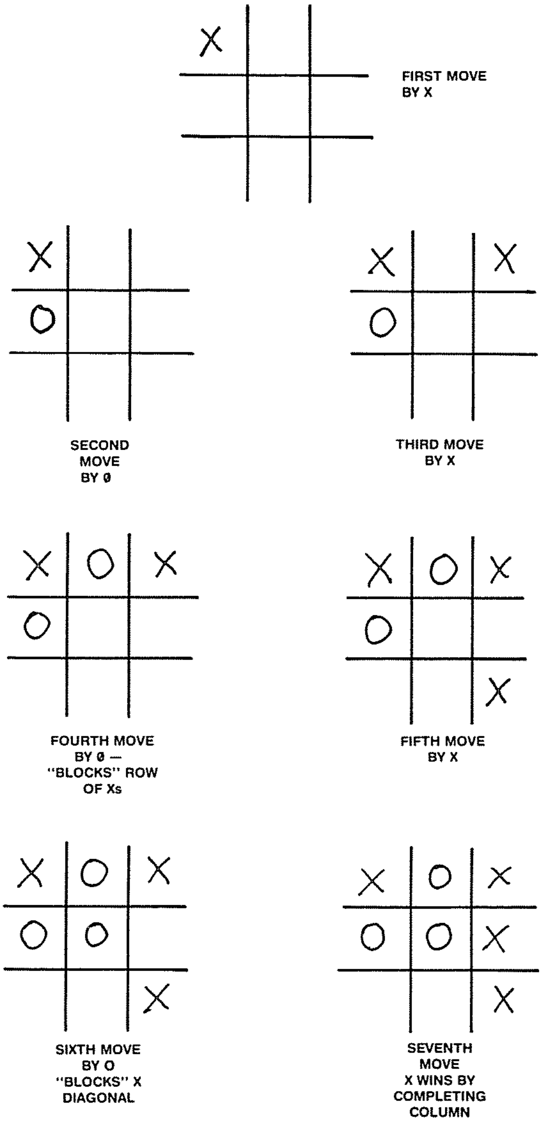


Figure 14-3. Sample Tic-Tac-Toe Game

Let's start our research by asking, "How many **different** games can be played?" At first, this seems like a question for a statistician, but we can draw some conclusions about it.

Each square can only have an X, O, or space (no play) in it. Therefore, we can represent any move of any game by listing all possible configurations of Xs, Os, and spaces. If we start to do this by hand, we can get confused quite easily, as shown in Figure 14-4.

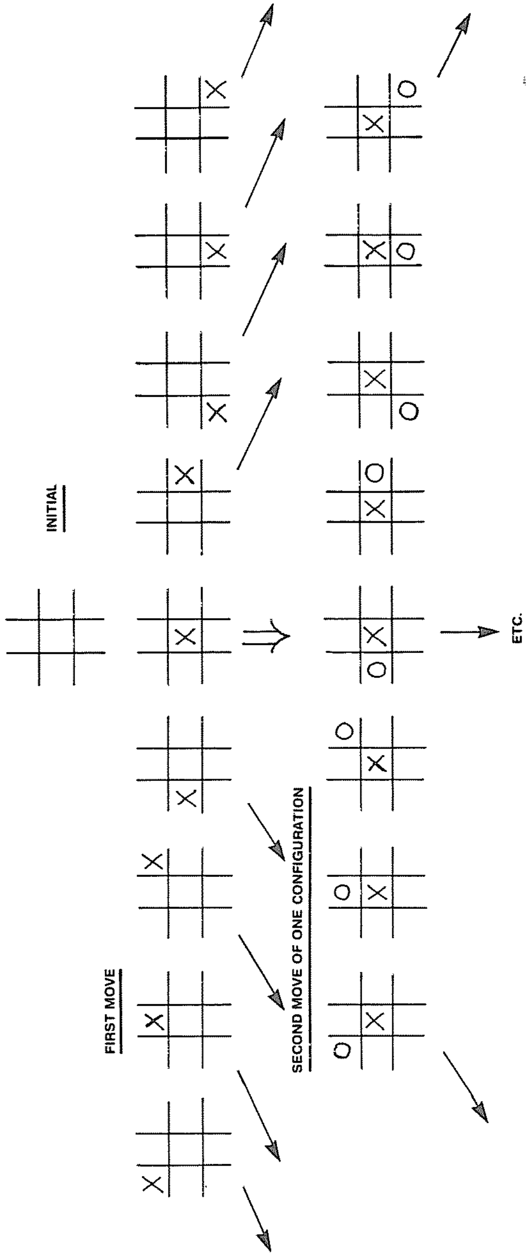


Figure 14-4. Manual Analysis of Tic-Tac-Toe

There are just too many combinations to keep track of! Let's try a different approach. We'll assign an X a one value, an O a two value, and a space a zero value. Now we can write down any configuration, beginning of game, middle of game, or end of game by a series of 0,1, and 2. We'll number the squares 0,1,2,3,4,5,6,7, and 8 as shown in Figure 14-5.

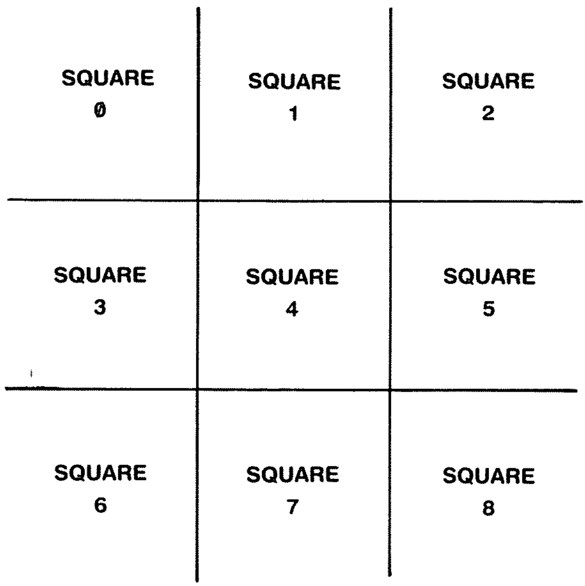
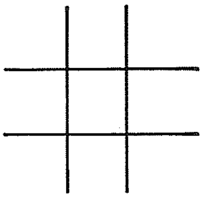


Figure 14-5. Tic-Tac-Toe Squares

Sample configurations are shown in Figure 14-6. The beginning of a game has all blanks and is therefore represented by 0-0-0-0-0-0-0-0-0. A typical "middle game" shown in the figure has a combination of Xs, Os, and blanks and is represented by 1-2-1-0-2-0-0-0-0. A typical end game shown in the figure is represented by 2-1-0-2-2-0-1-1-1.

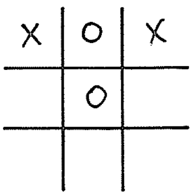


BEGINNING OF GAME

SQUARE

0	1	2	3	4	5	6	7	8

0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0

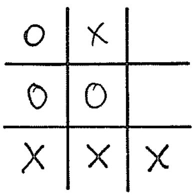


TYPICAL MIDDLE GAME

SQUARE

0	1	2	3	4	5	6	7	8

1 - 2 - 1 - 0 - 2 - 0 - 0 - 0 - 0



TYPICAL END GAME

SQUARE

0	1	2	3	4	5	6	7	8

2 - 1 - 0 - 2 - 2 - 0 - 1 - 1 - 1

Figure 14-6. Tic-Tac-Toe Configurations

Figuring Permutations

How many different combinations, or more precisely, **permutations** are there of Xs, Os, and spaces? We can find out by listing all permutations of the numbers. We'll start with 000000000, and end with 222222222. We've eliminated the dashes for compactness.

Wait, a minute, this looks suspiciously like a range of numbers, not binary, since there are **three** symbols, but **base three**. This is no more mysterious than binary! There are three symbols, 0, 1, and 2, and we count much the same way — 0,1,2,10,11,12,20,21,22, ...

Hints and Kinks 14-1
Base 3 Numbers

Base 3 numbers <u>(ternary)</u> use the same positional notation as binary, decimal, and hexadecimal. The rightmost digit is 3^0, next is 3^1, and so forth.

212

$$2 \times 3^0 = 2$$
$$1 \times 3^1 = 3$$
$$2 \times 3^2 = 18$$
$$23_{10}$$

The same conversion techniques used in binary or hexadecimal can be used to convert between decimal and base three – double dabble:

2∅21

$$2 \times 3 = 6 + \emptyset = 6$$
$$6 \times 3 = 18 + 2 = 2\emptyset$$
$$2\emptyset \times 3 = 6\emptyset + 1 = 61_{10}$$

and ''divide by 3, save remainders'':

$$\left.\begin{array}{ll} & \emptyset \text{ R2} \\ 3 \overline{)\ 2} & \text{R}\emptyset \\ 3 \overline{)\ 6} & \text{R2} \\ 3 \overline{)\ 2\emptyset} & \text{R1} \\ 3 \overline{)\ 61_{10}} & \end{array}\right\} 2\emptyset21_3$$

Just as a nine-digit binary number can hold 2 to the ninth permutations, a nine-digit base three number can hold 3 to the ninth permutations, or (let me use my pocket calculator here) 19,683 permutations.

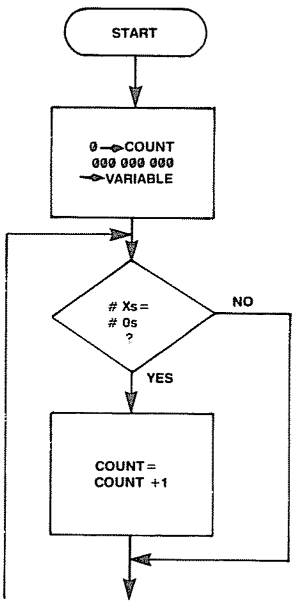
One of those permutations will represent any arrangement of Xs, Os, and spaces we care to define. Many of the permutations are not possible in a game, such as 111111111, 222222222, 222222000, and many others!

As a matter of fact, we can write a fairly simple assembly-language program to figure out the number of permutations we'll have to work with. 19,683 permutations are really too many to fit into memory with everything else. We'd like to see if we can reduce that number down to something manageable!

Reducing the Number of Permutations

We'll start by making some assumptions about the game to simplify things. First of all, the computer will always play first with an X. Secondly, we'll look for those permutations in which the computer is to play next, not in which the human is to play. We'll never have to deal with the latter case. This means that the number of Xs and Os have to be equal.

A flowchart for such a program is shown in Figure 14-7. It cycles a variable from 000000000 through 222222222 in base three. For each number, it tests the number of Xs (ones) and Os (twos). If they are equal, it adds 1 to a count. At the end, the count has the number of possible permutations of "computer to play next" (this includes 000000000 where the computer has not yet played, but #Xs = #Os = 0).



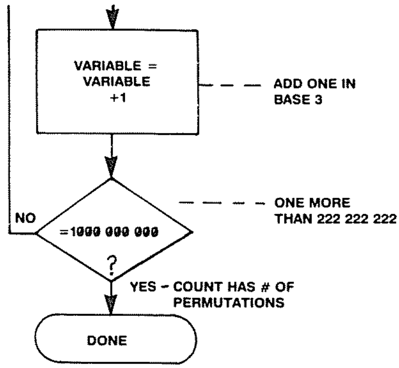
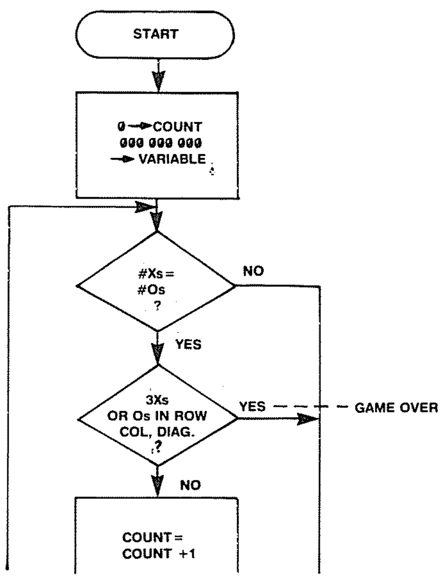


Figure 14-7. Flowchart Try # 1

When we run such a program, we find that there are about 3200 such permutations, a large reduction from all possible permutations of 19,683.

Still, we would like to reduce the permutations further. Well, we can delete all permutations where the game has already been won! We'd never continue from such a point in an actual game. This would be the case in which there are three Xs or Os in a row, column, or diagonal. The flowchart for such a program is shown in Figure 14-8.



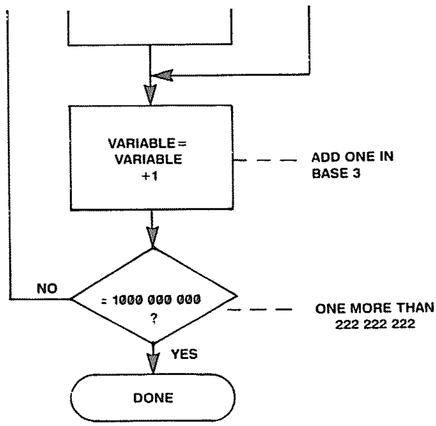


Figure 14-8. Flowchart Try # 2

When we implement the program, we find that there are now about 2460 valid permutations, a further reduction.

Another reduction we can make is to eliminate "9th move" permutations. In this case, 8 squares have been filled and the computer has only one choice. When this is added to the analysis program, we have cut down the allowable permutations by another 100 or so.

We can reduce further by eliminating permutations where there are two Xs in a row, column, or diagonal. This means the computer will win on the next move and knows enough to finish the game. This reduces the number of permutations we have to deal with still further.

Are there any other reductions possible? Yes, a major one. We know that there are many configurations of tic-tac-toe moves that are identical except for **rotation**. Figure 14-9 shows some of these. Also, there are identical games when "mirror images" are considered. We can cut down on the permutations we have to deal with drastically by considering rotations and mirror images.

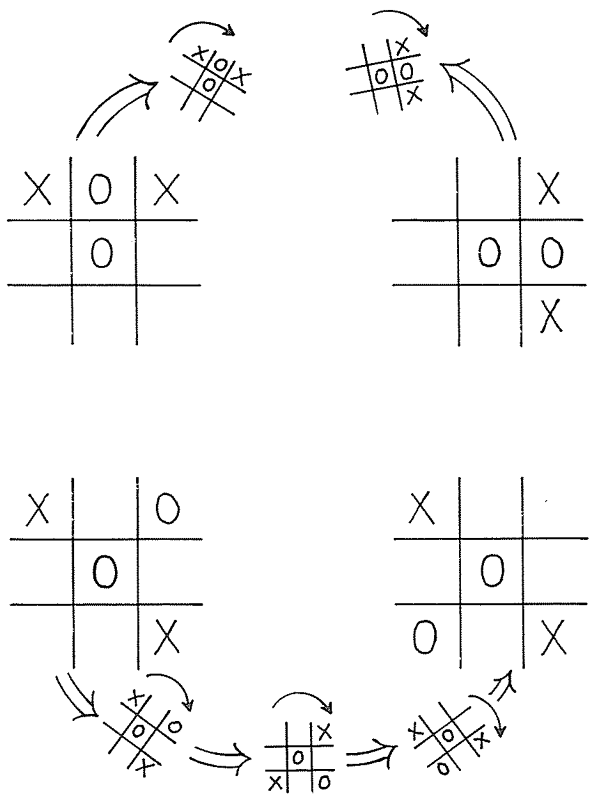
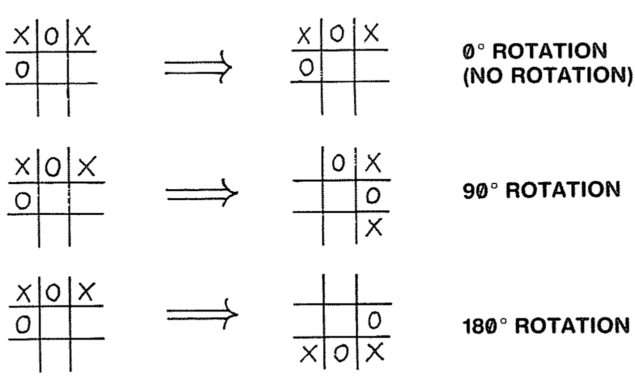


Figure 14-9. Rotation Identities

There are eight rotations and mirror images to consider, and they are shown in Figure 14-10.



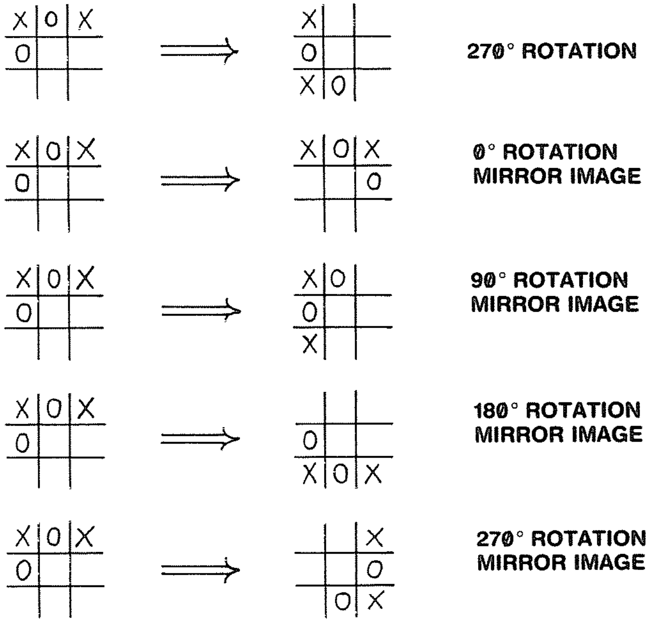


Figure 14-10. Rotations and Mirror Images

Our analysis program now eliminates:

1. All permutations in which the number of Xs and Os are not equal.
2. All permutations for the last move.
3. All permutations in which the game has already been completed.
4. All permutations in which the computer will win on the next move.
5. All permutations which are rotations or mirror images of other permutations already recorded.

The flowchart for figuring out the number of permutations we have to deal with is shown in Figure 14-11.

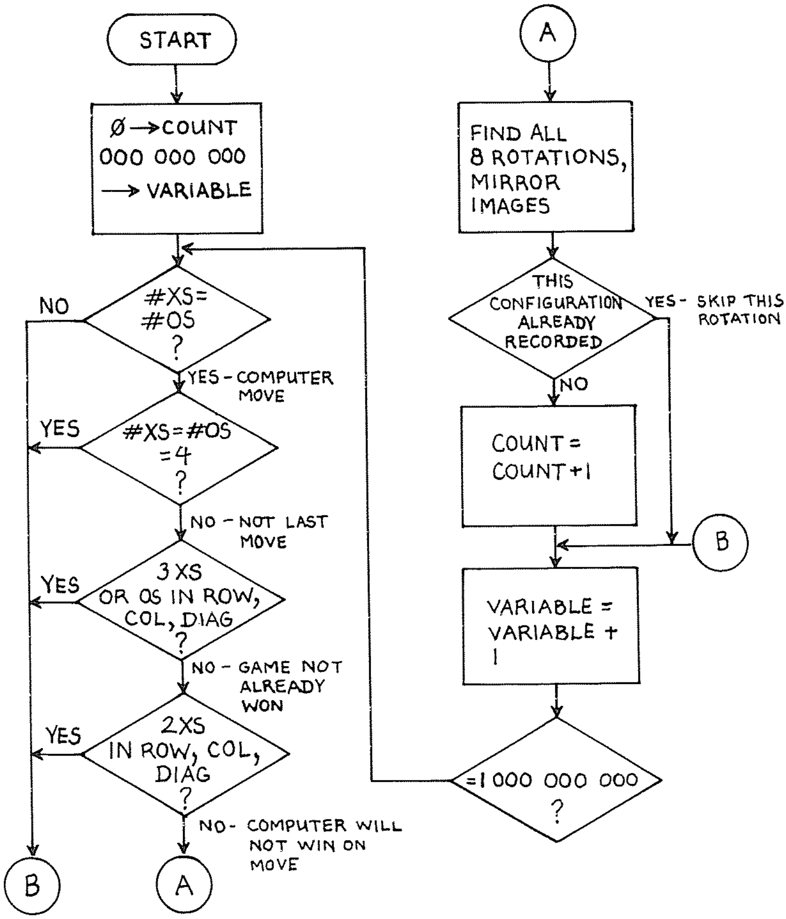


Figure 14-11. Final Flowchart

What do we really have after totaling the number of permutations in this fashion? We have a complete record of all configurations that we would encounter in any tic-tac-toe game in which the computer plays first and knows enough to complete a row, column, or diagonal; to recognize that the game has been lost; and to recognize that it will win on the next move. More importantly, the number of permutations we have to deal with have been reduced from 19,683 to 120! Now we can fit the permutations in memory and possibly implement the program!

Alternatives To Learning

Given a reasonable number of permutations to work with, how do we implement artificial intelligence in the program? How do we make it learn?

One approach would be to implement some sort of giant **binary tree** structure in the program as shown in Figure 14-12. We'd start off with the "empty" configuration and construct all possible courses a tic-tac-toe game could take. Then, if the computer lost, we would delete the last move so that the computer could never make that play again. When all possible lower branches were deleted, we'd delete the upper "limb." Given enough games, we'd have a computer that "learned" by never making the same mistake twice.

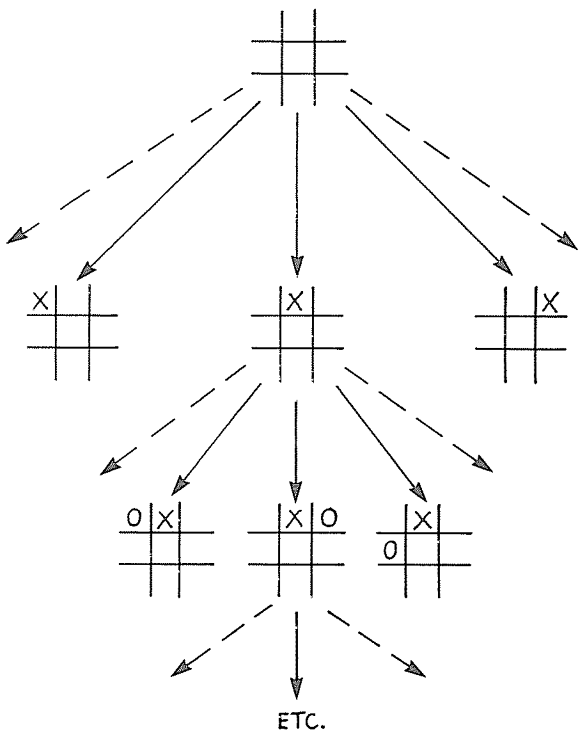


Figure 14-12. Tic-Tac-Toe Binary Tree

Hints and Kinks 14–2
Learning By Losing

If we implemented the ''learning by losing'' algorithm, we'd have a very mechanical learning machine, and we'd have to play hundreds of games to make much progress.

What we really want is not so much an infallible ''idiot savant'' as something that will emulate human learning – learning by trial and error with reinforcement of successful actions and rejection of unsuccessful approaches.

Another way to create a learning process was described in the pages of Scientific American some years ago, and attributed to work done by Michie at the University of Edinburgh. Suppose that we emulate the tree structure described above by a series of boxes. Each box is marked with the permutation that we've allowed by our analysis program.

Inside each box, we put different colored balls, each color representing one of the paths that could be taken. There could be nine colors for a "first move" configuration, seven for a "third move," five for a "fifth move," and so forth.

We'll put four balls of each color into first move boxes, three of each color into third move boxes, two of each color into fifth move boxes, and one of each color into seventh move boxes. By the ninth move the game will be over, and we have no ninth move boxes. This situation is shown in Figure 14-13.

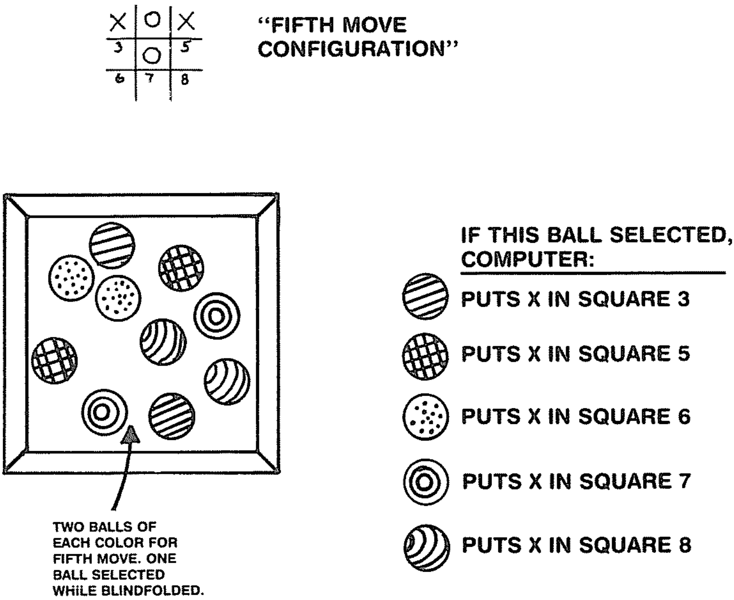


Figure 14-13. Boxes and Balls

Now we'll play a game. For each move, we'll shake the box and withdraw one ball. We'll note the color and put it back in the box. Its color will determine in which square the "computer" will play. The human now makes a move. After the human's move, we go through the process again, choosing one ball at random from the next box representing the current configuration.

Eventually the "computer" wins, loses, or draws. If the computer wins, we'll go back and add three balls for each color chosen to each box. For example, if the first move box was a red, signifying square 2, we'll add three red balls. We'll also add three balls of the appropriate color to the third move, fifth move, etc., boxes.

If the "computer" loses, we'll take away one ball of the proper color from each of the boxes involved. If the "computer" draws, we'll add only one ball of the appropriate color.

If a particular color (which is really a number of a square) consistently wins, we'll start accumulating a higher and higher proportion of balls of the color (square) that produces winning games. Similarly, if a color consistently loses, we'll have fewer and fewer balls of that color.

Since one ball is chosen at random for each move, we'll have a better chance to withdraw winning colors (squares) as more games are played. Winning games "reward" the square choice, losing games "punish" the square choice, and draws "reward" the choice slightly. This emulates the reward and punishment of human learning for a mechanical process and is much more interesting and faster than just deleting losing paths. This is the method we've chosen to adopt in our implementation of tic-tac-toe on the TRS-80 in this program.

Algorithms

The **algorithms** we'll be using for the tic-tac-toe program are described in the following section. These algorithms emulate the tic-tac-toe learning procedure described above — the "reward" and "punishment" method. They are divided into two parts:

1. Generation of a table of permutations representing legitimate tic-tac-toe games, and
2. Algorithms for playing the game itself.

Generation of a Permutation Table

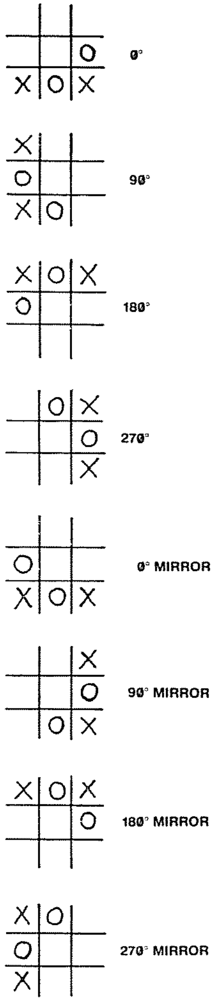
First of all we must generate a table of all the possible permutations the computer will encounter in playing the game of tic-tac-toe. As we described above, these represent configurations where the computer is to play: the so-called "first move," "third move," "fifth move," and "seventh move" conditions.

In generating the permutations, we'll discard those where the game has already been won or where there are two Xs (computer) in a row, column, or diagonal. We'll also look for the "rotations" and "mirror images" and pick only one out of eight possible, discarding the rest. At the end of the table generation we'll have a table of hundreds of entries that represent any configuration that the computer will encounter when it is playing the game.

The algorithm for table generation is as follows:

1. Start with a value of the base three number 000 000 000. Increment this value by one each time through the steps up to a maximum of 222 222 222. This value is known as the "current value."
2. Start with the address of a table in memory known as the "permutation table" or "PTABLE". For each valid permutation, we'll enter the current value and some other data. We'll build up this table so at the end of table generation, we'll have a table in memory that holds all configurations.
3. Take the current value. Test for the end of 1 000 000 000 (one more than 222 222 222). If not equal, continue; otherwise the table is done.
4. Count number of Xs and Os in current value. If they are not equal, this is not first, third, fifth, seventh, or last move (computer's turn); if not equal, go on to the next current value.
5. Test for number of Xs = number of Os = 4. If this is true, this is the last move. The computer knows what to do here, so go on to the next current value.
6. Test for all Xs or Os in a row, column, or diagonal. (All ones or twos). If there are all ones or twos, discard this permutation as the game has already been won. Go on to the next current value.
7. Test for two Xs (ones) in a row, column, or diagonal. If this is true, the computer will know enough to finish the game as it is the computer's move; go on to the next current value.
8. Rotate (and find the mirror image) of the current value seven different ways. **Take the lowest base three value** and compare it to the current value. If they are equal, save the current value in the PTABLE, otherwise go on to the next current value.
9. At this point we have a "valid" current value, one that will be saved in the PTABLE. A sample current

value that reached this point is shown in Figure 14-14. Follow the next steps to "process" that value.



STEPS

1. VALUE = 000 002 121
2. PTABLE ADDRESS = 9003H
3. VALUE ≠ 1 000 000 000
4. # Xs = # Os = 2 — OK
5. # Xs = # Os ≠ 4 — OK
6. ALL Xs OR Os IN ROW, COL, OR DIAG — NO
7. TWO Xs IN ROW, COL, OR DIAG — NO
8. ROTATE AND MIRROR IMAGE VALUES

0°	000 002 121
90°	100 200 120
180°	121 200 000
270°	021 002 001
0° MIRROR	000 200 121
90° MIRROR	001 002 021
180° MIRROR	121 002 000
270° MIRROR	120 200 100

9. LOWEST VALUE = CURRENT VALUE

Figure 14-14. PTABLE Initial Processing

10. Enter the current value in the next position of the PTABLE.
11. Count the number of spaces in the current value (number of zeroes). Put this number in the next position of the PTABLE. This will range from 9 spaces for a first move to 3 for a seventh move.

12. Allocate a number of bytes equal to the number of spaces from 11.
13. Into each of the bytes, put a value of 4,3,2, or 1, for 9,7,5, or 3 spaces, respectively. In other words, if the number of spaces is 5, put a value of 2 into the two bytes allocated for the spaces. The entry for this permutation is now completed. The sample is shown in Figure 14-15.

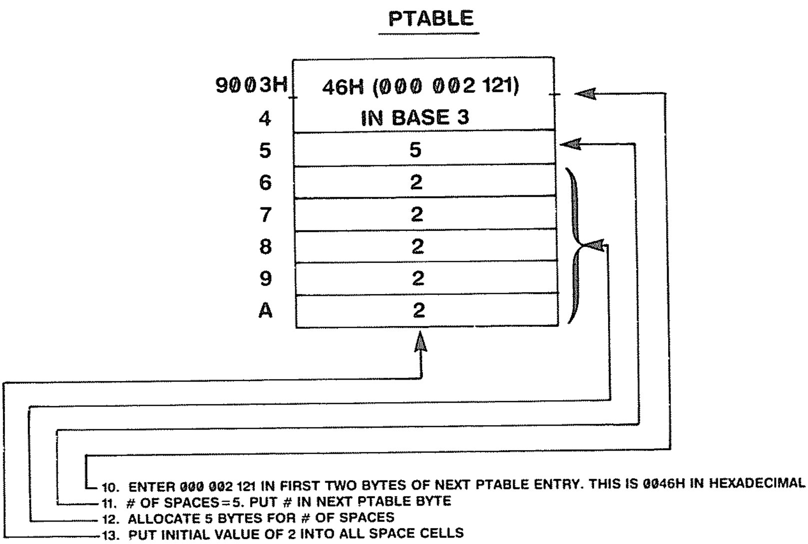


Figure 14-15. PTABLE Final Processing

What we've constructed in the PTABLE above is a computer analogy to the "boxes" we discussed earlier. Each box (entry of PTABLE) has a configuration associated with it represented by the current value. Each box (entry) has 9 to 3 "cells" (bytes), each representing a space or move the computer can make, reading left to right, top to bottom, the same way you would scan a page.

Into each space cell, we've put a value which is the same as putting in a certain number of colored balls. Considering all space cells together, we have a number of balls, each representing a certain move. We'll add to or subtract from this count by changing the number in the space cells as we total up games. We'll withdraw one ball by choosing a random space cell of the 9 to 3 space cells available. We'll discuss this in more detail in the next section.

Algorithms for Playing the Game

Having established the "PTABLE", which represents any possible configuration that the computer will have to deal with, we'll now look at how the game is played by using the table to emulate the "boxes and balls" approach.

The algorithm goes like this:

1. Start with a blank array drawn on the screen with the usual grid.
2. Take the current array configuration and "analyze" it — count the number of Xs(computer), Os(human), and spaces. If entering this step from step one, the array will be all blanks.
3. See if the computer can complete a row, column, or diagonal of Xs. If so, perform the completion and display; the computer wins. Record the win and go to step 9. If not, continue.
4. See if this is the last move. If it is, the computer makes the only move available and outputs it to the screen. Then the computer analyzes the new configuration and sees whether it wins, loses, or draws and goes to step 9.
5. At this point, the PTABLE has not been referenced, but the computer has looked for obvious moves. Now the PTABLE will be referenced. The computer now performs all seven rotations to find which of the eight configurations should be found in the PTABLE. The PTABLE is then searched to find the proper one. This search must be successful, as the PTABLE holds all permutations!
6. Now the computer looks for two Os (human) in a row, column, or diagonal. If this condition is found, the

computer "blocks" by putting an X in the proper space, displays the screen, and goes to step 8; otherwise, it continues.

7. At this point, no "block" was possible. The computer now "draws a ball" by randomly choosing one of the space cells from the PTABLE entry. It records the space cell number, puts an X in the array, displays the new array on the screen, and continues.
8. Now the computer inputs the human choice. It checks for the proper choice, displays the new array, and analyzes the new configuration. If the human wins, it records the win and goes to step 9. If there is no win (there cannot be a draw), the computer goes back to step 2.
9. This is the "end of game" step. The computer has a record of all configurations and which space cells were chosen for the computer's move. If the game was a win, it adds 3 "balls" to each of the space cells by increasing the count by three. This may require three adds for a five move game, four adds for a seven move game, or five adds for a nine move game. If the game was a loss, one "ball" is subtracted from the each of the space cells. If the game was a draw, one "ball" is added to each of the space cells. The computer now records the win/lose/draw for the record, and goes back to step 1 for a new game.

There is one slight addition to the above algorithm. If in step 7, all space cells were found to contain zeroes, the computer **concedes** and zeroes the **previous** space cell that resulted in the current permutation. In this case, the current configuration is considered so hopeless that it is discarded. This condition will rarely, if ever, happen.

Implementation

Modules

Tic-tac-toe is implemented as a series of five levels of modules as shown in Figure 14-16. The top levels of modules are the main **drivers** of the program, while the bottom level has the most rudimentary subroutines of the program.

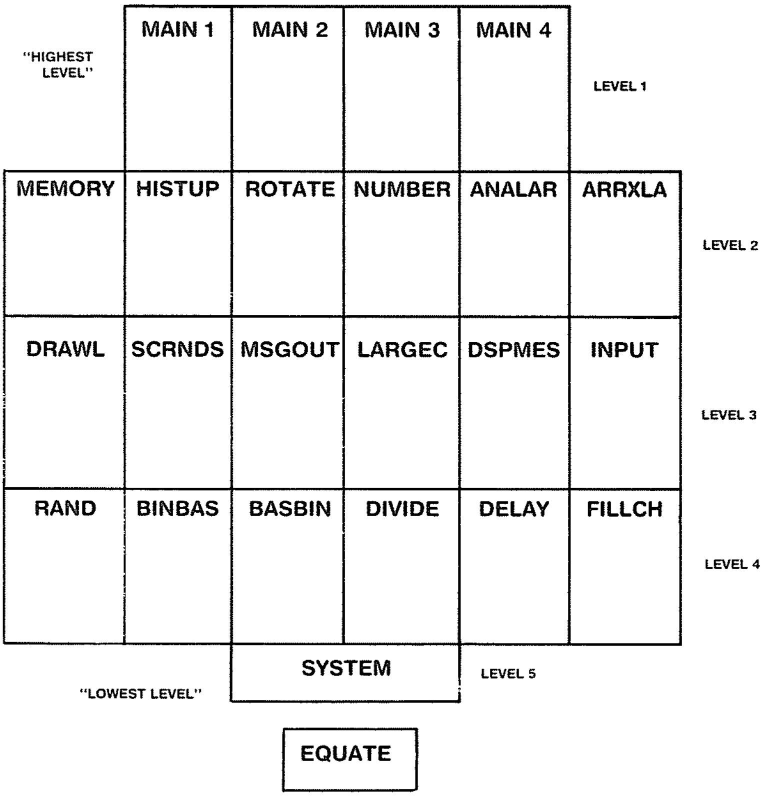


Figure 14-16. Tic-Tac-Toe Modules

As in the previous MORG program, each module is dedicated to a function applicable to the tic-tac-toe program or to a general application usable in programs such as division.

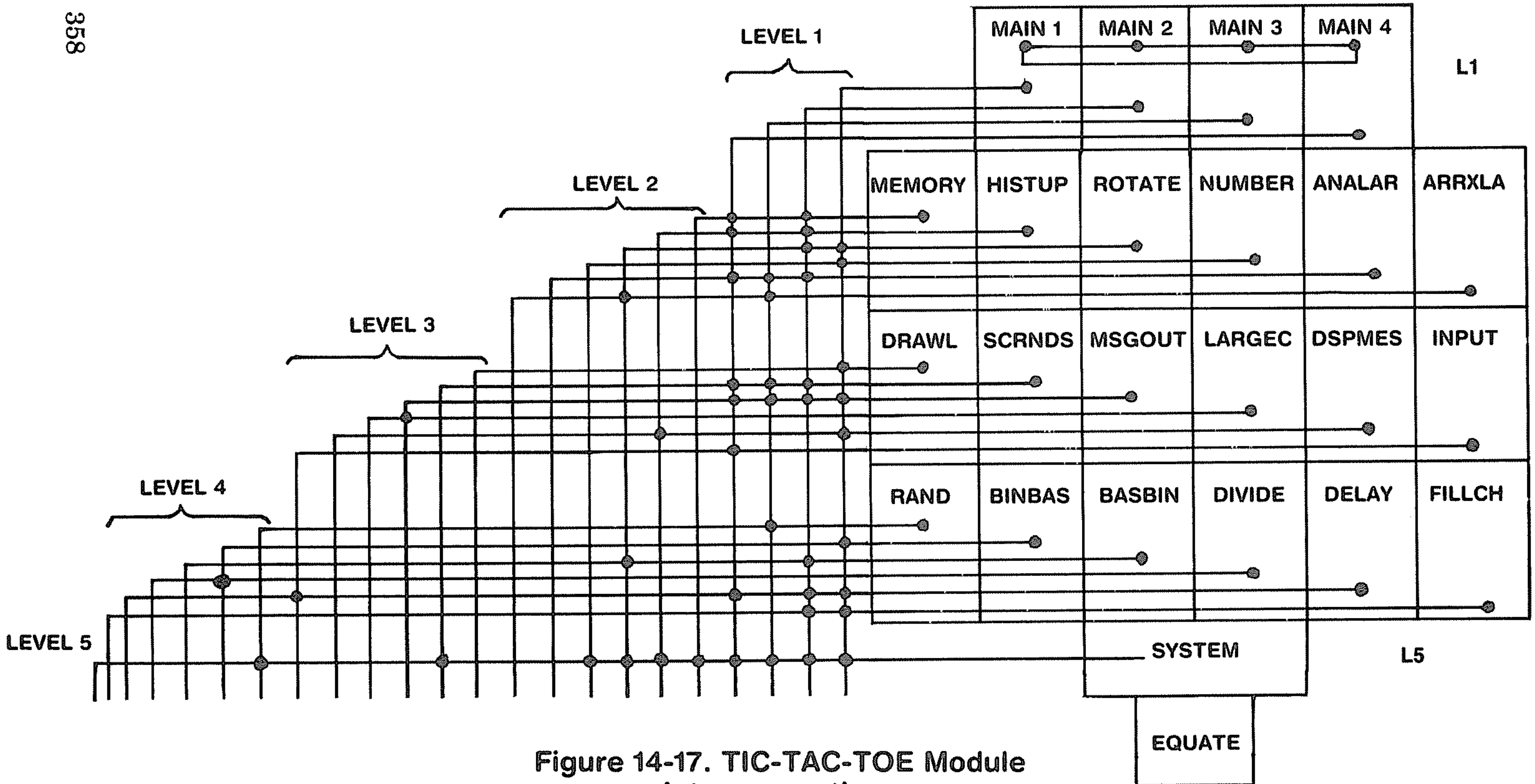


Figure 14-17. TIC-TAC-TOE Module Interconnections

Figure 14-17 shows the module interconnections. Each module generally calls a lower-level module by a CALL, with a set of parameters in the CPU registers. Communication is also done by the tables and variables in the SYSTEM module, which are commonly accessed by many different modules.

Hints and Kinks 14-3
Notes on Figure 14-17

Here, as in Figure 13-7, it's interesting to look at the interconnections to see how the modules relate to each other. Most of the action takes place in the level one modules, which call many of the other modules from all levels. One reason for this is that this program has heavy processing taking place in ''main-line'' code of MAIN1 through MAIN4.

The flow of the program is sequential in these modules and couldn't easily be broken down functionally into lower-level subroutines. There's very little downward communication from level two to other modules.

To find the modules called by any specific module, follow the horizontal line from the module to the extreme left. When the line turns downward, read the lower-level module called by referencing the connection dots. The arrangement of the modules is duplicated in the program listing. Higher-level modules appear first, followed by lower-level modules.

Tables, Buffers, and Variables

Refer to the tic-tac-toe listings, Figure 14-18. There is one listing for each module. These were assembled on the Disk Editor/Assembler, but could have just as easily been done on EDTASM as one large program, providing memory requirements were not a factor in your system.

```
                                TITLE   EQUATE
                                ENTRY   SCREEN,SCARRY,LINE13,LASTL
                        ;*************************************************
                        ;*                 SYSTEM EQUATES                *
                        ;*************************************************
                        ;
3C00                    SCREEN  EQU     3C00H           ;START OF VIDEO DISPLAY
3C3C                    LASTL   EQU     3C00H+60        ;LAST LARGE CHAR POS'N
3CD3                    SCARRY  EQU     3C00H+211       ;FIRST SCREEN ELEMENT ADDR
3F40                    LINE13  EQU     3C00H+832       ;LINE 13
                                END
                                TITLE   MAIN1
                                ENTRY   ARTIP
                                EXT     SCREEN,SCARRY,LINE13
                                EXT     NUMBER,ANALAR,ROTATE,ARRXLA,RAND,INPUT
                                EXT     FILLCH,DSPMES,DRAWL,MSGOUT,DELAY,BINBAS
                                EXT     SCRNDS,HISTUP,MEMORY,BASBIN
                                EXT     GRIDTB,MOVETB,HISTRY,ARRAY1,ARRAY2,ROTTAB
                                EXT     ANALTB,RINDT,RINDTR
                                EXT     MOVENO,FRSTF,NXTHIS,PTABLE,RINDW,NOPTE
                                EXT     NOX,NOO,NOSP,ROTPTR
                                EXT     MSG1,MSG2,MSG3,MSG4,MSG5,MSG6,MSG7,MSG8,MSG9
                                EXT     LASTL,MSG10,MSG11,STACK
                        ;*************************************************
                        ;*         TIC-TAC-TOE LEARNING PROGRAM          *
                        ;*************************************************
                        ;
                        ;************START OR RESTART ENTRY POINT*************
                        ;
0000'   31 0000*        ARTIP:  LD      SP,STACK        ;SET STACK
0003'   3E 80                   LD      A,80H           ;GRAPHICS ALL OFF
0005'   11 0000*                LD      DE,SCREEN       ;START OF SCREEN
0008'   01 0340                 LD      BC,832          ;13 LINES
000B'   CD 0000*                CALL    FILLCH          ;FILL DISPLAY AREA
000E'   21 0000*                LD      HL,MSG1         ;HISTORY MESSAGE
0011'   01 0000*                LD      BC,LINE13       ;START OF HISTORY AREA
0014'   CD 0000*                CALL    DSPMES          ;DISPLAY HISTORY
                        ; DRAW GRID HERE
0017'   DD 21 0000*             LD      IX,GRIDTB       ;TABLE FOR GRID
001B'   DD 7E 00        ART005: LD      A,(IX)            ;GET CHARACTER
001E'   FE FF                   CP      0FFH               ;TEST FOR TERMINATOR
0020'   28 16                   JR      Z,ART008        ;GO IF DONE
0022'   DD 4E 01                LD      C,(IX+1)        ;LOAD HORIZ/VERT
0025'   DD 46 02                LD      B,(IX+2)        ;LOAD # OF CHAR POSNS
0028'   DD 6E 03                LD      L,(IX+3)        ;START OF LINE, LSB
002B'   DD 66 04                LD      H,(IX+4)        ;START OF LINE, MSB
```

```
002E'  CD 0000*              CALL    DRAWL           ;DRAW LINE
0031'  01 0005               LD      BC,5            ;5 BYTES PER LINE
0034'  DD 09                 ADD     IX,BC           ;POINT TO NEXT LINE
0036'  18 E3                 JR      ART005          ;GO FOR NEXT LINE
                      ; ZERO MOVE TABLE
0038'  AF             ART008: XOR    A               ;0
0039'  11 0000*              LD      DE,MOVETB       ;MOVE TABLE ADDRESS
003C'  01 0008               LD      BC,8            ;8 BYTES
003F'  CD 0000*              CALL    FILLCH          ;ZERO MOVE TABLE
0042'  32 0000*              LD      (MOVENO),A      ;ZERO MOVE NUMBER
                      ; TEST FOR FIRST TIME ACTIONS
0045'  3A 0000*              LD      A,(FRSTF)       ;GET FIRST TIME FLAG
0048'  B7                    OR      A               ;TEST
0049'  C2 012C'              JP      NZ,MAIN2        ;GO IF NOT FIRST TIME
004C'                        PAGE
                      ;*****************************************************
                      ;*          FIRST TIME ACTIONS HERE                  *
                      ;*****************************************************
                      ;
004C'  3C                    INC     A               ;1 TO A
004D'  32 0000*              LD      (FRSTF),A       ;SET FIRST TIME FLAG
0050'  11 FFFF*              LD      DE,HISTRY-1     ;START OF HISTORY LINE-1
0053'  ED 53 0000*           LD      (NXTHIS),DE     ;INITIALIZE HISTORY POINTER
0057'  13                    INC     DE              ;START OF HISTORY LINE
0058'  3E 20                 LD      A,' '           ;BLANK
005A'  01 0080               LD      BC,128          ;128 BLANKS
005D'  CD 0000*              CALL    FILLCH          ;INITIALIZE HISTORY LINE
0060'  21 0000*              LD      HL,MSG2         ;TITLE MESSAGE
0063'  FD 21 0000*           LD      IY,SCREEN       ;SCREEN START
0067'  CD 0000*              CALL    MSGOUT          ;OUTPUT MESSAGE
006A'  21 0BB8               LD      HL,3000         ;3000 MS
006D'  CD 0000*              CALL    DELAY           ;DELAY 3 SECS
0070'  21 0000*              LD      HL,MSG3         ;"WAIT ONE" MESSAGE
0073'  FD 21 0000*           LD      IY,SCREEN       ;SCREEN START
0077'  CD 0000*              CALL    MSGOUT          ;OUTPUT MESSAGE
                      ; FIRST TIME ACTION: COMPUTE ALL POSSIBLE PERMUTATIONS
                      ; TIC-TAC-TOE GAME (COMPUTER TO MOVE NEXT, COMPUTER
                      ; MOVES FIRST) OTHER THAN THOSE WITH WHICH COMPUTER
                      ; KNOWS WHAT TO DO. COMPUTER KNOWS ENOUGH AT START TO
                      ; COMPLETE A ROW, COLUMN, OR DIAGONAL.
                      ;
007A'  21 0000*              LD      HL,PTABLE       ;ADDRESS OF PERM TABLE
007D'  E5                    PUSH    HL              ;SAVE
007E'  21 0000               LD      HL,0            ;ZERO
0081'  22 0000*              LD      (NOPTE),HL      ;INITIALIZE COUNT OF PERM
```

```
0084'  E5            00900  ART012: PUSH  HL             ;SAVE CURRENT #
0085'  01 4CE3       00910          LD    BC,19683       ;222 222 222 (BASE 3)
0088'  B7            00920          OR    A              ;CLEAR CARRY
0089'  ED 42         00930          SBC   HL,BC          ;TEST FOR END
008B'  CA 0129'      00940          JP    Z,ART060       ;GO IF DONE
008E'  09            00950          ADD   HL,BC          ;RESTORE ORIGINAL #
008F'  DD 21 0000*   00960          LD    IX,ARRAY1      ;POINT TO ARRAY #1
0093'  CD 0000*      00970          CALL  BINBAS         ;CONVERT TO BASE 3
0096'  22 0000*      00980          LD    (ROTTAB),HL    ;SAVE IN FIRST ENTRY
                     00990  ; COUNT NUMBER OF XS AND OS IN ARRAY. IF NOT EQUAL
                     01000  ; DISCARD THIS PERMUTATION.
0099'  21 0000*      01010          LD    HL,ARRAY1      ;ARRAY ADDRESS
009C'  CD 0000*      01020          CALL  NUMBER         ;FIND TOTALS
009F'  3A 0000*      01030          LD    A,(NOX)        ;GET # OF XS
00A2'  57            01040          LD    D,A            ;PUT IN D
00A3'  3A 0000*      01050          LD    A,(NOO)        ;GET # OF OS
00A6'  BA            01060          CP    D              ;EQUAL TO # OF OS?
00A7'  28 03         01070          JR    Z,ART015       ;GO IF YES
00A9'  E1            01080  ART014: POP   HL             ;GET #
00AA'  18 79         01090          JR    ART050         ;TRY NEXT
                     01100  ; # OF XS - # OF OS HERE. NOW TEST IF # XS = # YS = 4.
                     01110  ; COMPUTER ALWAYS KNOWS WHAT TO DO ON LAST MOVE!
00AC'  FE 04         01120  ART015: CP    4              ;IS THIS LAST MOVE
00AE'  28 F9         01130          JR    Z,ART014       ;GO IF YES
                     01140  ; NOW TEST FOR ALL XS OR ALL YS IN A ROW, COLUMN, OR
                     01150  ; DIAGONAL. DISCARD THIS PERMUTATION IF GAME HAS ALREADY
                     01160  ; BEEN WON!
                     01170  ; ALSO TEST FOR TWO XS IN A ROW, COLUMN, OR DIAGONAL.
                     01180  ; COMPUTER KNOWS WHAT TO DO HERE!
00B0'  DD 21 0000*   01190          LD    IX,ARRAY1      ;ADDRESS OF ARRAY
00B4'  CD 0000*      01200          CALL  ANALAR         ;ANALYZE ARRAY
00B7'  06 08         01210          LD    B,8            ;8 ROWS, COLS, DIAGS
00B9'  21 0000*      01220          LD    HL,ANALTB      ;ADDRESS OF TABLE
00BC'  7E            01230  ART020: LD    A,(HL)           ;GET VALUE
00BD'  FE 03         01240          CP    3                ;ALL XS (1+1+1=3)?
00BF'  CA 00A9'      01250          JP    Z,ART014         ;GO IF YES
00C2'  FE 0C         01260          CP    12               ;ALL OS (4+4+4=12)?
00C4'  CA 00A9'      01270          JP    Z,ART014         ;GO IF YES
00C7'  FE 02         01280          CP    2                ;TWO XS, ONE BLANK?
00C9'  CA 00A9'      01290          JP    Z,ART014         ;GO IF YES
00CC'  23            01300          INC   HL               ;BUMP POINTER
00CD'  10 ED         01310          DJNZ  ART020           ;GO FOR ALL R,C,D
                     01320  ; TO CUT DOWN ON STORAGE REQUIREMENTS AND TO MAKE FOR
                     01330  ; FASTER LEARNING, ROTATE ARRAY 7 WAYS, MAKING TOTAL
                     01340  ; OF 8 POSITIONS.
```

```
00CF'   CD 0000*              CALL ROTATE             ;ROTATE
                      ; NOW HAVE LOWEST VALUE IN DE
00D2'   E1                    POP     HL              ;GET ORIGINAL
00D3'   E5                    PUSH    HL              ;SAVE
00D4'   B7                    OR      A               ;CLEAR CARRY
00D5'   ED 52                 SBC     HL,DE           ;COMPARE
00D7'   20 D0                 JR      NZ,ART014       ;IF NOT LOWEST, DISCARD
                      ; WHEW! FINALLY HAVE VALUE IN HL OF ALLOWABLE PERMUTATION.
00D9'   E1                    POP     HL              ;GET #
00DA'   DD E1                 POP     IX              ;GET NEXT TABLE ADDRESS
00DC'   DD 75 00              LD      (IX),L          ;STORE LS BYTE
00DF'   DD 74 01              LD      (IX+1),H        ;STORE MS BYTE
                      ;*****WARNING*****  OP CODE FOR LD  IY,(  ) SHOULD
                      ;BE FD! SOME ASSEMBLER VERSIONS ERRONEOUSLY OUTPUT DD!
00E2'   DD 2A 0000*           LD      IY,(NOPTE)      ;GET # PERM TABLE ENTRIES
00E6'   FD 23                 INC     IY              ;BUMP BY 1
00E8'   FD 22 0000*           LD      (NOPTE),IY      ;STORE
                      ; NOW FIND NUMBER OF SPACES IN PERMUTATION
00EC'   3A 0000*              LD      A,(NOSP)        ;GET COUNT
00EF'   4F                    LD      C,A             ;PUT IN C
00F0'   DD 71 02              LD      (IX+2),C        ;STORE
00F3'   DD 23                 INC     IX
00F5'   DD 23                 INC     IX
00F7'   DD 23                 INC     IX              ;PNTR+3
00F9'   DD E5                 PUSH    IX              ;START
00FB'   D1                    POP     DE              ;TO DE
00FC'   06 00                 LD      B,0             ;COUNT NOW IN BC
00FE'   DD 09                 ADD     IX,BC           ;PNTR+# SPACES+3
0100'   DD E5                 PUSH    IX              ;SAVE FOR NEXT STORE
0102'   79                    LD      A,C             ;# SPACES
0103'   CB 3F                 SRL     A               ;#/2
0105'   D5                    PUSH    DE              ;SAVE START
0106'   CD 0000*              CALL    FILLCH          ;INITIALIZE SPACE COUNTS
0109'   3A 0000*              LD      A,(NOPTE)       ;GET # OF ENTRIES
010C'   E6 01                 AND     1               ;GET LAST BIT
010E'   21 0000*              LD      HL,MSG10        ;SPACE
0111'   28 03                 JR      Z,ART040        ;GO IF SPACE
0113'   21 0000*              LD      HL,MSG11        ;DASH
0116'   FD 21 0000*   ART040: LD      IY,LASTL        ;LAST CP OF LINE
011A'   CD 0000*              CALL    MSGOUT          ;BLINK
011D'   DD E1                 POP     IX              ;RETRIEVE START
011F'   DD 6E FD              LD      L,(IX-3)        ;GET #
0122'   DD 66 FE              LD      H,(IX-2)
0125'   23            ART050: INC     HL              ;BUMP NUMBER
0126'   C3 0084'              JP      ART012          ;CONTINUE
0129'   E1            ART060: POP     HL              ;DISCARD NUMBER
```

```
012A'   DD E1                    POP     IX              ;TABLE POINTER
                         ;*******************************************************
                         ;*              MAIN2 MODULE FOLLOWS HERE              *
                         ;*******************************************************
012C'                    MAIN2   EQU     $               ;DUMMY
                                 END
                                 TITLE   MAIN2
                                 ENTRY   MAINLP
                                 EXT     SCREEN,SCARRY,LINE13
                                 EXT     NUMBER,ANALAR,ROTATE,ARRXLA,RAND,INPUT
                                 EXT     FILLCH,DSPMES,DRAWL,MSGOUT,DELAY,BINBAS
                                 EXT     SCRNDS,HISTUP,MEMORY,BASBIN
                                 EXT     GRIDTB,MOVETB,HISTRY,ARRAY1,ARRAY2,ROTTAB
                                 EXT     ANALTB,RINDT,RINDTR
                                 EXT     MOVENO,FRSTF,NXTHIS,PTABLE,RINDW,NOPTE
                                 EXT     NOX,NOO,NOSP,ROTPTR
                                 EXT     MSG1,MSG2,MSG3,MSG4,MSG5,MSG6,MSG7,MSG8,MSG9
                                 EXT     MAINEA
                         ;*******************************************************
                         ;*           ENTRY IF NOT FIRST TIME                   *
                         ;*******************************************************
0000'   21 0000*         ART070: LD      HL,MSG2         ;TITLE MESSAGE
0003'   FD 21 0000*              LD      IY,SCREEN       ;SCREEN START
0007'   CD 0000*                 CALL    MSGOUT          ;OUTPUT MESSAGE
000A'   21 0BB8                  LD      HL,3000         ;3000 MS
000D'   CD 0000*                 CALL    DELAY           ;DELAY 3 SECS
0010'   01 0009                  LD      BC,9            ;FOR 9 ELEMENT
0013'   AF                       XOR     A               ;ZERO
0014'   11 0000*                 LD      DE,ARRAY1       ;ARRAY START
0017'   CD 0000*                 CALL    FILLCH          ;ZERO ARRAY
                         ;*******************************************************
                         ;*                   MAIN LOOP HERE                    *
                         ;*******************************************************
001A'                    MAINLP  EQU     $
001A'   21 0000*                 LD      HL,MOVENO       ;MOVE # ADDRESS
001D'   34                       INC     (HL)            ;BUMP MOVE #
                         ; FIRST SEE IF OBVIOUS COMPUTER MOVE - COMPLETING A
                         ; ROW, COLUMN, OR DIAGONAL
001E'   DD 21 0000*      ART100: LD      IX,ARRAY1       ;ARRAY ADDRESS
0022'   CD 0000*                 CALL    ANALAR          ;ANALYZE
0025'   21 0000*                 LD      HL,ANALTB       ;START OF ROW, COL, DIAG
0028'   06 08                    LD      B,8             ;8 VALUES
002A'   7E               ART101: LD      A,(HL)            ;GET VALUE
002B'   FE 02                    CP      2                 ;TEST FOR 2 XS
002D'   28 05                    JR      Z,ART102          ;GO IF 2 XS
```

```
002F'  23                     INC     HL              ;BUMP POINTER
0030'  10 F8                  DJNZ    ART101          ;CONTINUE IF NOT 8
0032'  18 3F                  JR      ART109         ;NO COMPLETION POSSIBLE
                      ; COMPUTER WINS! COMPLETE THE ROW, COLUMN, OR DIAGONAL
                      ; AND END GAME
0034'  E5             ART102: PUSH    HL
0035'  21 0000*               LD      HL,ARRAY1      ;ARRAY ADDRESS
0038'  7E             ART103: LD      A,(HL)          ;GET ELEMENT
0039'  B7                     OR      A               ;TEST FOR 0
003A'  28 03                  JR      Z,ART105        ;GO IF SPACE
003C'  23             ART104: INC     HL              ;BUMP POINTER
003D'  18 F9                  JR      ART103          ;TRY NEXT
003F'  E5             ART105: PUSH    HL              ;SAVE ELEMENT ADDRESS
0040'  3C                     INC     A               ;1 TO A
0041'  77                     LD      (HL),A          ;STORE X
0042'  DD 21 0000*            LD      IX,ARRAY1       ;ARRAY ADDRESS
0046'  CD 0000*               CALL    ANALAR          ;ANALYZE AGAIN
0049'  E1                     POP     HL              ;ELEMENT ADDRESS
004A'  DD E1                  POP     IX              ;VALUE ADDRESS
004C'  DD 7E 00               LD      A,(IX)          ;GET NEW TOTAL
004F'  FE 03                  CP      3               ;COMPLETE?
0051'  28 06                  JR      Z,ART106        ;GO IF YES
0053'  AF                     XOR     A               ;ZERO
0054'  77                     LD      (HL),A          ;ZERO ELEMENT AGAIN
0055'  DD E5                  PUSH    IX              ;SAVE VALUE ADDRESS
0057'  18 E3                  JR      ART104          ;MUST BE FOUND!
0059'  CD 0000*       ART106: CALL    SCRNDS         ;UPDATE SCREEN
005C'  3E 57                  LD      A,'W'          ;WIN
005E'  CD 0000*               CALL    HISTUP         ;HISTORY UPDATE
0061'  3E 03                  LD      A,3            ;3
0063'  CD 0000*               CALL    MEMORY         ;ADJUST CELLS
0066'  21 0000*               LD      HL,MSG8        ;"I WIN!" MESSAGE
0069'  FD 21 0000*            LD      IY,SCREEN      ;SCREEN MESSAGE AREA
006D'  CD 0000*               CALL    MSGOUT         ;DISPLAY
0070'  C3 0000*               JP      MAINEA         ;GO FOR END ACTION
                      ; NOW TEST FOR LAST MOVE
0073'  3A 0000*       ART109: LD      A,(MOVENO)     ;GET # OF MOVE
0076'  FE 05                  CP      5              ;IS IT LAST?
0078'  20 3A                  JR      NZ,ART114      ;GO IF NO
                      ; LAST MOVE HERE
007A'  21 0000*               LD      HL,ARRAY1      ;ARRAY ADDRESS
007D'  7E             ART110: LD      A,(HL)          ;GET ELEMENT
007E'  B7                     OR      A               ;TEST
007F'  28 03                  JR      Z,ART111        ;GO IF SPACE
0081'  23                     INC     HL              ;BUMP POINTER
0082'  18 F9                  JR      ART110          ;CONTINUE
```

```
0084'  3C                ART111: INC     A                 ;1=X
0085'  77                        LD      (HL),A            ;STORE X
0086'  DD 21 0000*               LD      IX,ARRAY1         ;ARRAY ADDRESS
008A'  CD 0000*                  CALL    ANALAR            ;ANALYZE
008D'  21 0000*                  LD      HL,ARRAY1         ;ARRAY ADDRESS
0090'  06 08                     LD      B,8               ;8 VALUES
0092'  7E                ART112: LD      A,(HL)              ;GET VALUES
0093'  FE 03                     CP      3                   ;TEST FOR WIN
0095'  28 C2                     JR      Z,ART106            ;GO IF WIN
0097'  23                        INC     HL                  ;BUMP POINTER
0098'  10 F8                     DJNZ    ART112              ;GO IF NOT ALL 8
                         ; DRAW HERE
009A'  CD 0000*                  CALL    SCRNDS            ;UPDATE SCREEN
009D'  3E 44                     LD      A,'D'             ;DRAW
009F'  CD 0000*                  CALL    HISTUP            ;HISTORY UPDATE
00A2'  3E 01                     LD      A,1               ;FOR DRAW
00A4'  CD 0000*                  CALL    MEMORY            ;ADJUST CELLS
00A7'  21 0000*                  LD      HL,MSG9           ;"DRAW!" MESSAGE
00AA'  FD 21 0000*               LD      IY,SCREEN         ;SCREEN MESSAGE
00AE'  CD 0000*                  CALL    MSGOUT            ;DISPLAY
00B1'  C3 0000*                  JP      MAINEA            ;GO FOR END ACTION
00B4'  FD 21 0000*       ART114: LD      IY,ARRAY1         ;POINT TO ARRAY
00B8'  CD 0000*                  CALL    BASBIN            ;CONVERT TO BASE 3 NUMBER
00BB'  22 0000*                  LD      (ROTTAB),HL       ;ORIGINAL
00BE'  CD 0000*                  CALL    ROTATE            ;FIND 7 ROTATE VALUES
                         ; SMALLEST VALUE IN DE. SEARCH PERMUTATION TABLE FOR VALUE
                         ; MUST BE FOUND!
00C1'  DD 21 0000*               LD      IX,PTABLE         ;START OF PERM TABLE
00C5'  DD 6E 00          ART120: LD      L,(IX)              ;GET ENTRY
00C8'  DD 66 01                  LD      H,(IX+1)
00CB'  B7                        OR      A                   ;CLEAR CARRY
00CC'  ED 52                     SBC     HL,DE               ;TEST FOR EQUAL
00CE'  28 0C                     JR      Z,ART125            ;GO IF FOUND
00D0'  DD 4E 02                  LD      C,(IX+2)            ;GET # OF SPACES
00D3'  06 00                     LD      B,0                 ;NOW IN BC
00D5'  03                        INC     BC                  ;BYPASS HEADER
00D6'  03                        INC     BC
00D7'  03                        INC     BC
00D8'  DD 09                     ADD     IX,BC               ;POINT TO NEXT ENTRY
00DA'  18 E9                     JR      ART120              ;GO FOR NEXT COMPARISON
                         ;****************************************************
                         ;*              MAIN3 MODULE FOLLOWS HERE           *
                         ;****************************************************
00DC'                    ART125  EQU     $
                                 END
```

```
                                 TITLE   MAIN3
                                 EXT     SCREEN,SCARRY,LINE13
                                 EXT     NUMBER,ANALAR,ROTATE,ARRXLA,RAND,INPUT
                                 EXT     FILLCH,DSPMES,DRAWL,MSGOUT,DELAY,BINBAS
                                 EXT     SCRNDS,HISTUP,MEMORY,BASBIN
                                 EXT     GRIDTB,MOVETB,HISTRY,ARRAY1,ARRAY2,ROTTAB
                                 EXT     ANALTB,RINDT,RINDTR
                                 EXT     MOVENO,FRSTF,NXTHIS,PTABLE,RINDW,NOPTE
                                 EXT     NOX,NOO,NOSP,ROTPTR
                                 EXT     MSG1,MSG2,MSG3,MSG4,MSG5,MSG6,MSG7,MSG8,MSG9
                                 EXT     MAINLP,ARTIP,MSG12,MAINE1
                        ; ENTRY FOUND IN PERMUTATION TABLE HERE
0000'   DD E5           ARTO00:  PUSH    IX              ;SAVE LOCATION
0002'   2A 0000#                 LD      HL,(ROTPTR)     ;GET ADDRESS OF ROTATION
0005'   01 0000#                 LD      BC,ROTTAB       ;ROTATION TABLE ADDR
0008'   B7                       OR      A               ;CLEAR CARRY
0009'   ED 42                    SBC     HL,BC           ;FIND DISPLACEMENT
000B'   CB 3D                    SRL     L               ;SHIFT LS BYTE
000D'   CB 3C                    SRL     H               ;SHIFT MS BYTE
000F'   30 02                    JR      NC,ART010       ;GO IF NO CARRY
0011'   CB FD                    SET     7,L             ;SET MS BIT
0013'   E5              ART010:  PUSH    HL              ;D#1
0014'   29                       ADD     HL,HL           ;D#2
0015'   29                       ADD     HL,HL           ;D#4
0016'   29                       ADD     HL,HL           ;D#8
0017'   C1                       POP     BC              ;D#1
0018'   09                       ADD     HL,BC           ;D#9
0019'   E5                       PUSH    HL              ;SAVE D
001A'   01 0000#                 LD      BC,RINDT        ;ROTATE INDICES TABLE
001D'   09                       ADD     HL,BC           ;POINT TO INDICES
001E'   DD 21 0000#              LD      IX,ARRAY1       ;ORIGINAL ARRAY
0022'   FD 21 0000#              LD      IY,ARRAY2       ;WORKING ARRAY
0026'   CD 0000#                 CALL    ARRXLA          ;TRANSLATE ARRAY
0029'   E1                       POP     HL              ;GET DISPLACEMENT
002A'   01 0000#                 LD      BC,RINDTR       ;ROTATE INDICES TABLE, REV
002D'   09                       ADD     HL,BC           ;POINT TO INVERSE INDICES
002E'   22 0000#                 LD      (RINDW),HL      ;SAVE
                        ; SEE IF COMPUTER "BLOCKING" MOVE - TWO OS IN ROW
                        ; COLUMN, OR DIAGONAL
0031'   DD 21 0000#              LD      IX,ARRAY2       ;ARRAY ADDRESS
0035'   CD 0000#                 CALL    ANALAR          ;ANALYZE
0038'   21 0000#                 LD      HL,ANALTB       ;START OF ROW, COL, DIAG
003B'   06 08                    LD      B,8             ;8 VALUES
003D'   7E              ART101:  LD      A,(HL)          ;GET VALUE
003E'   FE 08                    CP      8               ;TEST FOR TWO OS
0040'   28 05                    JR      Z,ART102        ;GO IF TWO OS
```

```
0042'  23            00560          INC     HL              ;BUMP POINTER
0043'  10 F8         00570          DJNZ    ART101          ;CONTINUE IF NOT 8
0045'  18 52         00580          JR      ART125         ;NO COMPLETION POSSIBLE
                     00590  ; TWO OS FOUND - PUT IN X FOR BLOCK
0047'  E5            00600  ART102: PUSH    HL             ;SAVE ANALYZE VALUE
0048'  21 0000*      00610          LD      HL,ARRAY2      ;ARRAY ADDRESS
004B'  7E            00620  ART103: LD      A,(HL)          ;GET ELEMENT
004C'  B7            00630          OR      A               ;TEST FOR 0
004D'  28 03         00640          JR      Z,ART105        ;GO IF SPACE
004F'  23            00650  ART104: INC     HL              ;BUMP PNTR
0050'  18 F9         00660          JR      ART103          ;TRY NEXT
0052'  E5            00670  ART105: PUSH    HL              ;SAVE ELEMENT ADDRESS
0053'  3C            00680          INC     A               ;1 TO A
0054'  77            00690          LD      (HL),A          ;STORE X
0055'  DD 21 0000*   00700          LD      IX,ARRAY2       ;ARRAY ADDRESS
0059'  CD 0000*      00710          CALL    ANALAR          ;ELEMENT ADDRESS
005C'  E1            00720          POP     HL              ;ELEMENT ADDRESS
005D'  DD E1         00730          POP     IX              ;VALUE ADDRESS
005F'  DD 7E 00      00740          LD      A,(IX)          ;GET NEW TOTAL
0062'  FE 09         00750          CP      9               ;COMPLETE?
0064'  28 06         00760          JR      Z,ART106        ;GO IF YES
0066'  AF            00770          XOR     A               ;ZERO
0067'  77            00780          LD      (HL),A          ;ZERO ELEMENT AGAIN
0068'  DD E5         00790          PUSH    IX              ;FOR LOOP
006A'  18 E3         00800          JR      ART104          ;MUST BE FOUND!
006C'  AF            00810  ART106: XOR     A              ;ZERO A
006D'  77            00820          LD      (HL),A         ;ZERO FOR NEXT PROCESSING
006E'  E5            00830          PUSH    HL             ;SAVE ELEMENT ADDRESS
006F'  01 FFFF*      00840          LD      BC,ARRAY2-1    ;ARRAY ADDRESS-1
0072'  B7            00850          OR      A              ;CLEAR C
0073'  ED 42         00860          SBC     HL,BC          ;INDEX OF ELEMENT+1
                     00870  ; HL NOW CONTAINS INDEX OF "BLOCK" ELEMENT
0075'  11 FFFF       00880          LD      DE,-1          ;COUNT
0078'  45            00890          LD      B,L            ;INDEX+1 TO B
0079'  21 0000*      00900          LD      HL,ARRAY2      ;ADDRESS OF ARRAY
007C'  7E            00910  ART108: LD      A,(HL)          ;GET ELEMENT
007D'  23            00920          INC     HL              ;BUMP TO NEXT
007E'  B7            00930          OR      A               ;TEST FOR ZERO
007F'  20 01         00940          JR      NZ,ART110       ;BYPASS IF NOT ZERO
0081'  13            00950          INC     DE              ;BUMP COUNT OF SPACES
0082'  10 F8         00960  ART110: DJNZ    ART108         ;COUNT DOWN IF ZERO
                     00970  ; DE NOW HAS INDEX OF ELEMENT CELL
0084'  E1            00980          POP     HL             ;GET ADDRESS OF ELEMENT
0085'  3E 01         00990          LD      A,1            ;1 TO A
0087'  77            01000          LD      (HL),A         ;STORE X IN ARRAY2
0088'  DD E1         01010          POP     IX             ;GET PERM TABLE ADDRESS
```

```
008A'   DD 19                 ADD   IX,DE           ;IX NOW PNTS TO SP CELL-3
008C'   DD 23                 INC   IX
008E'   DD 23                 INC   IX
0090'   DD 23                 INC   IX
0092'   DD 23                 INC   IX              ;NOW SP CELL ADD+1
0094'   DD E5                 PUSH  IX              ;SAVE FOR NEXT PROCESSING
0096'   C3 0116'              JP    ART172          ;NOW GO TO MOVE, DISPLAY
                ; FIND TOTAL COUNT IN ALL SPACES
0099'   DD E1         ART125: POP   IX              ;GET PERM TABLE ADDR
009B'   DD E5                 PUSH  IX              ;SAVE
009D'   DD 46 02      ART130: LD    B,(IX+2)        ;GET # OF SPACES
00A0'   21 0000               LD    HL,0            ;CLEAR TOTAL
00A3'   DD 5E 03      ART135: LD    E,(IX+3)          ;GET COUNT
00A6'   16 00                 LD    D,0               ;NOW IN DE
00A8'   19                    ADD   HL,DE             ;SUBTOTAL IN HL
00A9'   DD 23                 INC   IX                ;BUMP TO NEXT ENTRY
00AB'   10 F6                 DJNZ  ART135            ;GO IF NOT END
00AD'   7C                    LD    A,H             ;GET COUNT MSB
00AE'   B5                    OR    L               ;MERGE LSB
00AF'   DD E1                 POP   IX              ;GET POINTER
00B1'   DD E5                 PUSH  IX              ;SAVE
00B3'   20 25                 JR    NZ,ART140       ;GO IF NOT ZERO
                ; ALL SPACE CELLS OF THIS ENTRY ARE 0. REMOVE THIS
                ; ENTRY BY ZEROING PREVIOUS LINK AND CONCEDE.
00B5'   3A 0000*              LD    A,(MOVENO)      ;GET MOVE #
00B8'   3D                    DEC   A               ;FIND LAST MOVE INDEX
00B9'   07                    RLCA                  ;INDEX*2
00BA'   4F                    LD    C,A             ;NOW IN C
00BB'   06 00                 LD    B,0             ;NOW IN BC
00BD'   DD 21 0000*           LD    IX,MOVETB       ;MOVE TABLE
00C1'   DD 09                 ADD   IX,BC           ;POINT TO LINK ADDRESS
00C3'   DD 66 00              LD    H,(IX)          ;MS BYTE
00C6'   DD 6E 01              LD    L,(IX+1)        ;LS BYTE
00C9'   AF                    XOR   A               ;ZERO
00CA'   77                    LD    (HL),A          ;ZERO LINK
00CB'   21 0000*              LD    HL,MSG12        ;"I CONCEDE" MESSAGE
00CE'   FD 21 0000*           LD    IY,SCREEN       ;MESSAGE AREA
00D2'   CD 0000*              CALL  MSGOUT          ;DISPLAY
00D5'   3E 43                 LD    A,'C'           ;C FOR CONCEDE
00D7'   C3 0000*              JP    MAINE1          ;GO FOR END ACTION
00DA'   CD 0000*      ART140: CALL  RAND            ;GET RANDOM #
00DD'   C5                    PUSH  BC              ;TRANSFER TO DE
00DE'   D1                    POP   DE
00DF'   EB                    EX    DE,HL           ;RAND# IN HL, COUNT IN DE
00E0'   B7            ART150: OR    A                 ;CLEAR CARRY
00E1'   ED 52                 SBC   HL,DE             ;RAND#-COUNT
```

```
00E3'  30 FB          01480            JR     NC,ART150       ;CONTINUE TILL NEGATIVE
00E5'  19             01490            ADD    HL,DE          ;GET # LE COUNT
                      01500   ; NOW HAVE # LESS THAN OR EQUAL TO TOTAL COUNT IN SPACE
                      01510   ; CELLS. USE THIS NUMBER TO FIND A RANDOM SPACE IN WHICH
                      01520   ; TO PUT AN X.
00E6'  DD 4E 03       01530   ART160:  LD     C,(IX+3)        ;GET COUNT
00E9'  06 00          01540            LD     B,0             ;NOW IN BC
00EB'  B7             01550            OR     A               ;CLEAR CARRY
00EC'  ED 42          01560            SBC    HL,BC           ;#-COUNT
00EE'  DD 23          01570            INC    IX              ;BUMP POINTER
00F0'  28 02          01580            JR     Z,ART165        ;GO IF 0
00F2'  30 F2          01590            JR     NC,ART160       ;GO IF NOT NEGATIVE
                      01600   ; CHOOSE THIS SPACE CELL FOR AN X
00F4'  D1             01610   ART165:  POP    DE             ;GET START
00F5'  DD E5          01620            PUSH   IX             ;TRANSFER CELL ADDRESS PNTR
00F7'  E1             01630            POP    HL
00F8'  01 0003        01640            LD     BC,3           ;ADJUSTMENT
00FB'  DD 09          01650            ADD    IX,BC          ;POINT TO CELL
00FD'  DD E5          01660            PUSH   IX             ;CELL ADDRESS+1
00FF'  B7             01670            OR     A              ;CLEAR CARRY
0100'  ED 52          01680            SBC    HL,DE          ;FIND # OF CELLS+4
0102'  45             01690            LD     B,L            ;# OF CELL+1 TO B
0103'  DD 21 0000#    01700            LD     IX,ARRAY2      ;START OF WORKING ARRAY
0107'  DD 7E 00       01710   ART170:  LD     A,(IX)          ;GET ELEMENT
010A'  B7             01720            OR     A               ;TEST FOR 0
010B'  DD 23          01730            INC    IX              ;BUMP POINTER
010D'  20 F8          01740            JR     NZ,ART170       ;GO IF NOT 0
010F'  10 F6          01750            DJNZ   ART170          ;GO IF NOT ELEMENT
0111'  3E 01          01760            LD     A,1             ;X
0113'  DD 77 FF       01770            LD     (IX-1),A        ;STORE X
0116'  3A 0000#       01780   ART172:  LD     A,(MOVENO)     ;COMPUTER MOVE #
0119'  3D             01790            DEC    A              ;MOVE #-1
011A'  07             01800            RLCA                  ;INDEX*2
011B'  5F             01810            LD     E,A            ;NOW IN E
011C'  16 00          01820            LD     D,0            ;NOW IN DE
011E'  21 0000#       01830            LD     HL,MOVETB      ;MOVE TABLE ADDRESS
0121'  19             01840            ADD    HL,DE          ;POINT TO MOVE #
0122'  D1             01850            POP    DE             ;CELL ADDRESS+1
0123'  1B             01860            DEC    DE             ;ADDRESS OF SPACE
0124'  73             01870            LD     (HL),E         ;STORE LSB
0125'  23             01880            INC    HL
0126'  72             01890            LD     (HL),D         ;STORE MSB
                      01900   ; NOW HAVE STORED AN X IN WORKING ARRAY - CONVERT TO
                      01910   ; ACTUAL ARRAY.
0127'  2A 0000#       01920            LD     HL,(RINDW)     ;POINTER TO INDICES LIST
012A'  DD 21 0000#    01930            LD     IX,ARRAY2      ;SOURCE ARRAY
```

```
012E'   FD 21 0000*             LD      IY,ARRAY1       ;DESTINATION ARRAY
0132'   CD 0000*                CALL    ARRXLA          ;TRANSLATE
                        ; NOW CONVERT ARRAY1 TO SCREEN ARRAY
0135'   CD 0000*                CALL    SCRNDS          ;DISPLAY ARRAY
                                END
                                TITLE   MAIN4
                                ENTRY   MAINEA,MAINE1
                                EXT     SCREEN,SCARRY,LINE13
                                EXT     NUMBER,ANALAR,ROTATE,ARRXLA,RAND,INPUT
                                EXT     FILLCH,DSPMES,DRAWL,MSGOUT,DELAY,BINBAS
                                EXT     SCRNDS,HISTUP,MEMORY,BASBIN
                                EXT     GRIDTB,MOVETB,HISTRY,ARRAY1,ARRAY2,ROTTAB
                                EXT     ANALTB,RINDT,RINDTR
                                EXT     MOVENO,FRSTF,NXTHIS,PTABLE,RINDW,NOPTE
                                EXT     NOX,NOO,NOSP,ROTPTR
                                EXT     MSG1,MSG2,MSG3,MSG4,MSG5,MSG6,MSG7,MSG8,MSG9
                                EXT     MAINLP,ARTIP,MSG12
                        ; NOW GET USER INPUT AND DISPLAY
0000'   21 0000*        ART175: LD      HL,MSG4         ;"YOUR MOVE" MESSAGE
0003'   FD 21 0000*             LD      IY,SCREEN       ;MESSAGE AREA
0007'   CD 0000*                CALL    MSGOUT          ;DISPLAY
000A'   CD 0000*                CALL    INPUT           ;GET # 0-8
000D'   4F                      LD      C,A             ;NOW IN C
000E'   06 00                   LD      B,0             ;NOW IN BC
0010'   21 0000*                LD      HL,ARRAY1       ;WORKING ARRAY
0013'   09                      ADD     HL,BC           ;POINT TO ELEMENT
0014'   7E                      LD      A,(HL)          ;GET CONTENTS
0015'   B7                      OR      A               ;TEST FOR ZERO
0016'   28 12                   JR      Z,ART180        ;GO IF EMPTY
0018'   21 0000*                LD      HL,MSG5         ;"TRY AGAIN" MESSAGE
001B'   FD 21 0000*             LD      IY,SCREEN       ;MESSAGE AREA
001F'   CD 0000*                CALL    MSGOUT          ;DISPLAY
0022'   21 0BB8                 LD      HL,3000         ;3000 MILLISECS
0025'   CD 0000*                CALL    DELAY           ;DELAY 3 SECS
0028'   18 D6                   JR      ART175          ;TRY AGAIN
002A'   3E 04           ART180: LD      A,4             ;O
002C'   77                      LD      (HL),A          ;STORE USER O
002D'   CD 0000*                CALL    SCRNDS          ;DISPLAY
                        ; NOW ANALYZE USER INPUT
0030'   DD 21 0000*             LD      IX,ARRAY1       ;ARRAY ADDRESS
0034'   CD 0000*                CALL    ANALAR          ;ANALYZE
0037'   21 0000*                LD      HL,ANALTB       ;ANALYZE TABLE ADDR
003A'   06 08                   LD      B,8             ;8 VALUES
003C'   7E              ART185: LD      A,(HL)            ;GET VALUE
003D'   23                      INC     HL                ;BUMP PNTR
```

```
003E'   FE 0C          00500          CP      12              ;TEST FOR ALL OS
0040'   28 05          00510          JR      Z,ART190        ;GO IF ALL OS
0042'   10 F8          00520          DJNZ    ART185          ;NOT ALL 8
0044'   C3 0000*       00530          JP      MAINLP         ;BACK FOR NEXT MOVE
0047'   21 0000*       00540  ART190: LD      HL,MSG6        ;"YOU WIN" MESSAGE
004A'   FD 21 0000*    00550  ART191: LD      IY,SCREEN      ;MESSAGE AREA
004E'   CD 0000*       00560          CALL    MSGOUT         ;DISPLAY
0051'   3E 4C          00570          LD      A,'L'          ;L FOR LOSE
0053'   CD 0000*       00580  MAINE1: CALL    HISTUP         ;UPDATE HISTORY
0056'   3E FF          00590          LD      A,0FFH         ;-1
0058'   CD 0000*       00600          CALL    MEMORY         ;ADJUST CELLS
                       00610  ;*****************************************************
                       00620  ;*                END ACTION ENTRY POINT             *
                       00630  ;*****************************************************
005B'                  00640  MAINEA  EQU     $              ;DUMMY
005B'   21 0BB8        00650  ART193: LD      HL,3000        ;3000 MILLISEC
005E'   CD 0000*       00660          CALL    DELAY          ;DELAY 3 SECS
0061'   21 0000*       00670          LD      HL,MSG7        ;"ANOTHER?" MESSAGE
0064'   FD 21 0000*    00680          LD      IY,SCREEN      ;MESSAGE AREA
0068'   CD 0000*       00690          CALL    MSGOUT         ;DISPLAY
006B'   CD 0000*       00700          CALL    INPUT          ;GET KEY
006E'   C3 0000*       00710          JP      ARTIP          ;RESTART
                       00720          END
                       00100          TITLE   MEMOSR
                       00110          ENTRY   MEMORY
                       00120          EXT     MOVETB
                       00130  ;*****************************************************
                       00140  ;*                    MEMORY SUBROUTINE              *
                       00150  ;*   FOR WIN, LOSE, OR DRAW, ADJUSTS COUNTS FOR EACH *
                       00160  ;*   SPACE.                                          *
                       00170  ;*   ENTRY:  (A)=-1 FOR LOSE, 1 FOR DRAW, 3 FOR WIN  *
                       00180  ;*   EXIT:   PLAY CELLS ADJUSTED                     *
                       00190  ;*   ALL REGISTERS SAVED EXCEPT A                    *
                       00200  ;*****************************************************
                       00210  ;
0000'   F5             00220  MEMORY: PUSH    AF             ;SAVE REGISTERS
0001'   E5             00230          PUSH    HL
0002'   DD E5          00240          PUSH    IX
0004'   DD 21 0000*    00250          LD      IX,MOVETB      ;MOVE TABLE PNTR
0008'   DD 66 01       00260  MEM005: LD      H,(IX+1)         ;GET POINTER MSB
000B'   DD 6E 00       00270          LD      L,(IX)           ;LSB
000E'   F5             00280          PUSH    AF               ;SAVE A
000F'   7C             00290          LD      A,H              ;TEST FOR HL=0
0010'   B5             00300          OR      L
0011'   28 19          00310          JR      Z,MEM090         ;GO IF LAST MOVE
0013'   F1             00320          POP     AF
```

```
0014'   F5                       PUSH    AF
0015'   86                       ADD     A,(HL)           ;ADJUST COUNT
0016'   CA 001C'                 JP      Z,MEM008        ;GO IF ZERO
0019'   F2 001D'                 JP      P,MEM010        ;GO IF POSITIVE
001C'   AF               MEM008: XOR     A               ;COUNT OF 0
001D'   FE 64            MEM010: CP      100              ;TEST FOR LT 100
001F'   FA 0024'                 JP      M,MEM020         ;GO IF LT 100
0022'   3E 63                    LD      A,99             ;MAX COUNT
0024'   77               MEM020: LD      (HL),A           ;STORE COUNT
0025'   DD 23                    INC     IX               ;BUMP MOVE VARIABLE PNTR
0027'   DD 23                    INC     IX
0029'   F1                       POP     AF
002A'   18 DC                    JR      MEM005           ;CONTINUE
002C'   F1               MEM090: POP     AF              ;RESTORE REGISTERS
002D'   DD E1                    POP     IX
002F'   E1                       POP     HL
0030'   F1                       POP     AF
0031'   C9                       RET                     ;RETURN
                                 END
                                 TITLE   HISTSR
                                 ENTRY   HISTUP
                                 EXT     NXTHIS,HISTRY,MSG1,LINE13,DSPMES
                         ;*************************************************************
                         ;*                  HISTORY UPDATE                           *
                         ;*  UPDATES 128 PRINT POSITIONS OF HISTORY MESSAGE           *
                         ;*  ENTRY:  (A)='L','W', OR 'D' FOR LOSE, WIN, DRAW          *
                         ;*  EXIT:   BUFFER UPDATED                                   *
                         ;*  ALL REGISTERS SAVED                                      *
                         ;*************************************************************
                         ;
0000'   C5               HISTUP: PUSH    BC              ;SAVE REGISTERS
0001'   D5                       PUSH    DE
0002'   E5                       PUSH    HL
0003'   2A 0000*                 LD      HL,(NXTHIS)     ;HISTORY PNTR
0006'   23                       INC     HL              ;POINT TO NEXT
0007'   22 0000*                 LD      (NXTHIS),HL     ;STORE
000A'   01 0080*                 LD      BC,HISTRY+128   ;END OF BUFFER+1
000D'   B7                       OR      A               ;CLEAR CARRY
000E'   ED 42                    SBC     HL,BC           ;TEST FOR END
0010'   20 10                    JR      NZ,HIS010       ;GO IF NOT END
0012'   0B                       DEC     BC              ;REINITIALIZE AT END
0013'   ED 43 0000*              LD      (NXTHIS),BC     ;STORE
0017'   21 0001*                 LD      HL,HISTRY+1     ;SOURCE
001A'   11 0000*                 LD      DE,HISTRY       ;DESTINATION
001D'   01 007F                  LD      BC,127          ;127 BYTES
0020'   ED B0                    LDIR                    ;MOVE
```

```
0022'   2A 0000*       00370  HIS010: LD      HL,(NXTHIS)     ;HISTORY POINTER
0025'   77             00380          LD      (HL),A          ;STORE STATUS
0026'   21 0000*       00390          LD      HL,MSG1         ;HISTORY MESSAGE
0029'   01 0000*       00400          LD      BC,LINE13       ;START OF HISTORY AREA
002C'   CD 0000*       00410          CALL    DSPMES          ;DISPLAY HISTORY
002F'   E1             00420          POP     HL              ;RESTORE REGISTERS
0030'   D1             00430          POP     DE
0031'   C1             00440          POP     BC
0032'   C9             00450          RET
                       00460          END
                       00100          TITLE   ROTASR
                       00110          ENTRY   ROTATE
                       00120          EXT     ARRXLA,BASBIN
                       00130          EXT     RINDT9,ROTTAB,ARRAY1,ARRAY2
                       00140          EXT     ROTPTR
                       00150  ;*************************************************
                       00160  ;*              ROTATE SUBROUTINE                *
                       00170  ;*   FINDS ROTATION BASE 3 VALUES FOR ROTATION OF *
                       00180  ;*   ARRAY1 TO 90, 180, 270 DEGREES AND MIRROR VALUES *
                       00190  ;*   OF 0, 90, 180, 270 DEGREES.                 *
                       00200  ;*   ENTRY:  ARRAY1 HOLDS ORIGINAL               *
                       00210  ;*   EXIT:   (DE)=SMALLEST BASE 3 VALUE          *
                       00220  ;*           8 VALUES IN ROTTAB                  *
                       00230  ;*   ALL REGISTERS SAVED EXCEPT DE               *
                       00240  ;*************************************************
                       00250  ;
0000'   F5             00260  ROTATE: PUSH    AF              ;SAVE REGISTERS
0001'   C5             00270          PUSH    BC
0002'   E5             00280          PUSH    HL
0003'   DD E5          00290          PUSH    IX
0005'   FD E5          00300          PUSH    IY
0007'   06 07          00310          LD      B,7             ;SEVEN ROTATIONS
0009'   DD 21 0000*    00320          LD      IX,RINDT9        ;ROTATE INDICES TABLE
000D'   FD 21 0000*    00330          LD      IY,ROTTAB       ;ROTATE VALUES PNTR
0011'   FD 23          00340          INC     IY              ;POINT TO R90V
0013'   FD 23          00350          INC     IY
0015'   FD E5          00360  ROTO10: PUSH    IY                ;VARIABLE PNTR
0017'   DD E5          00370          PUSH    IX                ;INDICES PNTR
0019'   DD 21 0000*    00380          LD      IX,ARRAY1         ;ORIGINAL
001D'   FD 21 0000*    00390          LD      IY,ARRAY2         ;NEW
0021'   E1             00400          POP     HL                ;LIST OF INDICES
0022'   E5             00410          PUSH    HL                ;SAVE
0023'   CD 0000*       00420          CALL    ARRXLA            ;TRANSLATE
0026'   CD 0000*       00430          CALL    BASBIN            ;TRANSLATE TO #
0029'   DD E1          00440          POP     IX                ;LIST POINTER
002B'   FD E1          00450          POP     IY                ;VARIABLE PNTR
```

```
002D'  FD 75 00              LD     (IY),L          ;STORE LS BYTE
0030'  FD 74 01              LD     (IY+1),H        ;STORE MS BYTE
0033'  FD 23                 INC    IY              ;BUMP VARIABLE PNTR
0035'  FD 23                 INC    IY
0037'  11 0009               LD     DE,9            ;INCREMENT
003A'  DD 19                 ADD    IX,DE           ;POINT TO NEXT SET
003C'  10 D7                 DJNZ   ROTO10          ;GO IF MORE
003E'  06 08                 LD     B,8            ;PLUS ONE FOR 0 DEG
0040'  DD 21 0000*           LD     IX,ROTTAB      ;START OF ROTATE TABLE
0044'  11 7FFF               LD     DE,7FFFH       ;LOWEST VALUE
0047'  DD 66 01     ROTO30:  LD     H,(IX+1)        ;GET FIRST VALUE MSB
004A'  DD 6E 00              LD     L,(IX)          ;LSB
004D'  B7                    OR     A               ;CLEAR CARRY
004E'  ED 52                 SBC    HL,DE           ;COMPARE
0050'  F2 005A'              JP     P,ROTO40       .;GO IF NEW= OR GT OLD
0053'  19                    ADD    HL,DE           ;RESTORE
0054'  54                    LD     D,H             ;STORE NEW MSB
0055'  5D                    LD     E,L             ;STORE NEW LSB
0056'  DD 22 0000*           LD     (ROTPTR),IX     ;SAVE TYPE
005A'  DD 23        ROTO40:  INC    IX              ;BUMP INDEX TO NEXT
005C'  DD 23                 INC    IX
005E'  10 E7                 DJNZ   ROTO30          ;LOOP IF NOT DONE
0060'  FD E1                 POP    IY             ;RESTORE REGISTERS
0062'  DD E1                 POP    IX
0064'  E1                    POP    HL
0065'  C1                    POP    BC
0066'  F1                    POP    AF
0067'  C9                    RET                   ;RETURN
                             END
                             TITLE  NUMBSR
                             ENTRY  NUMBER
                             EXT    NOX,NOO,NOSP
                    ;*****************************************************
                    ;*                  NUMBER SUBROUTINE                *
                    ;*   COUNTS NUMBER OF XS, OS, AND SPACES IN GIVEN    *
                    ;*   ARRAY.                                          *
                    ;*   ENTRY:  (HL)=ARRAY ADDRESS                      *
                    ;*   EXIT:   NUMBER OF EACH IN NOX, NOO, AND NOSP    *
                    ;*   ALL REGISTERS SAVED                             *
                    ;*****************************************************
                    ;
0000'  F5           NUMBER:  PUSH   AF             ;SAVE REGISTERS
0001'  C5                    PUSH   BC
0002'  D5                    PUSH   DE
0003'  E5                    PUSH   HL
0004'  06 09                 LD     B,9            ;NINE POSITIONS
0006'  11 0000               LD     DE,0           ;D=# OF ONES,E=# OF FOURS
```

```
0009'  0E 00            LD      C,0          ;C=# OF SPACES
000B'  7E       NUM010: LD      A,(HL)         ;GET ELEMENT
000C'  FE 01            CP      1              ;IS IT AN X?
000E'  20 01            JR      NZ,NUM020      ;GO IF NO
0010'  14               INC     D              ;BUMP X COUNT
0011'  FE 04    NUM020: CP      4              ;IS IT AN O?
0013'  20 01            JR      NZ,NUM030      ;GO IF NO
0015'  1C               INC     E              ;BUMP O COUNT
0016'  B7       NUM030: OR      A              ;TEST FOR SPACE
0017'  20 01            JR      NZ,NUM040      ;GO IF NOT
0019'  0C               INC     C              ;BUMP # OF SPACES
001A'  23       NUM040: INC     HL             ;BUMP PNTR
001B'  10 EE            DJNZ    NUM010         ;GO IF NOT 9
001D'  7A               LD      A,D          ;GET X COUNT
001E'  32 0000*         LD      (NOX),A      ;STORE
0021'  7B               LD      A,E          ;GET O COUNT
0022'  32 0000*         LD      (NOO),A      ;STORE
0025'  79               LD      A,C          ;GET SPACE COUNT
0026'  32 0000*         LD      (NOSP),A     ;STORE
0029'  E1               POP     HL           ;RESTORE REGISTERS
002A'  D1               POP     DE
002B'  C1               POP     BC
002C'  F1               POP     AF
002D'  C9               RET                  ;RETURN
                        END
                        TITLE   ANALSR
                        ENTRY   ANALAR
                        EXT     ANALTB
                ;*****************************************************
                ;*                  ANALYZE ARRAY                    *
                ;*  FINDS NUMBER OF XS AND YS IN EACH ROW, COLUMN,   *
                ;*  AND DIAGONAL OF GIVEN ARRAY.                     *
                ;*  ENTRY:  (IX)=ADDRESS OF 9-ELEMENT ARRAY          *
                ;*  EXIT:   PARAMETERS IN ANALTB                     *
                ;*  ALL REGISTERS SAVED                              *
                ;*****************************************************
                ;
0000'  F5       ANALAR: PUSH    AF           ;SAVE REGISTERS
0001'  FD E5            PUSH    IY
0003'  FD 21 0000*      LD      IY,ANALTB    ;ADDRESS OF TABLE
0007'  DD 7E 00         LD      A,(IX+0)     ;ROW 0
000A'  DD 86 01         ADD     A,(IX+1)
000D'  DD 86 02         ADD     A,(IX+2)
0010'  FD 77 00         LD      (IY),A       ;STORE RESULT
0013'  DD 7E 03         LD      A,(IX+3)     ;ROW 1
0016'  DD 86 04         ADD     A,(IX+4)
```

```
0019'  DD 86 05         ADD     A,(IX+5)
001C'  FD 77 01         LD      (IY+1),A        ;STORE RESULT
001F'  DD 7E 06         LD      A,(IX+6)        ;ROW 2
0022'  DD 86 07         ADD     A,(IX+7)
0025'  DD 86 08         ADD     A,(IX+8)
0028'  FD 77 02         LD      (IY+2),A        ;STORE RESULT
002B'  DD 7E 00         LD      A,(IX+0)        ;COLUMN 0
002E'  DD 86 03         ADD     A,(IX+3)
0031'  DD 86 06         ADD     A,(IX+6)
0034'  FD 77 03         LD      (IY+3),A        ;STORE RESULT
0037'  DD 7E 01         LD      A,(IX+1)        ;COLUMN 2
003A'  DD 86 04         ADD     A,(IX+4)
003D'  DD 86 07         ADD     A,(IX+7)
0040'  FD 77 04         LD      (IY+4),A        ;STORE RESULT
0043'  DD 7E 02         LD      A,(IX+2)        ;COLUMN 2
0046'  DD 86 05         ADD     A,(IX+5)
0049'  DD 86 08         ADD     A,(IX+8)
004C'  FD 77 05         LD      (IY+5),A        ;STORE RESULT
004F'  DD 7E 00         LD      A,(IX+0)        ;DIAGONAL 0
0052'  DD 86 04         ADD     A,(IX+4)
0055'  DD 86 08         ADD     A,(IX+8)
0058'  FD 77 06         LD      (IY+6),A        ;STORE RESULT
005B'  DD 7E 02         LD      A,(IX+2)        ;DIAGONAL 1
005E'  DD 86 04         ADD     A,(IX+4)
0061'  DD 86 06         ADD     A,(IX+6)
0064'  FD 77 07         LD      (IY+7),A        ;STORE RESULT
0067'  FD E1            POP     IY              ;RESTORE REGISTERS
0069'  F1               POP     AF
006A'  C9               RET                     ;RETURN
                        END
                        TITLE   ARRXSR
                        ENTRY   ARRXLA
                ;*****************************************************
                ;*                  ARRAY TRANSLATOR                 *
                ;*   CONVERTS A 9-ELEMENT GIVEN ARRAY TO A SECOND    *
                ;*   TRANSLATED ARRAY                                *
                ;*   ENTRY:   (IX)=POINTER TO SOURCE ARRAY           *
                ;*            (IY)=POINTER TO DESTINATION ARRAY      *
                ;*            (HL)=POINTER TO LIST OF 9 INDICES      *
                ;*   EXIT:    ELEMENTS OF SOURCE ARRAY CONVERTED     *
                ;*            TO ELEMENTS OF DESTINATION ARRAY AC-   *
                ;*            CORDING TO INDEX LIST                  *
                ;*   ALL REGISTERS SAVED                             *
                ;*****************************************************
                ;
0000'  F5       ARRXLA: PUSH    AF              ;SAVE REGISTERS
```

```
0001'  C5                   PUSH    BC
0002'  D5                   PUSH    DE
0003'  E5                   PUSH    HL
0004'  DD E5                PUSH    IX
0006'  FD E5                PUSH    IY
0008'  06 09                LD      B,9             ;9 INDICES
000A'  FD E5                PUSH    IY              ;SAVE DESTINATION PNTR
000C'  5E          ARR020:  LD      E,(HL)            ;GET NEXT INDEX
000D'  16 00                LD      D,0               ;INDEX TO DE
000F'  FD E1                POP     IY                ;DESTINATION PNTR
0011'  FD E5                PUSH    IY                ;SAVE
0013'  FD 19                ADD     IY,DE             ;DEST ARRAY PNTR+INDEX
0015'  DD 7E 00             LD      A,(IX)            ;GET ARRAY 1 VALUE
0018'  FD 77 00             LD      (IY),A            ;STORE IN 2ND ARRAY
001B'  23                   INC     HL                ;BUMP INDEX INDEX
001C'  DD 23                INC     IX                ;BUMP SOURCE ARRAY PNTR
001E'  10 EC                DJNZ    ARR020            ;CONTINUE IF NOT 9
0020'  FD E1       ARR090:  POP     IY              ;RESET STACK FOR ARRAY 2
0022'  FD E1                POP     IY              ;RESTORE REGISTERS
0024'  DD E1                POP     IX
0026'  E1                   POP     HL
0027'  D1                   POP     DE
0028'  C1                   POP     BC
0029'  F1                   POP     AF
002A'  C9                   RET                     ;RETURN
                            END
                            TITLE   DRAWSR
                            ENTRY   DRAWL
                   ;*****************************************************
                   ;*               DRAW LINE SUBROUTINE                *
                   ;*  DRAWS HORIZONTAL OR VERTICAL LINE                *
                   ;*  ENTRY:  (A)=GRAPHICS CHARACTER TO BE USED        *
                   ;*          (C)=0 IF HORIZONTAL, NON-ZERO IF VERTICAL*
                   ;*          (HL)=SCREEN START POSITION               *
                   ;*          (B)=# OF CHARACTER POSITIONS TO BE USED  *
                   ;*  EXIT: ALL REGISTERS SAVED                        *
                   ;*****************************************************
                   ;
0000'  C5          DRAWL:   PUSH    BC              ;SAVE REGISTERS
0001'  D5                   PUSH    DE
0002'  E5                   PUSH    HL
0003'  F5                   PUSH    AF              ;SAVE CHARACTER
0004'  79                   LD      A,C             ;GET HORIZ/VERT FLAG
0005'  B7                   OR      A               ;TEST
0006'  20 07                JR      NZ,DRA060       ;GO IF VERTICAL
                   ; HORIZONTAL LINE HERE
```

```
0008'  F1                      POP     AF              ;RESTORE CHAR
0009'  77              DRA050: LD      (HL),A            ;STORE GRAPHICS
000A'  23                      INC     HL                ;BUMP POINTER
000B'  10 FC                   DJNZ    DRA050            ;GO IF MORE
000D'  18 08                   JR      DRA090          ;DONE
                       ;VERTICAL LINE HERE
000F'  F1              DRA060: POP     AF              ;RESTORE CHARACTER
0010'  11 0040                 LD      DE,64           ;INCREMENT
0013'  77              DRA065: LD      (HL),A            ;STORE GRAPHICS
0014'  19                      ADD     HL,DE             ;POINT TO NEXT
0015'  10 FC                   DJNZ    DRA065            ;GO IF MORE
0017'  E1              DRA090: POP     HL              ;RESTORE REGISTERS
0018'  D1                      POP     DE
0019'  C1                      POP     BC
001A'  C9                      RET                     ;RETURN
                               END
                               TITLE   SCRNSR
                               ENTRY   SCRNDS
                               EXT     MSGOUT,TEMP1,ARRAY1,SCARRY
                       ;**************************************************
                       ;*              DISPLAY SCREEN ARRAY              *
                       ;*    CONVERTS ARRAY1 TO SCREEN DISPLAY           *
                       ;*    ENTRY: NO PARAMETERS                        *
                       ;*    EXIT:  NO PARAMETERS                        *
                       ;*    ALL REGISTERS SAVED                         *
                       ;**************************************************
                       ;
0000'  F5              SCRNDS: PUSH    AF              ;SAVE REGISTERS
0001'  C5                      PUSH    BC
0002'  D5                      PUSH    DE
0003'  E5                      PUSH    HL
0004'  FD E5                   PUSH    IY
0006'  21 0000*                LD      HL,ARRAY1       ;ADDRESS OF ARRAY
0009'  06 09                   LD      B,9             ;NUMBER OF ELEMENTS
000B'  FD 21 0000*             LD      IY,SCARRY       ;SCREEN ARRAY ADDRESS
000F'  7E              SCR010: LD      A,(HL)            ;GET ELEMENT
0010'  B7                      OR      A                 ;TEST FOR BLANK
0011'  28 15                   JR      Z,SCR020          ;GO IF BLANK
0013'  FE 01                   CP      1                 ;TEST FOR X
0015'  20 04                   JR      NZ,SCR015         ;GO IF O
0017'  3E 58                   LD      A,'X'             ;X
0019'  18 02                   JR      SCR018            ; GO TO STORE
001B'  3E 4F           SCR015: LD      A,'O'           ;MUST BE O
001D'  32 0000*        SCR018: LD      (TEMP1),A       ;STORE CHARACTER
0020'  E5                      PUSH    HL              ;SAVE PNTR
0021'  21 0000*                LD      HL,TEMP1          ;MESSAGE BUFFER ADDR
```

```
0024'  CD 0000*              CALL    MSGOUT          ;DISPLAY
0027'  E1                    POP     HL              ;RESTORE PNTR
0028'  23           SCR020:  INC     HL              ;BUMP ARRAY PNTR
0029'  11 000B               LD      DE,11           ;INCREMENT
002C'  FD 19                 ADD     IY,DE           ;NEXT SCREEN ADDRESS
002E'  78                    LD      A,B             ;GET COUNT
002F'  FE 07                 CP      7               ;TEST FOR NEW ROW
0031'  28 04                 JR      Z,SCR030        ;GO IF NEW ROW
0033'  FE 04                 CP      4               ;TEST FOR NEW ROW
0035'  20 05                 JR      NZ,SCR040       ;GO IF NOT NEW ROW
0037'  11 009F      SCR030:  LD      DE,159          ;ADJUSTMENT
003A'  FD 19                 ADD     IY,DE           ;ADD ADJUSTMENT
003C'  10 D1        SCR040:  DJNZ    SCR010          ;GO IF NOT 3 ROWS
003E'  FD E1                 POP     IY             ;RESTORE REGISTERS
0040'  E1                    POP     HL
0041'  D1                    POP     DE
                             TITLE   MSGOSR
                             ENTRY   MSGOUT
                             EXT     LARGEC
                    ;**********************************************
                    ;*             LARGE MESSAGE OUTPUT ROUTINE         *
                    ;*   OUTPUTS MESSAGE IN LARGE FORMAT                *
                    ;*   ENTRY: (HL)=POINTER TO MESSAGE, MSG TERMINATED *
                    ;*                BY 0                              *
                    ;*          (IY)=POINTER TO SCREEN AREA             *
                    ;*   EXIT:  ALL REGISTERS SAVED                     *
                    ;**********************************************
                    ;
0000'  F5           MSGOUT:  PUSH    AF             ;SAVE REGISTERS
0001'  C5                    PUSH    BC
0002'  E5                    PUSH    HL
0003'  FD E5                 PUSH    IY
0005'  7E           MSG010:  LD      A,(HL)          ;GET NEXT CHARACTER
0006'  B7                    OR      A               ;TEST
0007'  28 0B                 JR      Z,MSG090        ;GO IF DONE
0009'  CD 0000*              CALL    LARGEC          ;DISPLAY
000C'  01 FFC1               LD      BC,-63          ;DISPLACEMENT FOR 5 CP
000F'  FD 09                 ADD     IY,BC           ;POINT TO NEXT CHAR POS
0011'  23                    INC     HL              ;POINT TO NEXT CHAR
0012'  18 F1                 JR      MSG010          ;CONTINUE
0014'  FD E1        MSG090:  POP     IY             ;RESTORE REGISTERS
0016'  E1                    POP     HL
0017'  C1                    POP     BC
0018'  F1                    POP     AF
0019'  C9                    RET                    ;RETURN
                             END
```

```
                             TITLE   LARGSR
                             ENTRY   LARGEC
                    ;*****************************************************
                    ;*            LARGE CHARACTER DISPLAY                 *
                    ;*   STORES LARGE CHARACTER AT GIVEN SCREEN POSITION  *
                    ;*   8 BY 6 MATRIX USED                               *
                    ;*   ENTRY:  (A)=CHARACTER TO BE STORED IN ASCII      *
                    ;*           (IY)=SCREEN POSITION                     *
                    ;*   EXIT:   ALL REGISTERS SAVED                      *
                    ;*****************************************************
                    ;
0000'  F5           LARGEC: PUSH  AF              ;SAVE REGISTERS
0001'  C5                   PUSH  BC
0002'  E5                   PUSH  HL
0003'  DD E5                PUSH  IX
0005'  DD 21 0049'          LD    IX,CTAB         ;CHARACTER TABLE ADDRESS
0009'  DD BE 00     LAR020: CP    (IX)              ;TEST FOR CHARACTER
000C'  DD 23                INC   IX                ;BUMP TO NEXT
000E'  20 F9                JR    NZ,LAR020         ;GO IF NOT FOUND
0010'  DD E5                PUSH  IX              ;TRANSFER IX TO HL
0012'  E1                   POP   HL
0013'  01 004A'             LD    BC,CTAB+1       ;CHAR TABLE START+1
0016'  B7                   OR    A               ;CLEAR CARRY
0017'  ED 42                SBC   HL,BC           ;FIND DISPLACEMENT
0019'  E5                   PUSH  HL              ;TRANSFER HL TO IX
001A'  DD E1                POP   IX
001C'  DD 29                ADD   IX,IX           ;INDEX*2
001E'  DD 29                ADD   IX,IX           ;INDEX*4
0020'  DD 29                ADD   IX,IX           ;INDEX*8
0022'  01 0067'             LD    BC,DOTTAB       ;8 BY 6 MATRIX
0025'  DD 09                ADD   IX,BC           ;POINT TO PATTERN
0027'  CD 0038'             CALL  MATSR           ;STORE TOP ROW
002A'  01 003C              LD    BC,60           ;LINE ADJUST
002D'  FD 09                ADD   IY,BC           ;ADD TO SCREEN PNTR
002F'  CD 0038'             CALL  MATSR           ;STORE BOTTOM ROW
0032'  DD E1                POP   IX              ;RESTORE REGISTERS
0034'  E1                   POP   HL
0035'  C1                   POP   BC
0036'  F1                   POP   AF
0037'  C9                   RET                   ;RETURN
0038'  06 04        MATSR:  LD    B,4             ;FOR ROW
003A'  DD 7E 00     MAT010: LD    A,(IX)            ;GET ELEMENTS
003D'  CB FF                SET   7,A               ;SET GRAPHICS
003F'  FD 77 00             LD    (IY),A            ;STORE
0042'  DD 23                INC   IX                ;BUMP ELEMENT PNTR
0044'  FD 23                INC   IY                ;BUMP SCREEN PNTR
```

```
0046'  10 F2                  DJNZ    MAT010          ;GO IF NOT ENTIRE ROW
0048'  C9                     RET                     ;RETURN
                       ; TABLE OF ALLOWABLE CHARACTERS
0049'  41 42 43 44     CTAB:  DB      'ABCDEFGHIJKLMNOPQRSTUVWXYZ -?!'
004D'  45 46 47 48
0051'  49 4A 4B 4C
0055'  4D 4E 4F 50
0059'  51 52 53 54
005D'  55 56 57 58
0061'  59 5A 20 2D
0065'  3F 21
                       ; TABLE OF MATRICES FOR DISPLAY
0067'  17 03 03 2B     DOTTAB: DB     23,3,3,43,23,3,3,43     ;A
006B'  17 03 03 2B
006F'  17 03 03 2B            DB      23,3,3,43,55,51,51,59   ;B
0073'  37 33 33 3B
0077'  17 03 03 0B            DB      23,3,3,11,53,48,48,56   ;C
007B'  35 30 30 38
007F'  17 03 03 29            DB      23,3,3,41,53,48,48,26   ;D
0083'  35 30 30 1A
0087'  17 03 03 03            DB      23,3,3,3,55,51,51,48    ;E
008B'  37 33 33 30
008F'  17 03 03 03            DB      23,3,3,3,23,3,3,1       ;F
0093'  17 03 03 01
0097'  17 03 03 0B            DB      23,3,3,11,53,48,48,59   ;G
009B'  35 30 30 3B
009F'  15 00 00 2A            DB      21,0,0,42,23,3,3,43     ;H
00A3'  17 03 03 2B
00A7'  00 2B 17 00            DB      0,43,23,0,0,58,53,0     ;I
00AB'  00 3A 35 00
00AF'  00 00 00 2B            DB      0,0,0,43,52,48,48,56    ;J
00B3'  34 30 30 38
00B7'  15 00 20 06            DB      21,0,32,6,23,3,3,36     ;K
00BB'  17 03 03 24
00BF'  15 00 00 00            DB      21,0,0,0,53,48,48,48    ;L
00C3'  35 30 30 30
00C7'  1F 10 20 2F            DB      31,16,32,47,21,2,1,42   ;M
00CB'  15 02 01 2A
00CF'  17 24 00 2A            DB      23,36,0,42,21,0,9,58    ;N
00D3'  15 00 09 3A
00D7'  16 03 03 29            DB      22,3,3,41,37,48,48,26   ;O
00DB'  25 30 30 1A
00DF'  17 03 03 2B            DB      23,3,3,43,23,3,3,3      ;P
00E3'  17 03 03 03
00E7'  16 03 03 29            DB      22,3,3,41,37,48,56,26   ;Q
00EB'  25 30 38 1A
```

```
00EF'  17 03 03 2B          DB      23,3,3,43,23,3,3,37      ;R
00F3'  17 03 03 25
00F7'  17 03 03 03          DB      23,3,3,3,51,51,51,59     ;S
00FB'  33 33 33 3B
00FF'  03 2B 17 03          DB      3,43,23,3,0,42,21,0      ;T
0103'  00 2A 15 00
0107'  15 00 00 2A          DB      21,0,0,42,53,48,48,58    ;U
010B'  35 30 30 3A
010F'  15 00 00 2A          DB      21,0,0,42,2,36,24,1      ;V
0113'  02 24 18 01
0117'  15 28 14 2A          DB      21,40,20,42,37,58,53,26 ;W
011B'  25 3A 35 1A
011F'  09 30 30 06          DB      9,48,48,6,24,3,3,36      ;X
0123'  18 03 03 24
0127'  15 00 00 2A          DB      21,0,0,42,3,43,23,3      ;Y
012B'  03 2B 17 03
012F'  03 03 33 0F          DB      3,3,51,15,60,51,48,48    ;Z
0133'  3C 33 30 30
0137'  00 00 00 00          DB      0,0,0,0,0,0,0,0          ;SPACE
013B'  00 00 00 00
013F'  00 00 00 00          DB      0,0,0,0,3,3,3,3          ;-
0143'  03 03 03 03
0147'  07 23 33 3B          DB      7,35,51,59,0,34,17,0     ;?
014B'  00 22 11 00
014F'  00 2A 15 00          DB      0,42,21,0,0,34,17,0      ;!
0153'  00 22 11 00
                            END
                            TITLE   DSPMSR
                            ENTRY   DSPMES
                    ;******************************************************
                    ;*     DISPLAY MESSAGE AT LOCATION N SUBROUTINE       *
                    ;*   DISPLAYS MESSAGE AT GIVEN SCREEN POSITION. TER-  *
                    ;*   MINATES ON NULL (ZERO).                          *
                    ;*   ENTRY:  (HL)=MESSAGE LOCATION                    *
                    ;*           (BC)=SCREEN POSITION                     *
                    ;*   ALL REGISTERS SAVED.                             *
                    ;******************************************************
                    ;
0000'  F5           DSPMES: PUSH    AF              ;SAVE REGISTERS
0001'  C5                   PUSH    BC
0002'  E5                   PUSH    HL
0003'  7E           DSP005: LD      A,(HL)          ;GET MESSAGE CHARACTER
0004'  B7                   OR      A               ;TEST FOR 0
0005'  28 05                JR      Z,DSP010        ;RETURN IF DONE
0007'  02                   LD      (BC),A          ;STORE CHARACTER
0008'  03                   INC     BC              ;BUMP SCREEN POINTER
```

```
0009'   23                    INC     HL              ;BUMP MESSAGE POINTER
000A'   18 F7                 JR      DSP005          ;CONTINUE
000C'   E1           DSP010:  POP     HL              ;RESTORE REGISTERS
000D'   C1                    POP     BC
000E'   F1                    POP     AF
000F'   C9                    RET                     ;RETURN
                              END
                              TITLE   INPUSR
                              ENTRY   INPUT
                              EXT     DELAY
                     ;*************************************************
                     ;*              INPUT SUBROUTINE                 *
                     ;*   SCANS KEYBOARD FOR INPUT OF 0,1,2,3,4,5,6,7, OR  *
                     ;*   8 KEYS. IGNORES ALL OTHERS. WAITS FOR KEY PRESS. *
                     ;*   ENTRY: NO PARAMETERS                        *
                     ;*   EXIT: (A)=0,1,2,3,4,5,6,7, OR 8 IN BINARY   *
                     ;*   ALL REGISTERS EXCEPT A SAVED                *
                     ;*************************************************
                     ;
0000'   C5           INPUT:   PUSH    BC              ;SAVE REGISTERS
0001'   E5                    PUSH    HL
0002'   3A 3810      INP002:  LD      A,(3810H)       ;KEYBOARD ROW 4
0005'   B7                    OR      A               ;TEST FOR NON-ZERO
0006'   28 09                 JR      Z,INP010        ;GO IF NCN
0008'   0E FF                 LD      C,0FFH          ;COLUMN COUNT
000A'   0C           INP005:  INC     C               ;BUMP COUNT
000B'   0F                    RRCA                    ;SHIFT OUT
000C'   30 FC                 JR      NC,INP005       ;GO IF NOT CARRY
000E'   79                    LD      A,C             ;COUNT TO A
000F'   18 09                 JR      INP020          ;CONTINUE
0011'   3A 3820      INP010:  LD      A,(3820H)       ;KEYBOARD ROW 5
0014'   E6 01                 AND     1               ;GET "8" KEY ONLY
0016'   28 EA                 JR      Z,INP002        ;GO IF NONE
0018'   3E 08                 LD      A,8             ;FOR 8 KEY
001A'   21 0064      INP020:  LD      HL,100          ;100 MILLISEC DELAY
001D'   CD 0000*              CALL    DELAY           ;DELAY
0020'   E1                    POP     HL              ;RESTORE REGISTERS
0021'   C1                    POP     BC
0022'   C9                    RET
                              END
                              TITLE   RANDSR
                              ENTRY   RAND
                              EXT     SEED
                     ;*************************************************
                     ;*            RANDOM NUMBER ROUTINE              *
                     ;*   GENERATES A PSEUDO-RANDOM NUMBER FROM 0 TO 65535 *
```

```
                          ;*   ENTRY: NO PARAMETERS                                   *
                          ;*   EXIT:  (BC)=RANDOM # 0-65535                           *
                          ;*   ALL REGISTERS SAVED EXCEPT BC                          *
                          ;***********************************************************
                          ;
0000'   F5                RAND:   PUSH    AF                ;SAVE REGISTERS
0001'   D5                        PUSH    DE
0002'   E5                        PUSH    HL
0003'   ED 5B 0000*               LD      DE,(SEED)         ;GET SEED
0007'   2A 0002*                  LD      HL,(SEED+2)
000A'   06 07                     LD      B,7               ;COUNT FOR MULTIPLY BY 128
000C'   CD 0025'          RDM010: CALL    SHIFT               ;SHIFT ONE BIT LEFT
000F'   10 FB                     DJNZ    RDM010              ;SEED*128
0011'   06 03                     LD      B,3               ;FOR SUBTRACT
0013'   CD 002B'          RDM020: CALL    SUB                 ;SUBTRACT ONE
0016'   10 FB                     DJNZ    RDM020              ;SEED*128-3*SEED=SEED*125
0018'   ED 53 0000*               LD      (SEED),DE         ;STORE NEW SEED
                          ;*****WARNING*****  POSSIBLE LOADER ERROR IN SOME VERSIONS
                          ;OF ASSEMBLER. ASSEMBLER LOAD ADDRESS SHOULD BE SEED+2 IS
                          ;SEED!
001C'   22 0002*                  LD      (SEED+2),HL
001F'   43                        LD      B,E               ;NOW IN B
0020'   4C                        LD      C,H               ;NOW IN BC
0021'   E1                        POP     HL                ;RESTORE REGISTERS
0022'   D1                        POP     DE
0023'   F1                        POP     AF
0024'   C9                        RET                       ;RETURN
                          ; SHIFT SUBROUTINE
0025'   29                SHIFT:  ADD     HL,HL             ;SHIFT HL
0026'   EB                        EX      DE,HL             ;GET MS BYTE
0027'   ED 6A                     ADC     HL,HL             ;SHIFT MS 2 BYTES
0029'   EB                        EX      DE,HL             ;NOW ORIGINAL*2
002A'   C9                        RET                       ;RETURN
                          ; SUBTRACT SEED SUBROUTINE
002B'   C5                SUB:    PUSH    BC                ;SAVE REGISTERS
002C'   ED 4B 0002*               LD      BC,(SEED+2)       ;GET LS BYTE
0030'   B7                        OR      A                 ;RESET CARRY
0031'   ED 42                     SBC     HL,BC             ;SUBTRACT LS 2 BYTES
0033'   EB                        EX      DE,HL             ;GET MS 2 BYTES
0034'   ED 4B 0000*               LD      BC,(SEED)         ;GET MS 2 BYTES
0038'   ED 42                     SBC     HL,BC             ;SUBTRACT MS 2 BYTES AND CY
003A'   EB                        EX      DE,HL             ;NOW ORIGINAL-SEED
003B'   C1                        POP     BC                ;RESTORE REGISTERS
003C'   C9                        RET                       ;RETURN
                                  END
                                  TITLE   BINBSR
```

```
                     ENTRY   BINBAS
                     EXT     DIVIDE
               ;************************************************
               ;*           BINARY TO BASE 3 CONVERSION          *
               ;*   CONVERTS A TWO-BYTE BINARY NUMBER TO BASE 3  *
               ;*   EQUIVALENT.                                  *
               ;*   ENTRY:  (HL)=BINARY NUMBER                   *
               ;*           (IX)=POINTER TO ARRAY ENTRY 0        *
               ;*   EXIT:   NUMBER IN ARRAY(0) TO ARRAY(8)       *
               ;*   ALL REGISTERS SAVED                          *
               ;************************************************
               ;
0000'  F5      BINBAS: PUSH  AF              ;SAVE REGISTERS
0001'  C5              PUSH  BC
0002'  E5              PUSH  HL
0003'  DD E5           PUSH  IX
0005'  01 0008         LD    BC,8            ;DISPLACEMENT
0008'  DD 09           ADD   IX,BC           ;POINT TO LAST ARRAY SLOT
000A'  1E 03           LD    E,3             ;DIVISOR
000C'  06 09           LD    B,9             ;NUMBER OF BASE 3 DIGITS
000E'  C5      BIN020: PUSH  BC                ;SAVE
000F'  CD 0000*        CALL  DIVIDE            ;DIVIDE BY 3
0012'  79              LD    A,C               ;REMAINDER
0013'  FE 02           CP    2                 ;IS IT A 2?
0015'  20 02           JR    NZ,BIN030         ;GO IF NO
0017'  0E 04           LD    C,4               ;YES-CHANGE TO 4
0019'  DD 71 00 BIN030: LD   (IX),C            ;SAVE REMAINDER
001C'  DD 2B           DEC   IX                ;POINT TO NEXT ARRAY SLOT
001E'  C1              POP   BC
001F'  10 ED           DJNZ  BIN020            ;GO IF NOT 9 TIMES
0021'  DD E1           POP   IX              ;RESTORE REGISTERS
0023'  E1              POP   HL
0024'  C1              POP   BC
0025'  F1              POP   AF
0026'  C9              RET                   ;RETURN
                       END
                       TITLE   BASBSR
                       ENTRY   BASBIN
               ;************************************************
               ;*          BASE 3 TO BINARY CONVERSION           *
               ;*  CONVERTS A BASE 3 NUMBER TO BINARY EQUIVALENT *
               ;*  ENTRY:  (IY)=POINTER TO ARRAY ENTRY 0         *
               ;*  EXIT:   (HL)=BINARY NUMBER, UNSIGNED          *
               ;*  ALL REGISTERS SAVED                           *
               ;************************************************
               ;
```

```
0000'  F5                BASBIN: PUSH    AF              ;SAVE REGISTERS
0001'  C5                        PUSH    BC
0002'  D5                        PUSH    DE
0003'  FD E5                     PUSH    IY
0005'  21 0000                   LD      HL,0            ;CLEAR RESULT
0008'  06 09                     LD      B,9             ;LOOP COUNT
000A'  E5                BAS020: PUSH    HL                ;TRANSFER RESULT TO DE
000B'  D1                        POP     DE
000C'  19                        ADD     HL,DE             ;RESULT*2
000D'  19                        ADD     HL,DE             ;RESULT*3
000E'  FD 7E 00                  LD      A,(IY)            ;GET NEXT BASE 3 DIGIT
0011'  FE 04                     CP      4                 ;IS IT A 4?
0013'  20 02                     JR      NZ,BAS030         ;GO IF NO
0015'  3E 02                     LD      A,2               ;YES-BACK TO BASE 3
0017'  85                BAS030: ADD     A,L               ;MERGE
0018'  6F                        LD      L,A               ;NOW IN HL
0019'  30 01                     JR      NC,BAS040         ;GO IF NO CARRY
001B'  24                        INC     H                 ;BUMP MS BYTE
001C'  FD 23             BAS040: INC     IY                ;POINT TO NEXT SLOT
001E'  10 EA                     DJNZ    BAS020            ;GO IF NOT 9
0020'  FD E1                     POP     IY              ;RESTORE REGISTERS
0022'  D1                        POP     DE
0023'  C1                        POP     BC
0024'  F1                        POP     AF
0025'  C9                        RET                     ;RETURN
                                 END
                                 TITLE   DIVISR
                                 ENTRY   DIVIDE
                         ;*****************************************************
                         ;*                  DIVIDE SUBROUTINE                *
                         ;*   PERFORMS AN UNSIGNED DIVIDE OF 16 BITS BY 8-BIT *
                         ;*   DIVISOR.                                        *
                         ;*   ENTRY:  (HL)=DIVIDEND                           *
                         ;*           (E)=DIVISOR                             *
                         ;*   EXIT:   (HL)=QUOTIENT                           *
                         ;*           (BC)=REMAINDER                          *
                         ;*   ALL REGISTERS SAVED EXCEPT HL,BC                *
                         ;*****************************************************
                         ;
0000'  D5                DIVIDE: PUSH    DE              ;SAVE REGISTERS
0001'  DD E5                     PUSH    IX
0003'  E5                        PUSH    HL              ;DIVIDEND TO IX
0004'  DD E1                     POP     IX
0006'  21 0000                   LD      HL,0            ;ZERO HL
0009'  16 00                     LD      D,0             ;DIVISOR IN DE
000B'  06 10                     LD      B,16            ;ITERATION COUNT
```

```
000D'   29                  DIV010: ADD     HL,HL           ;SHIFT 16 MS BITS
000E'   DD 29                       ADD     IX,IX           ;SHIFT 16 LS BITS
0010'   30 01                       JR      NC,DIV020       ;GO IF NO CARRY
0012'   23                          INC     HL              ;CARRY TO 16 MS BITS
0013'   DD 23               DIV020: INC     IX              ;SET Q BIT
0015'   B7                          OR      A               ;CLEAR CARRY
0016'   ED 52                       SBC     HL,DE           ;TRY SUBTRACT
0018'   30 03                       JR      NC,DIV030       ;GO IF DIVIDE WENT
001A'   19                          ADD     HL,DE           ;RESTORE
001B'   DD 2B                       DEC     IX              ;RESET Q BIT
001D'   10 EE               DIV030: DJNZ    DIV010          ;GO IF NOT 16 ITERATIONS
001F'   E5                          PUSH    HL             ;REMAINDER
0020'   C1                          POP     BC
0021'   DD E5                       PUSH    IX             ;QUOTIENT
0023'   E1                          POP     HL
0024'   DD E1                       POP     IX             ;RESTORE REGISTERS
0026'   D1                          POP     DE
0027'   C9                          RET                    ;RETURN
                                    END
                                    TITLE   DELASR
                                    ENTRY   DELAY
                            ;**********************************************
                            ;*              DELAY SUBROUTINE              *
                            ;*  DELAYS 1 TO 65536 MILLISECONDS.           *
                            ;*  ENTRY: (HL)=DELAY COUNT IN MILLISECONDS   *
                            ;*         0=65536                            *
                            ;*  ALL REGISTERS SAVED                       *
                            ;**********************************************
                            ;
0000'   C5                  DELAY:  PUSH    BC             ;SAVE REGISTERS
0001'   D5                          PUSH    DE
0002'   E5                          PUSH    HL
0003'   11 FFFF                     LD      DE,-1          ;-1 FOR DECREMENT
0006'   2B                          DEC     HL             ;ADJUST FOR "JP NC"
0007'   06 83               DEL010: LD      B,131            ;INNER LOOP COUNT
0009'   10 FE               DEL020: DJNZ    DEL020             ;LOOP FOR DELAY
000B'   19                          ADD     HL,DE           ;DECREMENT OUTER LOOP CNT
000C'   DA 0007'                    JP      C,DEL010        ;CONTINUE
000F'   E1                          POP     HL             ;RESTORE REGISTERS
0010'   D1                          POP     DE
0011'   C1                          POP     BC
0012'   C9                          RET                    ;RETURN
                                    END
                                    TITLE   FILLSR
                                    ENTRY   FILLCH
                            ;**********************************************
```

```
                    ;*                FILL CHARACTER SUBROUTINE                 *
                    ;*   FILLS DESIGNATED AREA WITH GIVEN CHARACTER             *
                    ;*   ENTRY:  (A)=CHARACTER                                  *
                    ;*           (DE)=AREA                                      *
                    ;*           (BC)=NUMBER OF BYTES, 1-65535; 0=65536         *
                    ;*   ALL REGISTERS SAVED EXCEPT BC,DE                       *
                    ;***********************************************************
                    ;
0000'   12          FILLCH: LD      (DE),A          ;FILL CHARACTER
0001'   13                  INC     DE              ;BUMP POINTER
0002'   0B                  DEC     BC              ;DECREMENT COUNT
0003'   F5                  PUSH    AF              ;SAVE FILL CHARACTER
0004'   78                  LD      A,B             ;TEST FOR ZERO
0005'   B1                  OR      C
0006'   28 03               JR      Z,FIL010        ;GO IF DONE
0008'   F1                  POP     AF              ;RESTORE FILL CHAR
0009'   18 F5               JR      FILLCH          ;CONTINUE
000B'   F1          FIL010: POP     AF              ;RESTORE A
000C'   C9                  RET                    ;RETURN
                            END
                            TITLE   SYSTEM
                            ENTRY   MSG1,HISTRY,MSG2,MSG3,MSG4,MSG5
                            ENTRY   MSG6,MSG7,MSG8,MSG9,MSG10,MSG11,MSG12
                            ENTRY   FRSTF,NXTHIS,ARRAY1,ARRAY2,SEED,RINDW
                            ENTRY   ANALTB,NORO,NOR1,NOR2,NOC0,NOC1,NOC2
                            ENTRY   NODO,NOD1,ROTTAB,NOPTE,ROTPTR,MOVETB
                            ENTRY   MOVENO,NOX,NOO,NOSP,TEMP1,GRIDTB,RINDT,RINDT9
                            ENTRY   RINDTR,PTABLE,STACK
                    ;***********************************************************
                    ;*      SYSTEM MESSAGES, VARIABLES, TABLES                  *
                    ;***********************************************************
                    ;
                    ; SYSTEM MESSAGES
                    ;
0000'   20 20 20 20 MSG1:   DB      '                    HISTORY: LAST 128 GAMES'
0004'   20 20 20 20
0008'   20 20 20 20
000C'   20 20 20 20
0010'   20 20 20 20
0014'   48 49 53 54
0018'   4F 52 59 3A
001C'   20 4C 41 53
0020'   54 20 31 32
0024'   38 20 47 41
0028'   4D 45 53
002B'   20 20 20 20         DB      '                    '
```

```
002F'   20 20 20 20
0033'   20 20 20 20
0037'   20 20 20 20
003B'   20 20 20 20
003F'   20
0040'   20 20 20 20        00260   HISTRY: DB       '
0044'   20 20 20 20
0048'   20 20 20 20
004C'   20 20 20 20
0050'   20 20 20 20
0054'   20 20 20 20
0058'   20 20 20 20
005C'   20 20 20 20
0060'   20 20 20 20
0064'   20 20 20 20
0068'   20 20 20 20
006C'   20 20 20 20
0070'   20 20 20 20
0074'   20 20 20 20
0078'   20 20 20 20
007C'   20 20 20 20
0080'   20 20 20 20        00270           DB       '
0084'   20 20 20 20
0088'   20 20 20 20
008C'   20 20 20 20
0090'   20 20 20 20
0094'   20 20 20 20
0098'   20 20 20 20
009C'   20 20 20 20
00A0'   20 20 20 20
00A4'   20 20 20 20
00A8'   20 20 20 20
00AC'   20 20 20 20
00B0'   20 20 20 20
00B4'   20 20 20 20
00B8'   20 20 20 20
00BC'   20 20 20 20
00C0'   00                 00280           DB       0
                           00290   ; LARGE MESSAGES. 13 CHARACTERS PER LINE
00C1'   20 54 49 43        00300   MSG2:   DB       ' TIC-TAC-TOE ',0
00C5'   2D 54 41 43
00C9'   2D 54 4F 45
00CD'   20 00
00CF'   20 20 57 41        00310   MSG3:   DB       '  WAIT  ONE  ',0
00D3'   49 54 20 20
00D7'   4F 4E 45 20
```

```
00DB'  20 00
00DD'  20 20 59 4F     MSG4:   DB      '  YOUR MOVE  ',0
00E1'  55 52 20 4D
00E5'  4F 56 45 20
00E9'  20 00
00EB'  20 20 54 52     MSG5:   DB      '  TRY AGAIN  ',0
00EF'  59 20 41 47
00F3'  41 49 4E 20
00F7'  20 00
00F9'  20 20 59 4F     MSG6:   DB      '  YOU  WIN!  ',0
00FD'  55 20 20 57
0101'  49 4E 21 20
0105'  20 00
0107'  20 20 4F 4E     MSG7:   DB      '  ONE MORE?  ',0
010B'  45 20 4D 4F
010F'  52 45 3F 20
0113'  20 00
0115'  20 20 20 49     MSG8:   DB      '   I  WIN!   ',0
0119'  20 20 57 49
011D'  4E 21 20 20
0121'  20 00
0123'  20 20 20 20     MSG9:   DB      '    DRAW!    ',0
0127'  44 52 41 57
012B'  21 20 20 20
012F'  20 00
0131'  20 00           MSG10:  DB      ' ',0
0133'  2D 00           MSG11:  DB      '-',0
0135'  20 20 49 20     MSG12:  DB      '  I CONCEDE  ',0
0139'  43 4F 4E 43
013D'  45 44 45 20
0141'  20 00
                       ;
                       ; VARIABLES
                       ;
0143'  00              FRSTF:  DB      0             ;0=FIRST TIME;1=NOT
0144'  003F'           NXTHIS: DW      HISTRY-1      ;POINTER TO HISTORY LINES
0146'  0000            NOPTE:  DW      0             ;NUMBER PERMUTATION TB ENTRIES
0148'  0000            ROTPTR: DW      0             ;POINTS TO CURRENT ROTATION
014A'  00              NOX:    DB      0             ;NUMBER OF XS
014B'  00              NOO:    DB      0             ;NUMBER OF OS
014C'  00              NOSP:   DB      0             ;NUMBER OF SPACES
014D'  0000            TEMP1:  DW      0             ;TEMPORARY BUFFER
014F'  00              MOVENO: DB      0             ;MOVE #
0150'  1234 5678       SEED:   DW      1234H,5678H   ;RANDOM # SEED
0154'  0000            RINDW:  DW      0             ;POINTER TO INDICES
                       ;
```

```
                          00560  ; TABLES AND ARRAYS
                          00570  ;
0156'                     00580  ARRAY1: DS     9                ;ARRAY 1
015F'                     00590  ARRAY2: DS     9                ;ARRAY 2
0168'                     00600  ANALTB: DS     0                ;ANALYSIS TABLE
0168'   00                00610  NOR0:   DB     0                ;ROW 0
0169'   00                00620  NOR1:   DB     0                ;ROW 1
016A'   00                00630  NOR2:   DB     0                ;ROW 2
016B'   00                00640  NOC0:   DB     0                ;COLUMN 0
016C'   00                00650  NOC1:   DB     0                ;COLUMN 1
016D'   00                00660  NOC2:   DB     0                ;COLUMN 2
016E'   00                00670  NOD0:   DB     0                ;DIAGONAL 0
016F'   00                00680  NOD1:   DB     0                ;DIAGONAL 1
0170'                     00690  ROTTAB: DS     0                ;ROTATION TABLE
0170'   0000 0000         00700          DW     0,0,0,0,0,0,0,0 ;0,90,180,270+4 MIRRORS
0174'   0000 0000
0178'   0000 0000
017C'   0000 0000
0180'                     00710  MOVETB: DS     0                ;MOVE TABLE
0180'   0000 0000         00720          DW     0,0,0,0          ;MOVE 1,2,3,4 PNTRS
0184'   0000 0000
0188'   0000              00730          DW     0                ;TERMINATOR
                          00740  ; GRID TABLE. DEFINES TIC-TAC-TOE GRID
018A'                     00750  GRIDTB: DS     0
018A'   8C 00 22          00760          DB     8CH,0,34
018D'   3C8F              00770          DW     3C00H+128+15
018F'   8C 00 22          00780          DB     8CH,0,34
0192'   3D4F              00790          DW     3C00H+320+15
0194'   8C 00 22          00800          DB     8CH,0,34
0197'   3E0F              00810          DW     3C00H+512+15
0199'   8C 00 22          00820          DB     8CH,0,34
019C'   3ECF              00830          DW     3C00H+704+15
019E'   BC 01 01          00840          DB     0BCH,1,1
01A1'   3C8F              00850          DW     3C00H+128+15
01A3'   BC 01 01          00860          DB     0BCH,1,1
01A6'   3C9A              00870          DW     3C00H+128+26
01A8'   BC 01 01          00880          DB     0BCH,1,1
01AB'   3CA5              00890          DW     3C00H+128+37
01AD'   BC 01 01          00900          DB     0BCH,1,1
01B0'   3CB0              00910          DW     3C00H+128+48
01B2'   BF 01 09          00920          DB     0BFH,1,9
01B5'   3CCF              00930          DW     3C00H+192+15
01B7'   BF 01 09          00940          DB     0BFH,1,9
01BA'   3CDA              00950          DW     3C00H+192+26
01BC'   BF 01 09          00960          DB     0BFH,1,9
01BF'   3CE5              00970          DW     3C00H+192+37
```

```
01C1'   BF 01 09                      DB      0BFH,1,9
01C4'   3CF0                          DW      3C00H+192+48
01C6'   8F 01 01                      DB      8FH,1,1
01C9'   3ECF                          DW      3C00H+704+15
01CB'   8F 01 01                      DB      8FH,1,1
01CE'   3EDA                          DW      3C00H+704+26
01D0'   8F 01 01                      DB      8FH,1,1
01D3'   3EE5                          DW      3C00H+704+37
01D5'   8F 01 01                      DB      8FH,1,1
01D8'   3EF0                          DW      3C00H+704+48
01DA'   FF                            DB      0FFH
                              ; ROTATE INDICES TABLE. HOLDS INDICES FOR ROTATIONS
01DB'   00 01 02 03           RINDT:  DB      0,1,2,3,4,5,6,7,8  ;0 DEG
01DF'   04 05 06 07
01E3'   08
01E4'   02 05 08 01           RINDT9: DB      2,5,8,1,4,7,0,3,6  ;90 DEG
01E8'   04 07 00 03
01EC'   06
01ED'   08 07 06 05           .       DB      8,7,6,5,4,3,2,1,0  ;180
01F1'   04 03 02 01
01F5'   00
01F6'   06 03 00 07                   DB      6,3,0,7,4,1,8,5,2  ;270
01FA'   04 01 08 05
01FE'   02
01FF'   02 01 00 05                   DB      2,1,0,5,4,3,8,7,6  ;MIRROR 0 DEG
0203'   04 03 08 07
0207'   06
0208'   00 03 06 01                   DB      0,3,6,1,4,7,2,5,8  ;MIRROR 90
020C'   04 07 02 05
0210'   08
0211'   06 07 08 03                   DB      6,7,8,3,4,5,0,1,2  ;MIRROR 180
0215'   04 05 00 01
0219'   02
021A'   08 05 02 07                   DB      8,5,2,7,4,1,6,3,0  ;MIRROR 270
021E'   04 01 06 03
0222'   00
0223'   00 01 02 03           RINDTR: DB      0,1,2,3,4,5,6,7,8  ;INVERSE
0227'   04 05 06 07
022B'   08
022C'   06 03 00 07                   DB      6,3,0,7,4,1,8,5,2
0230'   04 01 08 05
0234'   02
0235'   08 07 06 05                   DB      8,7,6,5,4,3,2,1,0
0239'   04 03 02 01
023D'   00
023E'   02 05 08 01                   DB      2,5,8,1,4,7,0,3,6
```

```
0242'   04 07 00 03
0246'   06
0247'   02 01 00 05                       DB      2,1,0,5,4,3,8,7,6
024B'   04 03 08 07
024F'   06
0250'   00 03 06 01                       DB      0,3,6,1,4,7,2,5,8
0254'   04 07 02 05
0258'   08
0259'   06 07 08 03                       DB      6,7,8,3,4,5,0,1,2
025D'   04 05 00 01
0261'   02
0262'   08 05 02 07                       DB      8,5,2,7,4,1,6,3,0
0266'   04 01 06 03
026A'   00
                              ; STACK AREA
026B'                                     DS      100             ;100 BYTES
02CF'                         STACK       EQU     $               ;TOP OF STACK
                              ; PERMUTATION TABLE. HOLDS BASE 3 NUMBER REPRESENTING CONFIGURATIO
02CF'                         PTABLE:     DS      0
                                          END
```

**Figure 14-18.
Tic-Tac-Toe
Listings**

The last module of the listings is the SYSTEM module. SYSTEM contains all messages, variables, and tables for the program, since we don't know its length beforehand.

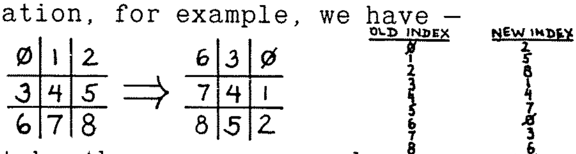
The last, and most important, table is PTABLE, the permutation table we've been talking about. Since this is generated completely by the program, it's simply defined as the last location of the program.

Moving backwards, we find RINDT. This is a "Rotation Indices" table used by subroutine ROTATE. It contains the indices for converting from the "standard" array into rotations and mirror images. The last half of the table contains the indices from reconversion.

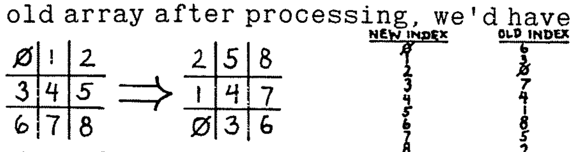
Hints and Kinks 14-4

Retranslation

We've talked about translating from one array to another one representing a rotation or mirror image. Unfortunately, retranslation back from the second to the first is not straightforward. To convert to a 90 degree rotation, for example, we have —



To take the new array and reconvert back into the old array after processing, we'd have —



The translation indices on reconversion bear some relationship to the conversion indices (each is 8-conversion index), but this relationship doesn't hold for other reconversions!

It's probably simplest to establish another table of reconversion indices, which we have done in the last half of RINDT.

There must be an easier way to implement this program! (Maybe you'll be the one to discover the way.)

The GRIDTB is a table of values used by the DRAWL subroutine to draw the tic-tac-toe "grid." Each five bytes defines one line.

The MOVETB table holds the address of each space cell used in a computer move. Each of the four entries may point to a space cell in the PTABLE so that it may be "adjusted" at the end of each game.

ANALTB is the "Analysis Table." It holds a count for each row, column, and diagonal of the current array. We'll discuss this under the ANALSR.

The *ROTTAB* table is the "Rotation Table." It holds the base three values for each of the eight rotations and mirror images of the current array. As these may be up to 19683, the entries are eight "words" of 16 bits.

There are 12 system messages at the beginning of SYSTEM. All of these are terminated by a 0.

The variables are described in the comments field of each variable, and we'll discuss them in the subroutine description.

Program Description

Here, as in MORG, we'll use a "bottom-up" approach to describe the system modules, starting from the least complex.

Fill Character Subroutine (FILLCH)

The FILLCH subroutine fills a given character into any area of memory. It decrements the byte count in BC down to zero and fills memory by using DE as a pointer register pair.

Delay Subroutine DELAY

The Delay subroutine delays from 1 through 65536 milliseconds, depending upon the count in the HL register pair. It is used only for large delays for display of system messages in tic-tac-toe.

Divide Subroutine (DIVIDE)

The Divide subroutine implements a divide of a 16-bit number in HL by an 8-bit number in E. The quotient is in HL and the remainder in BC on exit. DIVIDE is used by BINBAS for binary to base three conversion, although it is a general-purpose divide.

Base Three to Binary Subroutine (BASBIN)

BASBIN is a specialized subroutine for tic-tac-toe. It converts an array representing a tic-tac-toe configuration to a base three number. The array is either ARRAY1 or ARRAY2 in the system; each is a nine-byte table, holding a 0 for a space, 1 for an X, and 4 for an 0. At the end of the conversion, HL holds the base three number in binary, 0 through 19683. The array is organized as shown in Figure 14-19.

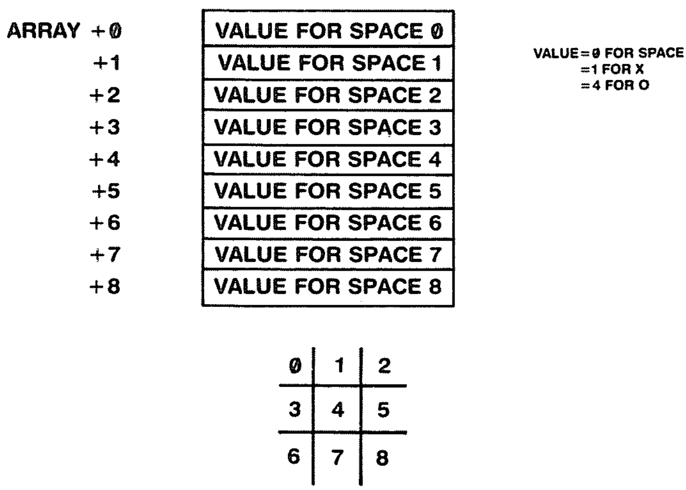


Figure 14-19. ARRAY1/2 Format

Binary to Base Three Subroutine (BINBAS)

The Binary to Base Three subroutine does the opposite of BASBIN — it converts a binary number of 0 through 19683 to nine base-three digits of 0, 1, or 2. It then converts the 2 to a 4 and stores the nine digits in a given array (ARRAY1 or ARRAY2). The binary number is the "current value" of the configuration.

Random Number Routine (RAND)

RAND generates a pseudo-random number from a "seed" value. Its operation is virtually identical to RAND in the MORG program. On exit, BC contains a random number from 0 through 65535. This number is used to "shuffle" the "balls," or space cells, for the current configuration, to choose **one** of the space cells for a computer move.

Input Subroutine (INPUSR)

The Input subroutine detects a key press of 0 through 9 and ignores all others. Tic-tac-toe uses only these keys for user input to define the play. A debounce of 100 milliseconds is performed by calling subroutine DELAY.

Display Message Subroutine (DSPMES)

DSPMES is a subroutine to display a given string of ASCII characters on the screen. On entry, HL points to the string, and BC points to the screen position. The routine picks up a character from the HL pointer and then outputs it by using BC as a pointer. The string is terminated on a zero (null) character. DSPMES is only called by HISTUP to output the "history" message along the bottom of the screen.

Large Character Display Subroutine (LARGEC)

This subroutine is a general-purpose subroutine to output "large" characters of 8 by 8 pixels to the screen. The subroutine is called with A containing the character to be output in ASCII and IY pointing to the upper left-hand pixel of the 8 by 8 block to be used.

The characters A through Z, space, "–", "?", and "!" may be output. They are defined in the DOTTAB table of LARGEC as an 8 by 8 dot matrix, reading from left to right and top to bottom.

Large Message Output Subroutine (MSGOUT)

MSGOUT is similar to DSPMES except that it calls LARGEC to output a large character message. On entry, HL points to the message and IY points to the start of the screen area for which the message is intended. IY will point to the upper left-hand pixel of the first character position of the message.

MSGOUT calls LARGEC until a terminating zero (null) character is detected in the message. Most tic-tac-toe messages are large character messages, except for the "history" message.

Display Screen Array Subroutine (SCRNDS)

SCRNDS is a specialized subroutine to convert an array (ARRAY1 or ARRAY2) to a tic-tac-toe grid with Xs and Os. The "grid" of the tic-tac-toe pattern is never rewritten during the program. SCRNDS, therefore, only fills in Xs and Os in the nine tic-tac-toe positions.

SCRNDS scans through the array from top to bottom and tests to see whether an X or O is present. If a blank is found, nothing is output for the position. If an X or O is found, SCRNDS calls MSGOUT with a "dummy" string consisting of an X or O followed by a zero (null). It puts this dummy string in the TEMP1 variable, a 16-bit variable allocated for this purpose.

Draw Line Subroutine (DRAWL)

The DRAWL subroutine draws either a vertical or horizontal line on the screen. The line is drawn through **character positions** rather than **pixel** positions. This makes the screen addressing a trivial rather than complex task.

On entry A contains the graphics character to be used. C contains a zero if a horizontal line is to be drawn, or a non-zero if a vertical line is to be drawn. HL contains the screen start position. B contains the number of character positions to be used.

DRAWL first decides whether a vertical or horizontal line is to be drawn. A branch is made to the proper code for each. If a horizontal line is to be drawn, A is stored indirect to

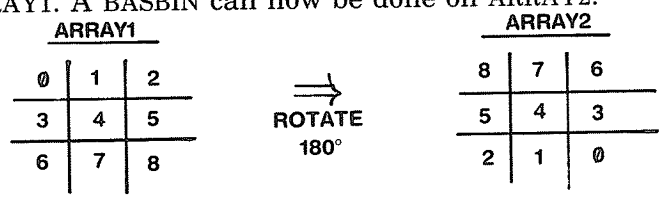
HL, HL is incremented, and B is decremented down to zero. If a vertical line is to be drawn, A is stored, HL is incremented by 64, and B is decremented down to zero. Note that vertical lines must start at the top position and horizontal lines must start at the left.

DRAWL is called by MAIN1. MAIN1 uses the GRIDTB table in the system to draw a complete set of lines representing the tic-tac-toe grid. The GRIDTB table is set up so that each entry is five bytes long, corresponding to values to be put into A, B, C, and HL. It calls DRAWL repeatedly until it has used every GRIDTB entry.

Array Translator Subroutine (ARRXLA)

The Array Translator Subroutine performs a rotation or mirror image translation from one array to the next. It is used in conjunction with BASBIN to find the new array and then to find the new base 3 value represented from the new array.

As an example, suppose that a rotation of 180 degrees is to be done on the current array. ARRXLA is entered with IX pointing to ARRAY1 in SYSTEM. IY points to ARRAY2 in SYSTEM. HL points to the list of 9 indices for the 180 degree rotation in SYSTEM at 1ED'. The nine indices of ARRAY1 are converted to the nine indices of ARRAY2 as shown in Figure 14-20. When the conversion is done, ARRAY2 holds a tic-tac-toe configuration rotated 180 degrees from ARRAY1. A BASBIN can now be done on ARRAY2.



ARRAY1 INDEX		ARRAY2 INDEX
0	→	8
1	→	7
2	→	6
3	→	5
4	→	4

5	→	3
6	→	2
7	→	1
8	→	0

Figure 14-20. Array Translation

ARRXLA is chiefly called by the ROTATE subroutine, which performs the seven rotations for finding the smallest value initially for the PTABLE, and then, later on, rotates the current tic-tac-toe array to find the smallest value for the PTABLE search.

Analyze Array (ANALAR)

ANALAR analyzes the current array to find eight **hash** values for the rows, columns, and diagonals of the given array. Although we've been discussing a base three value that represents the current configuration, the values actually held in the current (ARRAY1) or working (ARRAY2) arrays are actually 0 for space, 1 for X, and 4 for O. The reason for **these** values is that a simple add can give us unique values for the numbers of Xs, Os, and spaces in each row, column, or diagonal.

Hints and Kinks 14-5
Hash Values

A hashing technique uses an approach something like this: Is there a way to convert all possible actions to a series of unique numeric values that can then be used efficiently in processing?

An example from an assembler: Some assemblers add the ASCII characters in a symbol together to get a hash total. It turns out that this is (relatively) unique. The sum of "NAME", for example, is different from "START". This one-byte hash value can then be used in a more efficient search of the symbol table than a six- or seven-byte string comparison; this speeds up symbol table searches and reduces assembly time.

If the row, column, or diagonal has three spaces, the hash value will be 0. If it has two spaces and an X, the hash will be 1. Two spaces and an O yields a 4. One space and two Xs gives a 2. One space and two Os gives an 8. One space and an X and O gives a 5. No spaces and three Xs gives a 3. No spaces and two Xs and one O gives 6. No spaces and one X and two Os gives 9. No spaces and three Os gives 12. All of these values are **unique** and represent only one defined configuration. A test can, therefore, be easily made for "two Xs and a space," or "three Os." This greatly simplifies testing for tic-tac-toe conditions.

At the end of ANALAR, the eight hash values are saved in the ANALTB (Analyze Table) of SYSTEM.

Number Subroutine (NUMBER)

The NUMBER subroutine also helps to analyze an array. It counts the number of Xs, Os, and spaces in the given array (ARRAY1 or ARRAY2). The number of each is put in variables NOX, NOO, or NOSP in SYSTEM. NUMBER is called by MAIN1 to analyze the configuration for PTABLE.

Rotate Subroutine (ROTATE)

ROTATE calls ARRXLA eight times to perform the rotations and mirror image translations of the current array. After each call, the translated array is converted to a numeric value by BASBIN. This numeric value is then put in ROTTAB in SYSTEM. At the end of ROTATE, the eight base three values are in ROTTAB and the DE register holds the lowest value. This lowest value is then used either to establish the PTABLE entry (MAIN1) or for the search of PTABLE.

History Update (HISTUP)

HISTUP is called at the end of the game with a L (lose), W (win), D (draw), or C (concede) in the A register. This code is then put in the last position of the history buffer, and the entire history message is then output to show the history of the last 128 games. If 128 have been played, HISTUP "slides" the last 127 games into the first 127 game positions by an LDIR and then stores the current game code in the 128th position.

Memory Subroutine (MEMORY)

MEMORY implements the "reward"/"punishment" action at the end of each game. It is entered with a -1 in A for a lost game, a 3 for a win, and a 1 for a draw.

MEMORY then uses the MOVETB, which has a record of the space cell addresses, to add or subtract counts from the space cells that were used in the games. A comparison is made for increments over 99. If the count is greater than 99 (on a win or draw), the space cell is set to 99. A comparison is also made for decrements below 0. If below 0 (on a loss), the space cell is set to 0.

Main Driver Modules

There are four main driver modules, named MAIN1, MAIN2, MAIN3, and MAIN4. MAIN1 is used to perform "first time" actions, primarily to build up the PTABLE. Since this action takes several minutes, it is only done one time on each load. MAIN2 through MAIN4 are used to implement the actual game-playing.

MAIN1

MAIN1 first clears the screen with graphics 80H characters (FILLCH) and then displays the history message (DSPMES). Initially the history message will be filled with blanks, since no games have been played. DRAWL is then used to draw the grid.

Next, the Move Table MOVETB is cleared. An entry of 0 is used as a **terminator** since 0 is not a valid address for a PTABLE entry.

Now the First Time Flag (FRSTF) is tested to see if this is the first time through the program. If not, the program goes on to MAIN2 actions. If it is the first time, the history line is filled with blanks (to handle any restart of an already executed program), and a `WAIT ONE` message is displayed on the screen (MSGOUT).

Now the Permutation Table (PTABLE) is generated. The algorithm used is identical to that described above under "Generation of a Permutation Table." A count is incremented from 000000000 through 222222222 in binary form. BINBAS is used to find the equivalent tic-tac-toe array. NUMBER is called to count the number of Xs, Os, and spaces. ANALAR is called to analyze the eight rows, columns, and diagonals.

If the array passes all of the tests, ROTATE is then called to find the lowest value of the eight possible rotations and mirror images. If the current array is the one producing the lowest value, a new PTABLE entry is made. The number of spaces in the array is then counted and stored in the next byte. Space cells are allocated dependent upon the number of spaces. The number of spaces is then divided by 2 to produce the initial count to be put in each space cell. The quotient of this divide is conveniently equal to 4,3,2, or 1 per the algorithm.

If an entry in PTABLE is made, a dash is alternately blinked on and off at the end of the WAIT ONE message to show processing activity.

MAIN2

MAIN2 is entered at the end of PTABLE computations for the first time or reentered for the main loop in playing games. The main loop is ART070, entered from MAIN1 for each new game. Location MAINLP is reentered for each new move of a game.

If this is a start of a new game, ART070 outputs the title message and then clears the main array, ARRAY1 (FILLCH). MAINLP is then entered. When MAINLP is entered, the game has either just started or has been going on for a number of moves. In either case the action is the same. A call is made to ANALAR to analyze ARRAY1.

If two Xs exist, the computer finishes the game in the code at ART102. To do this, however, it must try an X in spaces until it finds one that produces an analyzed value of 3 (three Xs). It laboriously calls ANALAR after each try.

When the proper place for the X is found, it outputs an I WIN message and goes to the "end action" code at MAINEA in MAIN4.

If two Xs are not present, the program then tests the Move Number variable MOVENO for 5. MOVENO represents the number of times the computer has played and is incremented after each computer play. If this is the last (5th) move, the computer plays in the only space, and then tests for a win. If there's a win, it goes to the action in ART106. If there's a draw (there cannot be a loss), it outputs a DRAW! message and goes to the end action code in MAINEA.

If ART114 is entered, none of the above applies. ROTATE is now called to find the eight rotation values. ROTATE returns the smallest value in DE. This is the PTABLE value that will be used for the search. A search is now made (ART120). The PTABLE value must be found if PTABLE has been constructed properly. After the value is found, MAIN3 is entered.

MAIN3

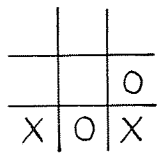
With the PTABLE entry found, the program now must handle two arrays. The first, in ARRAY1, is the present tic-tac-toe array. The second, in ARRAY2, represents the translated array from PTABLE. If the PTABLE value was the same as the present array, both arrays will be identical (0 degrees rotation), otherwise ARRAY2 will hold a rotated or mirror image array.

ARRXLA is called to convert the present array to the rotation or mirror image in ARRAY2. Then ARRAY2 is analyzed by ANALAR. If two Os are found, the computer "blocks" the move by putting in an X in the proper space. The code at ART102 is used to find the space which produces the proper block, and the code at ART106 finds the proper space cell for the blocking move. After the block, a JP is made to ART172 for further processing.

If a blocking move is not possible, the code at ART125 is performed. This code finds the total count of all space cells associated with the permutation. If this total count is zero, this path is hopeless and will be deleted. The MOVETB is used to find the last move, and the space cell for the last move is zeroed. An `I CONCEDE` message is then output, and the end action at MAINEA is performed. This concession action will rarely happen.

If the total count is non-zero, a random number **less than or equal to** the total count is found by calling RAND. The space cells are totaled until the total is equal to or greater than the random number. This action is analogous to stirring the balls in the box and picking one! (See Figure 14-21.)

CONFIGURATION:



PTABLE ENTRY:

46H (000 002 121)	CONFIGURATION IN BASE 3
5	# OF SPACE CELLS
10	COUNT FOR SPACE 0
2	COUNT FOR SPACE 1
7	COUNT FOR SPACE 2
1	COUNT FOR SPACE 3
1	COUNT FOR SPACE 4

PICKING A SPACE:

1. TOTAL COUNT OF SPACE CELLS = 10+2+7+1 = 21
2. FIND RANDOM # <=21 : 12
3. O ⟶ TOTAL
4. ADD SPACE CELL 0 : TOTAL = 10, < RANDOM # OF 12
5. ADD SPACE CELL 1 : TOTAL = 12, = RANDOM # OF 12
6. SPACE CELL 1 WILL BE USED FOR MOVE, CAUSING X TO BE PUT INTO SPACE 1

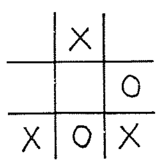


Figure 14-21. Using RAND To Pick A Space

Now an X is stored in the array (ARRAY2), and a record is made of the space cell move in MOVETB. We worked with ARRAY2 and then converted it back to ARRAY1 by a call to ARRXLA. RINDW holds the pointer to the "reverse indices." SCRNDS is then called to display ARRAY1, and MAIN4 is entered.

MAIN4

MAIN4 is used to get the human input. It outputs the YOUR MOVE message and waits for user input from INPUT. A check is made of the validity of the number of the square chosen and a TRY AGAIN message output.

When the human inputs a valid move, an O is stored in ARRAY1 and SCRNDS is called to display the array. ANALAR is then called (ART185) to analyze the new array. Only a human win can occur at this point. If the human wins, a YOU WIN message is output and the end action at MAINEA is performed. If no win occurred, the next computer move must be done and MAINLP in MAIN2 is reentered.

The end action at MAINEA is entered at the completion of every game. Before this point the HISTUP (History Update) and MEMORY routines have been called to update the history message and "reward" or "punish" the PTABLE space cell. MAINEA outputs a ANOTHER? message and waits for a key press. On the key press, ARTIP in MAIN1 is reentered for a new game.

Using This Program

As in the case of MORG, this program was designed and coded to let you see a significant chunk of presumably worthwhile code. If you'd like to experiment with the program, you may enter the machine code by using Disk DEBUG or T-BUG. Figure 14-22 gives the machine code after the program load. There are several thousand bytes, but a fast typist could enter them in an hour. **Checkpoint** by saving partial results! You can do this by dumping to disk (by DUMP) or to cassette (P command in T-BUG) and then reloading to take up where you left off.

```
8000    31 E7 8A 3E 80 11 00 3C 01 40 03 CD 0B 88 21 18
8010    88 01 40 3F CD 13 87 DD 21 A2 89 DD 7E 00 FE FF
8020    28 16 DD 4E 01 DD 46 02 DD 6E 03 DD 66 04 CD 42
8030    85 01 05 00 DD 09 18 E3 AF 11 98 89 01 08 00 CD
8040    0B 88 32 67 89 3A 5B 89 B7 C2 2C 81 3C 32 5B 89
8050    11 57 88 ED 53 5C 89 13 3E 20 01 80 00 CD 0B 88
8060    21 D9 88 FD 21 00 3C CD A2 85 21 B8 0B CD F8 87
8070    21 E7 88 FD 21 00 3C CD A2 85 21 E7 8A E5 21 00
8080    00 22 5E 89 E5 01 E3 4C B7 ED 42 CA 29 81 09 DD
8090    21 6E 89 CD 83 87 22 88 89 21 6E 89 CD 7E 84 3A
80A0    62 89 57 3A 63 89 BA 28 03 E1 18 79 FE 04 28 F9
80B0    DD 21 6E 89 CD AC 84 06 08 21 80 89 7E FE 03 CA
80C0    A9 80 FE 0C CA A9 80 FE 02 CA A9 80 23 10 ED CD
80D0    16 84 E1 E5 B7 ED 52 20 D0 E1 DD E1 DD 75 00 DD
80E0    74 01 FD 2A 5E 89 FD 23 FD 22 5E 89 3A 64 89 4F
80F0    DD 71 02 DD 23 DD 23 DD 23 DD E5 D1 06 00 DD 09
8100    DD E5 79 CB 3F D5 CD 0B 88 3A 5E 89 E6 01 21 49
8110    89 28 03 21 4B 89 FD 21 3C 3C CD A2 85 DD E1 DD
8120    6E FD DD 66 FE 23 C3 84 80 E1 DD E1 21 D9 88 FD
8130    21 00 3C CD A2 85 21 B8 0B CD F8 87 01 09 00 AF
8140    11 6E 89 CD 0B 88 21 67 89 34 DD 21 6E 89 CD AC
8150    84 21 80 89 06 08 7E FE 02 28 05 23 10 F8 18 3F
8160    E5 21 6E 89 7E B7 28 03 23 18 F9 E5 3C 77 DD 21
8170    6E 89 CD AC 84 E1 DD E1 DD 7E 00 FE 03 28 06 AF
8180    77 DD E5 18 E3 CD 5D 85 3E 57 CD E3 83 3E 03 CD
8190    B1 83 21 2D 89 FD 21 00 3C CD A2 85 C3 9B 83 3A
81A0    67 89 FE 05 20 3A 21 6E 89 7E B7 28 03 23 18 F9
81B0    3C 77 DD 21 6E 89 CD AC 84 21 6E 89 06 08 7E FE
81C0    03 28 C2 23 10 F8 CD 5D 85 3E 44 CD E3 83 3E 01
81D0    CD B1 83 21 3B 89 FD 21 00 3C CD A2 85 C3 9B 83
81E0    FD 21 6E 89 CD AA 87 22 88 89 CD 16 84 DD 21 E7
81F0    8A DD 6E 00 DD 66 01 B7 ED 52 28 0C DD 4E 02 06
8200    00 03 03 03 DD 09 18 E9 DD E5 2A 60 89 01 88 89
8210    B7 ED 42 CB 3D CB 3C 30 02 CB FD E5 29 29 29 C1
8220    09 E5 01 F3 89 09 DD 21 6E 89 FD 21 77 89 CD 17
8230    85 E1 01 3B 8A 09 22 6C 89 DD 21 77 89 CD AC 84
8240    21 80 89 06 08 7E FE 08 28 05 23 10 F8 18 52 E5
8250    21 77 89 7E B7 28 03 23 18 F9 E5 3C 77 DD 21 77
8260    89 CD AC 84 E1 DD E1 DD 7E 00 FE 09 28 06 AF 77
8270    DD E5 18 E3 AF 77 E5 01 76 89 B7 ED 42 11 FF FF
8280    45 21 77 89 7E 23 B7 20 01 13 10 F8 E1 3E 01 77
8290    DD E1 DD 19 DD 23 DD 23 DD 23 DD 23 DD E5 C3 1E
82A0    83 DD E1 DD E5 DD 46 02 21 00 00 DD 5E 03 16 00
82B0    19 DD 23 10 F6 7C B5 DD E1 DD E5 20 25 3A 67 89
82C0    3D 07 4F 06 00 DD 21 98 89 DD 09 DD 66 00 DD 6E
82D0    01 AF 77 21 4D 89 FD 21 00 3C CD A2 85 3E 43 C3
82E0    93 83 CD 46 87 C5 D1 EB B7 ED 52 30 FB 19 DD 4E
82F0    03 06 00 B7 ED 42 DD 23 28 02 30 F2 D1 DD E5 E1
8300    01 03 00 DD 09 DD E5 B7 ED 52 45 DD 21 77 89 DD
8310    7E 00 B7 DD 23 20 F8 10 F6 3E 01 DD 77 FF 3A 67
8320    89 3D 07 5F 16 00 21 98 89 19 D1 1B 73 23 72 2A
8330    6C 89 DD 21 77 89 FD 21 6E 89 CD 17 85 CD 5D 85
8340    21 F5 88 FD 21 00 3C CD A2 85 CD 23 87 4F 06 00
8350    21 6E 89 09 7E B7 28 12 21 03 89 FD 21 00 3C CD
8360    A2 85 21 B8 0B CD F8 87 18 D6 3E 04 77 CD 5D 85
8370    DD 21 6E 89 CD AC 84 21 80 89 06 08 7E 23 FE 0C
8380    28 05 10 F8 C3 46 81 21 11 89 FD 21 00 3C CD A2
8390    85 3E 4C CD E3 83 3E FF CD B1 83 21 B8 0B CD F8
83A0    87 21 1F 89 FD 21 00 3C CD A2 85 CD 23 87 C3 00
83B0    80 F5 E5 DD E5 DD 21 98 89 DD 66 01 DD 6E 00 F5
83CC    7C B5 28 19 F1 F5 86 CA CD 83 F2 CE 83 AF FE 64
83D0    FA D5 83 3E 63 77 DD 23 DD 23 F1 18 DC F1 DD E1
83E0    E1 F1 C9 C5 D5 E5 2A 5C 89 23 22 5C 89 01 D8 88
83F0    B7 ED 42 20 10 0B ED 43 5C 89 21 59 88 11 58 88
8400    01 7F 00 ED B0 2A 5C 89 77 21 18 88 01 40 3F CD
8410    13 87 E1 D1 C1 C9 F5 C5 E5 DD E5 FD E5 06 07 DD
8420    21 FC 89 FD 21 88 89 FD 23 FD 23 FD E5 DD E5 DD
8430    21 6E 89 FD 21 77 89 E1 E5 CD 17 85 CD AA 87 DD
8440    E1 FD E1 FD 75 00 FD 74 01 FD 23 FD 23 11 09 00
8450    DD 19 10 D7 06 08 DD 21 88 89 11 FF 7F DD 66 01
8460    DD 6E 00 B7 ED 52 F2 70 84 19 54 5D DD 22 60 89
8470    DD 23 DD 23 10 E7 FD E1 DD E1 E1 C1 F1 C9 F5 C5
8480    D5 E5 06 09 11 00 00 0E 00 7E FE 01 20 01 14 FE
8490    04 20 01 1C B7 20 01 0C 23 10 EE 7A 32 62 89 7B
84A0    32 63 89 79 32 64 89 E1 D1 C1 F1 C9 F5 FD E5 FD
84B0    21 80 89 DD 7E 00 DD 86 01 DD 86 02 FD 77 00 DD
84C0    7E 03 DD 86 04 DD 86 05 FD 77 01 DD 7E 06 DD 86
84D0    07 DD 86 08 FD 77 02 DD 7E 00 DD 86 03 DD 86 06
84E0    FD 77 03 DD 7E 01 DD 86 04 DD 86 07 FD 77 04 DD
```

```
84F0   7E 02 DD 86 05 DD 86 08 FD 77 05 DD 7E 00 DD 86
8500   04 DD 86 08 FD 77 06 DD 7E 02 DD 86 04 DD 86 06
8510   FD 77 07 FD E1 F1 C9 F5 C5 D5 E5 DD E5 FD E5 06
8520   09 FD E5 5E 16 00 FD E1 FD E5 FD 19 DD 7E 00 FD
8530   77 00 23 DD 23 10 EC FD E1 FD E1 DD E1 E1 D1 C1
8540   F1 C9 C5 D5 E5 F5 79 B7 20 07 F1 77 23 10 FC 18
8550   08 F1 11 40 00 77 19 10 FC E1 D1 C1 C9 F5 C5 D5
8560   E5 FD E5 21 6E 89 06 09 FD 21 D3 3C 7E B7 28 15
8570   FE 01 20 04 3E 58 18 02 3E 4F 32 65 89 E5 21 65
8580   89 CD A2 85 E1 23 11 0B 00 FD 19 78 FE 07 28 04
8590   FE 04 20 05 11 9F 00 FD 19 10 D1 FD E1 E1 D1 C1
85A0   F1 C9 F5 C5 E5 FD E5 7E B7 28 0B CD BC 85 01 C1
85B0   FF FD 09 23 18 F1 FD E1 E1 C1 F1 C9 F5 C5 E5 DD
85C0   E5 DD 21 05 86 DD BE 00 DD 23 20 F9 DD E5 E1 01
85D0   06 86 B7 ED 42 E5 DD E1 DD 29 DD 29 DD 29 01 23
85E0   86 DD 09 CD F4 85 01 3C 00 FD 09 CD F4 85 DD E1
85F0   E1 C1 F1 C9 06 04 DD 7E 00 CB FF FD 77 00 DD 23
8600   FD 23 10 F2 C9 41 42 43 44 45 46 47 48 49 4A 4B
8610   4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 20
8620   2D 3F 21 17 03 03 2B 17 03 03 2B 17 03 03 2B 37
8630   33 33 3B 17 03 03 0B 35 30 30 38 17 03 03 29 35
8640   30 30 1A 17 03 03 03 37 33 33 30 17 03 03 03 17
8650   03 03 01 17 03 03 0B 35 30 30 3B 15 00 00 2A 17
8660   03 03 2B 00 2B 17 00 00 3A 35 00 00 00 00 2B 34
8670   30 30 38 15 00 20 06 17 03 03 24 15 00 00 00 35
8680   30 30 30 1F 10 20 2F 15 02 01 2A 17 24 00 2A 15
8690   00 09 3A 16 03 03 29 25 30 30 1A 17 03 03 2B 17
86A0   03 03 03 16 03 03 29 25 30 38 1A 17 03 03 2B 17
86B0   03 03 25 17 03 03 03 33 33 33 3B 03 2B 17 03 00
86C0   2A 15 00 15 00 00 2A 35 30 30 3A 15 00 00 2A 02
86D0   24 18 01 15 28 14 2A 25 3A 35 1A 09 30 30 06 18
86E0   03 03 24 15 00 00 2A 03 2B 17 03 03 03 33 0F 3C
86F0   33 30 30 00 00 00 00 00 00 00 00 00 00 00 00 03
8700   03 03 03 07 23 33 3B 00 22 11 00 00 2A 15 00 00
8710   22 11 00 F5 C5 E5 7E B7 28 05 02 03 23 18 F7 E1
8720   C1 F1 C9 C5 E5 3A 10 38 B7 28 09 0E FF 0C 0F 30
8730   FC 79 18 09 3A 20 38 E6 01 28 EA 3E 08 21 64 00
8740   CD F8 87 E1 C1 C9 F5 D5 E5 ED 5B 68 89 2A 6A 89
8750   06 07 CD 6B 87 10 FB 06 03 CD 71 87 10 FB ED 53
8760   68 89 22 6A 89 43 4C E1 D1 F1 C9 29 EB ED 6A EB
8770   C9 C5 ED 4B 6A 89 B7 ED 42 EB ED 4B 68 89 ED 42
8780   EB C1 C9 F5 C5 E5 DD E5 01 08 00 DD 09 1E 03 06
8790   09 C5 CD D0 87 79 FE 02 20 02 0E 04 DD 71 00 DD
87A0   2B C1 10 ED DD E1 E1 C1 F1 C9 F5 C5 D5 FD E5 21
87B0   00 00 06 09 E5 D1 19 19 FD 7E 00 FE 04 20 02 3E
87C0   02 85 6F 30 01 24 FD 23 10 EA FD E1 D1 C1 F1 C9
87D0   D5 DD E5 E5 DD E1 21 00 00 16 00 06 10 29 DD 29
87E0   30 01 23 DD 23 B7 ED 52 30 03 19 DD 2B 10 EE E5
87F0   C1 DD E5 E1 DD E1 D1 C9 C5 D5 E5 11 FF FF 2B 06
8800   83 10 FE 19 DA FF 87 E1 D1 C1 C9 12 13 0B F5 78
8810   B1 28 03 F1 18 F5 F1 C9 20 20 20 20 20 20 20 20
8820   20 20 20 20 20 20 20 20 20 20 20 20 48 49 53 54
8830   4F 52 59 3A 20 4C 41 53 54 20 31 32 38 20 47 41
8840   4D 45 53 20 20 20 20 20 20 20 20 20 20 20 20 20
8850   20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
8860   20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
8870   20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
8880   20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
8890   20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
88A0   20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
88B0   20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
88C0   20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
88D0   20 20 20 20 20 20 20 20 00 20 54 49 43 2D 54 41
88E0   43 2D 54 4F 45 20 00 20 20 57 41 49 54 20 20 4F
88F0   4E 45 20 20 00 20 20 59 4F 55 52 20 4D 4F 56 45
8900   20 20 00 20 20 54 52 59 20 41 47 41 49 4E 20 20
8910   00 20 20 59 4F 55 20 20 57 49 4E 21 20 20 00 20
8920   20 4F 4E 45 20 4D 4F 52 45 3F 20 20 00 20 20 20
8930   49 20 20 57 49 4E 21 20 20 20 00 20 20 20 20 44
8940   52 41 57 21 20 20 20 20 00 20 00 2D 00 20 20 49
8950   20 43 4F 4E 43 45 44 45 20 20 00 00 57 88 00 00
8960   00 00 00 00 00 00 00 00 34 12 78 56 00 00 00 00
8970   00 00 00 00 00 00 00 00 00 23 FF FF FF FF FF FF
8980   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8990   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
89A0   00 00 8C 00 22 8F 3C 8C 00 22 4F 3D 8C 00 22 0F
89B0   3E 8C 00 22 CF 3E BC 01 01 8F 3C BC 01 01 9A 3C
89C0   BC 01 01 A5 3C BC 01 01 B0 3C BF 01 09 CF 3C BF
89D0   01 09 DA 3C BF 01 09 E5 3C BF 01 09 F0 3C 8F 01
```

```
89E0   01 CF 3E 8F 01 01 DA 3E 8F 01 01 E5 3E 8F 01 01
89F0   F0 3E FF 00 01 02 03 04 05 06 07 08 02 05 08 01
8A00   04 07 00 03 06 08 07 06 05 04 03 02 01 00 06 03
8A10   00 07 04 01 08 05 02 02 01 00 05 04 03 08 07 06
8A20   00 03 06 01 04 07 02 05 08 06 07 08 03 04 05 00
8A30   01 02 08 05 02 07 04 01 06 03 00 00 01 02 03 04
8A40   05 06 07 08 06 03 00 07 04 01 08 05 02 08 07 06
8A50   05 04 03 02 01 00 02 05 08 01 04 07 00 03 06 02
8A60   01 00 05 04 03 08 07 06 00 03 06 01 04 07 02 05
8A70   08 06 07 08 03 04 05 00 01 02 08 05 02 07 04 01
8A80   06 03 00 FF FF FF FF FF FF FF FF FF FF FF FF FF
8A90   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
8AA0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
8AB0   FF FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00
8AC0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8AD0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8AE0   00 00 00 00 00 00 00 00
```

Figure 14-22. Machine Code to Tic-Tac-Toe

> **Hints and Kinks 14-6**
> **Checkpointing**
>
> How often should you checkpoint? I'm from the old school – I never trust computers. As a matter of fact, you may employ Barden's Law: The more you checkpoint, the less you'll need it! If you don't checkpoint, an operator error or power failure will surely wipe out several hours of work. But seriously – the TRS-80 is much less prone to losing data than minicomputers of several years ago! And reliability will continue to get better with newer hardware.

You may also key in the source for the program and use the Disk Assembler to assemble and load your own version for experimentation. This is quite a task, but it's certainly possible, especially since the program is modular. EDTASM may also be used to assemble one huge program.

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 INC IY
Memory
DEC HL DEC (IX+d) DEC (IY+d)
INC (HL) INC (IX+d) INC (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 JR NC,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)
BCD   RLD   RRD

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 (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   LD (nn),HL

Subroutine Action

Conditional CALLs   CALL cc,nn

Unconditional CALLs   CALL nn

Conditional Return   RET cc

Unconditional Return   RET

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/0 input, repeat	●	●	●
INI	11101101	10100010		Block I/O input from (C)	●	●	●
INIR	11101101	10110010		Block I/0 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				●

Mnemonic	Format				Description	S	Z	P/V	C
RLD	11101101	01101111			Rotate bcd digit left (HL)	●	●	●	
RR r	11001011	00011 r			Rotate right thru carry r	●	●	●	●
RR (HL)	11001011	00011110			Rotate right thru carry (HL)	●	●	●	●
RR (IX+d)	11011101	11001011	d	00011110	Rotate right thru cy (IX+d)	●	●	●	●
RR (IY+d)	00011110	11001011	d	00011110	Rotate left thru cy (IY+d)	●	●	●	●
RRA	00011111				Rotate A right thru carry				●
RRC r	11001011	00001 r			Rotate r right circular	●	●	●	●
RRC (HL)	11001011	00001110			Rotate (HL) right circular	●	●	●	●
RRC (IX+d)	11011101	11001011	d	00001110	Rotate (IX+d) right circular	●	●	●	●
RRC (IY+d)	11111101	11001011	d	00001110	Rotate (IY+d) right circular	●	●	●	●
RRCA	00001111				Rotate A right circular				●
RRD	11101101	01100111			Rotate bcd digit right (HL)	●	●	●	
RST p	11 t 110				Restart to location p				
SBC A,r	10011 r				A-r-CY to A	●	●	●	●
SBC A,n	11011110	n			A-n-CY to A	●	●	●	●
SBC A,(HL)	10011110				A-(HL)-CY to A	●	●	●	●
SBC A,(IX+d)	11011101	10011110	d		A-(IX+d)-CY to A	●	●	●	●
SBC A,(IY+d)	11111101	10011110	d		A-(IY+d)-CY to A	●	●	●	●
SBC HL,ss	11101101	01ss0010			HL-ss-CY to HL	●	●	●	●

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	●	●	●	●

Mnemonic	Format			Description	S	Z	P/V	C
SUB (IX+d)	11011101	10010110	d	A-(IX+d) to A	●	●	●	●
SUB (IY+d)	11111101	10010110	d	A-(IY+d) to A	●	●	●	●
XOR r	10101 r			A EXCLUSIVE OR r to A	●	●	●	0
XOR n	11101110	n		A EXCLUSIVE OR n to A	●	●	●	0
XOR (HL)	10101110			A EXCLUSIVE OR (HL) to A	●	●	●	0
XOR (IX+d)	11011101	10101110	d	A EXCLUSIVE OR (IX+d) to A	●	●	●	0
XOR (IY+d)	11111101	10101110	d	A EXCLUSIVE OR (IY+d) to A	●	●	●	0

Key:

Instruction Fields:

- b bit field 0-7
- c condition field 0=NZ, 1=Z, 2=NC, 3=C, 4=PO, 5=PE, 6=P, 7=M
- d indexing displacement +127 to −128
- dd register pair: 0=BC, 1=DE, 2=HL, 3=SP
- e relative jump displacement +127 to −128
- n immediate or address value
- pp register pair: 0=BC, 1=DE, 2=IX, 3=SP
- qq register pair: 0=BC, 1=DE, 2=IY, 3=SP
- r register: 0=B, 1=C, 2=D, 3=E, 4=H, 5=L, 7=A
- r′ register: same as r
- ss register pair: 0=BC, 1=DE, 2=HL, 3=SP
- t RST field: Location=t*8

Condition Codes:

- ● = affected
- 0 = reset
- 1 = set
- = unaffected

INDEX

Radio Shack

More TRS-80 Assembly Language Programming

Cat. No. 62-2075

Radio Shack
A Division of Tandy Corporation
Fort Worth, Texas 76102