A REWARD BOOK

ASSEMBLY LANGUAGE GRAPHICS for the TRS-80 COLOR COMPUTER

Don Inman Kurt Inman
with Dymax

Assembly Language Graphics for the TRS-80 Color Computer

Don Inman and Kurt Inman
with Dymax

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

Library of Congress Cataloging in Publication Data
Inman, Don.
Assembly language graphics for the TRS-80 color computer.
Includes index.
1. TRS-80 (Computer)--Programming. 2. Assembler language (Computer program language) 3. Computer graphics. I. Inman, Kurt. II. Dymax (Firm)
III. Title.
QA76.8.T18I54 1983 001.64'43 82-16548
ISBN 0-8359-0318-4
ISBN 0-8359-0317-6 (pbk.)

A Prentice-Hall Company
Reston, Virginia 22090

10 9 8 7 6 5 4 3 2

Printed in the United States of America

Contents

Preface

This book is specific to the TRS-80 Color Computer with applications using sound and graphics to illustrate how an assembler can be used to perform feats that would be quite difficult, if not impossible, in BASIC language.

Rather than introduce machine and assembly languages through a mathematical approach, we have chosen sound and graphics as the vehicle for learning.

Computer architecture and number systems are discussed only when necessary. For more technical information on the 6809 assembly language, there are other sources of more detailed information. More complete information on binary, hexadecimal, and other number systems is also available elsewhere.

We feel that it is valuable in the learning process to perform an act as well as to read about how it is done. Therefore, we have written this book as a "doing" experience. You are often encouraged to run programs before they are thoroughly explained. Instructions are explained in the context of their use with a minimum of detail. You are encouraged to go beyond the demonstrations presented to gain a more complete understanding of assembly language programming.

The book is not intended to cover the entire field of 6809 assembly language programming. It does provide an introduction to assembly language as implemented on the TRS-80 Color Computer.

We have used two tools in writing this book that are produced by The Micro Works of Del Mar, California. Their CBUG machine language monitor was used to develop Chapters 1 and 2. Their SDS80C Software Development System was used in the remainder of the book to edit, assemble, and debug programs. We highly recom-

mend that some assembler be used in conjunction with reading the book. The one that you use may differ in detail from the SDS80C system, but the basic techniques will be the same.

We have also made use of information from articles specific to the TRS-80 Color Computer that have appeared in the magazine, The *Color Computer News.*

In the last chapter we have made use of a PROM Programmer from Spectral Associates in Tacoma, Washington, to show how you may write your own software and save it permanently on ROM.

A summary and test are provided at the end of each chapter. Answers are provided for the odd-numbered test exercises.

Chapter 1

Introduction to Machine Language

Although machine language programming may be a more time-consuming and detailed task than programming in BASIC or some other high level language, it brings you into much closer contact with the computer. When you speak to the computer in machine language, you are talking to it directly. You will get quick responses and will gain a better understanding of your computer's "personality," its full capabilities, and also its shortcomings. You will find that the computer speaks and understands a very limited, formal language. Each word is the same length and follows a rigid format. However, its rules of form and syntax are much simpler than the English language.

The computer can be imagined as a gigantic array of electronic switches that are either opened or closed. Machine language instructions are formed by groups of eight switches. A different pattern is formed by opening or closing different combinations of these eight switches. The computer recognizes each different pattern as a distinct instruction or piece of data.

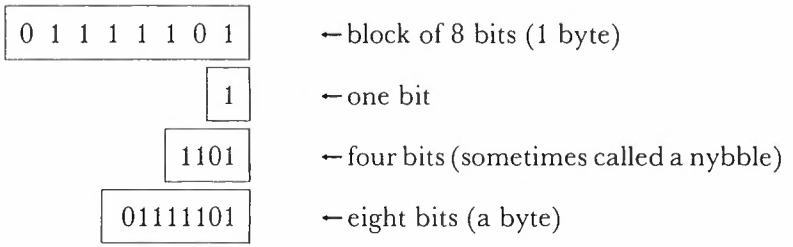
To communicate with the computer, we use a system of binary symbols, 0 for switch open and 1 for switch closed. A group of eight of these symbols tells the computer the switch pattern that is desired. Each of the symbols is called a bit (binary digit). Therefore, we need to learn this symbolism if we are to communicate directly with the computer.

An example of a pattern of eight computer bits is

0 1 0 1 1 1 0 0

The computer would recognize this pattern as a unique instruction or piece of data. It would respond by taking the specified action requested by the pattern of the instruction or by using the piece of data in the way that the previous instruction had requested.

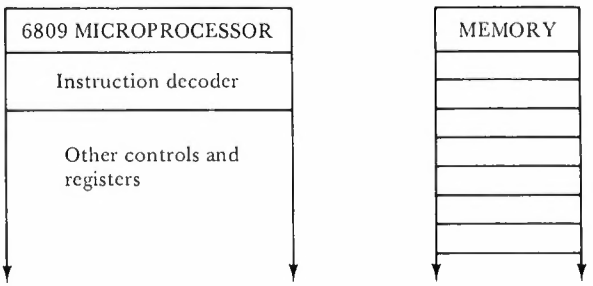
The Central Processing Unit (CPU) used by the TRS-80 Color Computer is named the 6809. It is a member of the 68xx family of microprocessors manufactured by Motorola, Inc. It is called a central processing unit because all instructions and numerical values are routed there for processing. The 6809 microprocessor, and hence the Color Computer, only understands instructions that are coded in blocks of eight binary digits called bytes.



Because the Color Computer can digest words whose size is one byte, all instructions and numerical values must be sent to its central processing unit in this byte size.

The computer is composed of many functional parts that we will introduce as needed to understand the operations taking place. In addition to the CPU, other important parts are the instruction decoder and the memory in which instructions and data are stored.

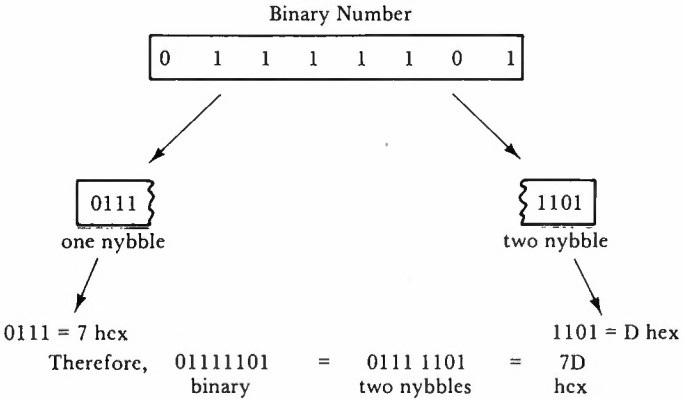
The instruction decoder of the 6809 "reads" the instruction and decodes it. Memory is separate from the microprocessor but is connected to it by address and data lines (usually referred to as address and data busses).



A Bit about Numbers

You can imagine that it would be very tedious to enter many of those long, binary-coded instructions and bytes of data. Because there are only two symbols (0 and 1), the binary representation of numbers is

quite cumbersome. Most computers, including the TRS-80 Color Computer, have the ability to accept a shorthand representation of binary numbers. This shorthand is the hexadecimal number system (often referred to as hex, for short). Four binary digits may be represented by one hex digit. Thus, one 8-bit long instruction can be represented by a 2-digit hex number by breaking the binary value into two parts.



The hexadecimal number system has sixteen symbols (0,1,2,3,4, 5,6,7,8,9,A,B,C,D,E,F). The relationships between decimal, binary, and hex values are shown in the following table.

Table 1-1. Decimal/ Binary/Hex Equivalents

Decimal	*Binary*	*Hex*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

To give the table more meaning, let's look at the binary system. Each place value in the binary system is a power of two. Two is the base of the binary system, and ten is the base of the decimal system. If we look at the place values of the binary numbers 0000 through 1111, we can attach more meaning to them.

Binary Places				*Decimal*
2^3	2^2	2^1	2^0	*Equivalent*
0	0	0	1	0 + 0 + 0 + 1 = 1
0	0	1	0	0 + 0 + 2 + 0 = 2
0	1	0	0	0 + 4 + 0 + 0 = 4
1	0	0	0	8 + 0 + 0 + 0 = 8

Using combinations of these place values, we may obtain any decimal value from 0 through 15 or any hex value from 0 through F.

Examples:
0101 = $2^2 + 2^0 = 4 + 1 = 5$ decimal and also 5 hex
1010 = $2^3 + 2^1 = 8 + 2 = 10$ decimal, which is A hex
1100 = $2^3 + 2^2 = 8 + 4 = 12$ decimal, which is C hex
1101 = $2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$ decimal, which is D hex

Let's now take a closer look at how we may express any 8-bit binary number by two hex digits. We saw earlier that the highest hex digit (F) corresponds to the 4-bit binary value 1111. The next higher binary value is 10000. The 1 is in the 2^4 place, which equals 16. Therefore, we have one 16 and nothing else. This can be expressed by the hex value 10, which means one 16 and no 1s. There is a direct relationship between the upper four bits of an 8-bit binary number and the 16's place digit of a hex number.

Binary Places				*Hex Value*
2^7	2^6	2^5	2^4	16^1
0	0	0	1	1
0	0	1	0	2
0	1	0	0	4
1	0	0	0	8

$2^4 = 16$
2*16 = 32
4*16 = 64
8*16 = 128

Next, look at the binary place values of the complete 16-bit number.

Binary Places								*Decimal*	*Hex*
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	*Equivalent*	*Equivalent*
0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	2	2
0	0	0	0	0	1	0	0	4	4
0	0	0	0	1	0	0	0	8	8
0	0	0	1	0	0	0	0	16	10
0	0	1	0	0	0	0	0	32	20
0	1	0	0	0	0	0	0	64	40
1	0	0	0	0	0	0	0	128	80

Using combinations of all eight bits, you may obtain any decimal value from 0 through 255, or any hex value from 0 through FF. If we break an 8-bit binary number into two 4-bit parts, each part may be represented by one hex digit.

Examples:

binary	01111101
split	0111 1101
hex	7 D
binary	11000011
split	1100 0011
hex	C 3
binary	10101010
split	1010 1010
hex	A A

Instruction manuals for machine language quite often list the instruction codes in both binary and hex forms. The hexadecimal form is commonly used for communicating with the computer when the operator is using a machine language monitor or an assembler. Assemblers usually have options for both decimal and hex entries.

The Machine Language Monitor

A machine language monitor is a great aid to the machine language programmer. The monitor is a program that allows you to enter machine codes and other data into the computer's memory. It also

allows you to examine and change data in memory locations and in registers. Thus, it allows control of data entry, examination, and even the execution of programs.

Although other monitors are available, the CBUG Monitor from The Micro Works* was used in developing Chapters 1 and 2 of this book. The actual commands that are implemented will vary from monitor to monitor, but the methods used are similar.

The CBUG version that we are using is stored on Read Only Memory (ROM) and enclosed in a cartridge that plugs into the Color Computer's cartridge slot. The beginning address of this version of CBUG is $C000. The $ sign preceding a number will be used to indicate a hex value ($C000 = 12*4096, or 49152 decimal). When CBUG is installed and the Color Computer turned on, you see the following:

```
EXTENDED COLOR BASIC 1.0
COPYRIGHT (C) 1980 BY TANDY
UNDER LICENSE FROM MICROSOFT

OK
■
```

you can tell we are using a 16K Color Computer and we are in Extended Color BASIC

To access CBUG, we type:

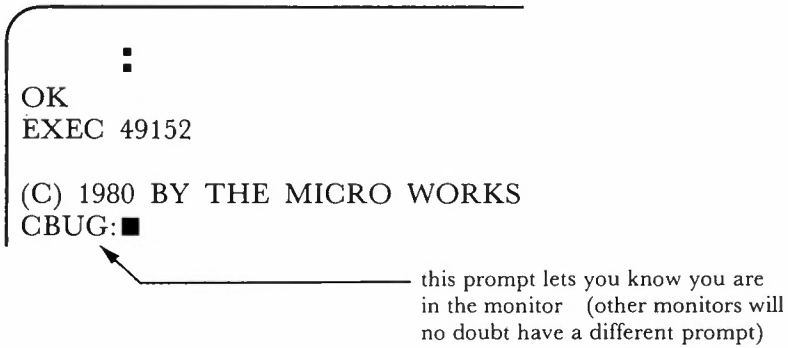
```
EXEC 49152
```

the decimal address of the beginning of CBUG

If you are using a different monitor, the EXEC address may be different.

*The MICRO WORKS, P.O. Box 1110, Del Mar, CA 92014

We see

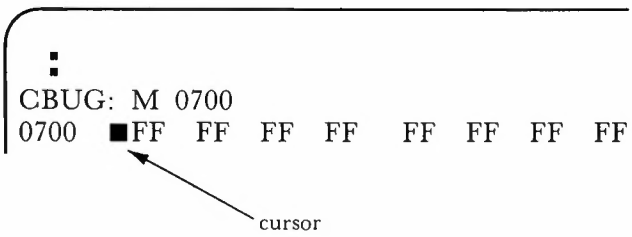


We'll demonstrate the entry and execution of a machine language program that directs a sound to the audio of the TV set. The instructions used and their functions will be explained after you enter and run the program. If you do not have a machine language monitor, you may POKE the instructions and data into memory from BASIC. However, be sure to POKE in the decimal values of the instructions and data into decimal locations.

We are going to use the sound program listed in the CBUG Monitor Owner's Manual. The beginning of the program will be located at memory address $0700.

To examine and change memory values, the CBUG command is: M—typing M 0700 (use all 4 digits) will display a line of eight memory locations (8 bytes) with the cursor in front of the value of the memory location that you type (0700 in this case).

We see



The cursor may be moved up, down, left, or right with the arrow keys (or to the right with the space bar) to display more memory.

> A carriage return (the ENTER key) will exit the memory command and return the computer to the CBUG monitor ready for a new command.

If an inverted (reverse video) number is displayed, you have tried to write to an area where there is no Random Access Memory (RAM).

Here is the program we want to enter.

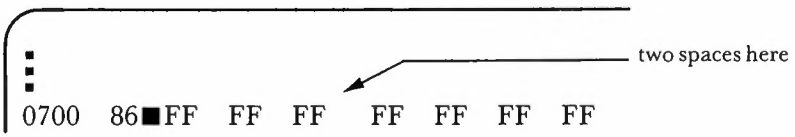
Table 1-2. Program 1 Sounds

Instruction Number	*Memory in Hex*	*Value Entered*
1	0700 0701	86 3F
2	0702 0703 0704	B7 FF 23
3	0705 0706	1F 89
4	0707 0708 0709	F7 FF 20
5	070A 070B 070C	12 12 12
6	070D	5C
7	070E 070F	26 F7
8	0710	4C
9	0711 0712	2A 01
10	0713	4F
11	0714 0715	20 EF

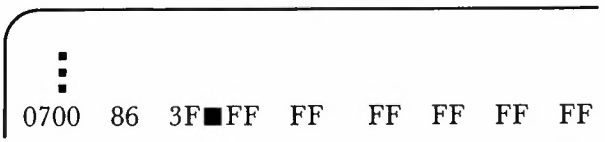
Since we have typed M 0700, the data may now be entered, two digits (one entry) at a time. You *do not* have to press the space bar between entries. The monitor will enter the data and automatically move the cursor in front of the value in the next memory location. If you do press the space bar, the computer will skip over one memory location. In that case, press the left arrow and it will move back. If you make an error, it can be corrected by using the arrow keys to position the cursor to the appropriate memory location and then retyping the correct entry.

This is how the display looks as successive entries are made.

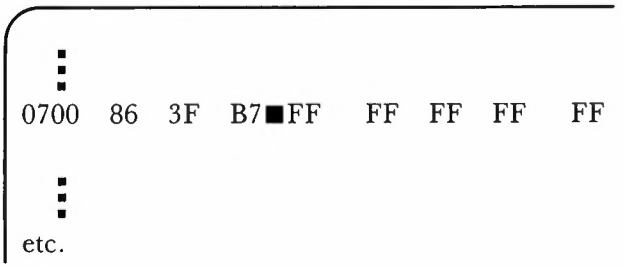
Type 86



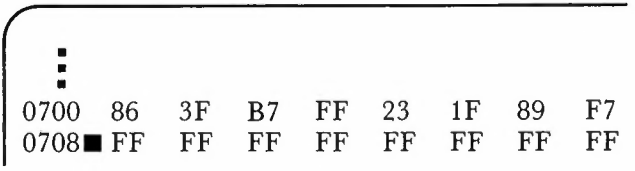
Type 3F



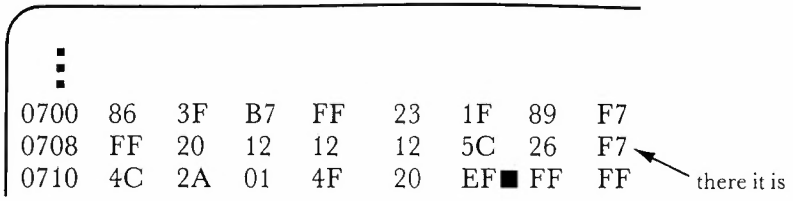
Type B7



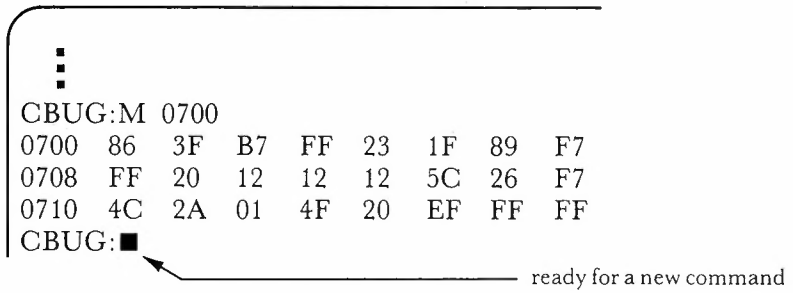
After the eighth entry is made, the next eight bytes of memory are displayed.



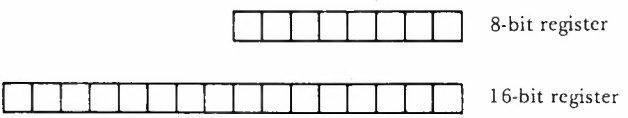
Continue entering data until the complete program is in memory.



Check the data on the display to make sure it is correct. If you spot an error, move the cursor to a position just in front of the error and retype the entry (correctly this time, of course). When everything is correct, exit memory/examine by typing a carriage return (the ENTER key).

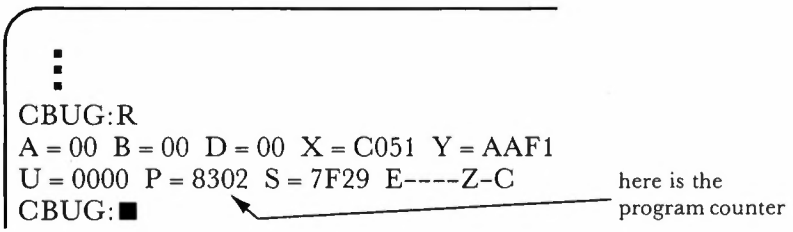


The 6809 has a number of registers that contain vital information. A register can be thought of as a special type of memory location. Some registers hold one byte (eight bits) of information. Others hold two bytes (sixteen bits).



One of these registers is called the program counter. It holds two bytes that keep track of which instruction the computer is working with. As one instruction is being executed, the program counter is "pointing to" the memory location of the next instruction. Since our program starts at location $0700, we want to be sure that the program counter has this value in it when we start the program.

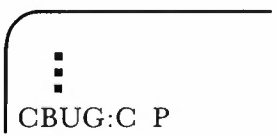
The CBUG command that will display the register contents is: R. Again, you should check the program counter before executing the program.



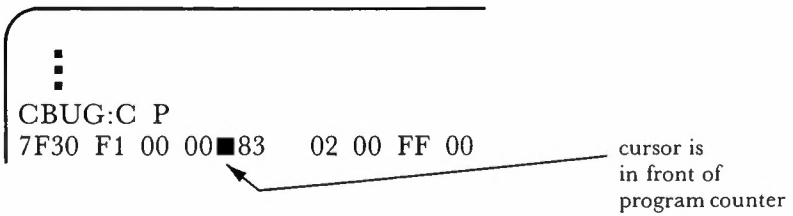
Ignore all the other registers for the time being. We'll discuss each of them as necessary later on. Notice the value of the program counter (8302). This is not the memory location where our program starts; it must be changed to 0700. Notice that control is returned to CBUG after the registers are displayed.

The CBUG command for changing one or more registers is the letter C followed by a single letter that indicates the register to be changed. P is the letter for the program counter.

Type: C P



This transfers control to the memory examine/change function (M) at the address on the stack (to be discussed later) where the specified register is stored.



Type: 0700 then press the ENTER key to leave the memory/examine function.

Now type: R to display the registers again.

```
⋮
CBUG:R
A = 00  B = 00  D = 00  X = C051  Y-AAF1
U = 0000  P = 0700  S = 7F29  E----Z-C
```

there is the correct value in the program counter for our program

Now we are ready to execute the program. The CBUG command to use is: J. J is a jump to machine language subroutine. The J is followed by the starting address of our program.

```
⋮
CBUG:J 0700
```

The program starts immediately, and the noise emitted by the TV sounds like laser guns firing in rapid succession.

How do we stop it? Press the BREAK key? No, it keeps on blasting away. Maybe the ENTER key? No, it keeps on. Well, about the only thing left to do is to press the RESET button on the back of the computer. Ahh! It stopped.

```
OK
■
```

But, what's this? The OK prompt indicates that we're back in BASIC. To get the machine language program back, you must access your monitor again.

1. Type: EXEC 49152 to get CBUG
2. Type: M 0700 and press the ↓ key a couple of times and there it is.

```
OK
EXEC 49152

CBUG:M 0700
0700 86 3F B7 FF  23 1F 89 F7
0708 FF 20 12 12  12 5C 26 F7
0710 4C 2A 01 4F  20 EF FF FF
```

just as before

How Program 1 Works

Now let's discuss the machine language instructions that were used in the program. You may have noticed that the instructions in Table 1-2 were blocked off in groups of 1, 2, and 3. Some instructions require more bytes than others.

Step 1. The first instruction consisted of two bytes with each byte occupying one memory location.

0700 86 ← operation code for the instruction, load accumulator A with the data that follows

0701 3F ← 3F is the data

Accumulator A is an 8-bit register

Remember, we said earlier that some registers hold eight bits and some hold sixteen bits. We will be using the A register frequently in the transfer of data from one place in the computer to another. When the first instruction is executed, the data (3F) is loaded into accumulator A.

0	0	1	1	1	1	1	1
		3			F		

← accumulator A now contains 3F

Step 2. The second instruction consisted of three bytes.

0702 B7 ← the operation code for the instruction, store accumulator A into the memory location that follows

0703 FF
0704 23 } ← memory location $FF23

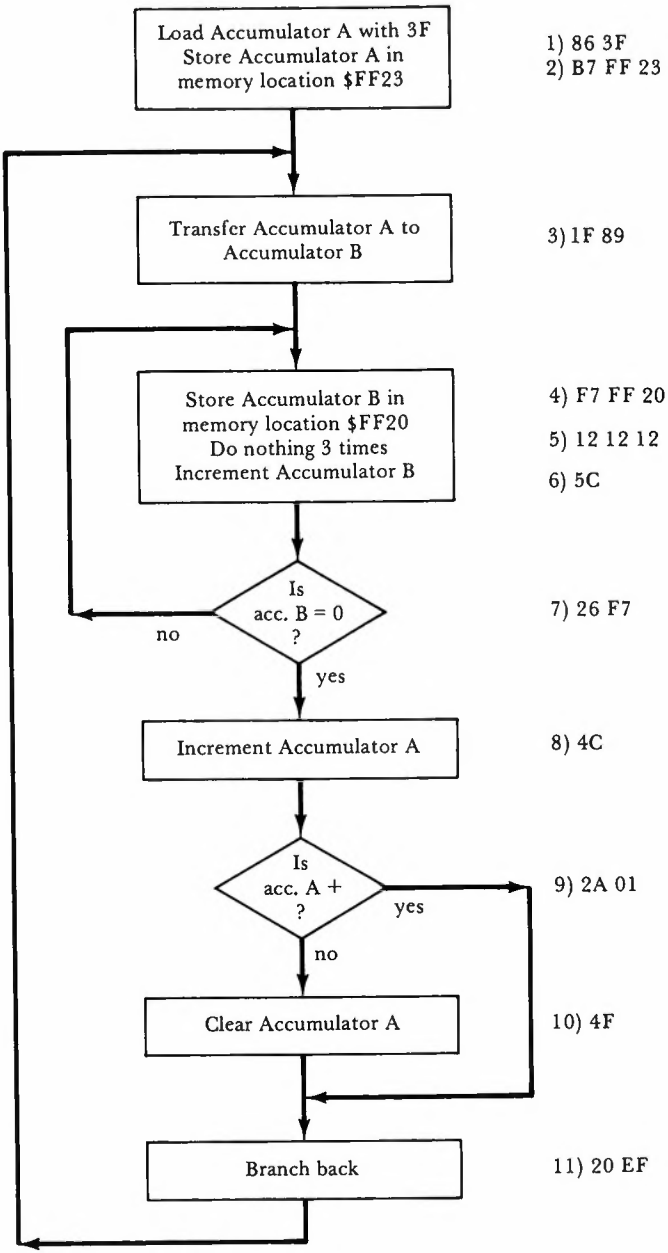


Figure 1-1. Flowchart of Program 1—Sounds

FF23 is associated with the output device used to send the sound to the TV. The data sent to $FF23 "turns on" the audio of the TV.

Step 3. The third instruction has 2 bytes.

```
0705 1F   ← operation code for transfer data from
            one register to another
0706 89   ← 89 indicates the data is copied from
            accumulator A to accumulator B
```

Accumulator B is an 8-bit register

Accumulators A and B may be used together to hold sixteen bits. In that case, it is referred to as double accumulator D. You will see it used in this way later in the book.

Step 4. The fourth instruction has 3 bytes.

```
0707 F7   ← operation code for store accumulator B
            in memory
0708 FF }
0709 20 }   $FF20 is the memory location
```

The memory location $FF20 controls the tone of the sound going to the TV. The data (3F) in accumulator B is the tone loaded.

Step 5. The fifth group is really one instruction (12) used three times. All three are NOP instructions (No Operation). They don't do anything but take up some time.

```
070A 12
070B 12
070C 12
```

Step 6. The sixth instruction 5C uses only one byte.

```
070D 5C   ← op code to increment accumulator B
```

This instruction adds one to the data in accumulator B.

Step 7. A 2-byte instruction is next.

```
070E 26   ←op code for Branch if Not Equal to
            zero (accumulator B this time)
070F F7   ←F7 is the signed number −9. The
            value in the program counter is
            reduced by 9 if the result of
            incrementing B is not zero
```

Program counter is 0710 if B = 0
Program counter changes to 0707 if B is not 0

Step 8. If the branch at instruction 7 is not taken, this 1-byte instruction would be executed next.

```
0710 4C   ←op code to increment accumulator A
```

This instruction will affect step 9.

Step 9. This 2-byte instruction is then executed.

```
0711 2A   ←op code for branch if positive
            (accumulator A from step 8)
0712 01   ←increase the program counter
            by 1 (if A is positive)
```

This instruction (if A is +) will skip over the instruction at step 10. The program counter will be changed from 0713 to 0714. If A is −, the program counter will stay at 0713 for step 10.

Step 10. This is a 1-byte instruction.

```
0713 4F   ←op code for clear accumulator A
```

This instruction sets accumulator A to 0.

Step 11. Last of all is this 2-byte instruction.

```
0714 20   ←op code for branch always
0715 EF   ←negative 17 is added to the
            program counter.
```

The program counter changes to 0705, and the program proceeds from that point. Thus, the program loops back to step 3. The laser sound is repeated over and over until you reset the computer.

We used quite a mixture of instructions in this program. Let's see if we can make some sense out of the mix.

Instructions Used in Program 1

There are several ways that instructions can be classified. If we classify them according to their function, you have seen four types in this program:

1. Data moves
   Load accumulator A—op code 86
   Store accumulator A—op code B7
   Store accumulator B—op code F7
   Transfer A to B—op code 1F 89

Table 1-3. Instructions Used in Program 1

Op Code	*Mnemonic*	*Bytes*	*Address Mode*	*Remarks*
86	LDA	2	Immediate	accumulator A is loaded with data that follows
B7	STA	3	Extended	data in A copied into memory that follows
F7	STB	3	Extended	data in B copied into memory that follows
1F	TFR	2	Inherent	data copied from one register to another
4C	INCA	1	Inherent	add 1 to A
5C	INCB	1	Inherent	add 1 to B
4F	CLA	1	Inherent	zero in A
12	NOP	1	Inherent	no operation
26	BNE	2	Relative	branch if result is not equal to zero
2A	BPL	2	Relative	branch if result is plus (positive)
20	BRA	2	Relative	branch always

2. Register alterations
   Increment accumulator A—op code 4C
   Increment accumulator B—op code 5C
   Clear accumulator A—op code 4F

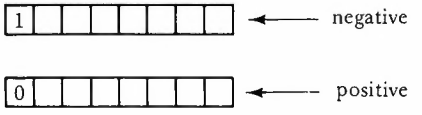
3. Branches
   Branch on result not 0—op code 26
   Branch on result positive—op code 2A
   Branch always—op code 20

4. Do nothing
   No operation—op code 12

Instructions can also be classified by their addressing modes. We will be discussing this later. Here are the addressing modes we have used so far. Also shown are their op codes, mnemonic (abbreviations used in assembly language), number of bytes in the instruction, addressing mode, and a brief description.

Signed Numbers

For some instructions, hexadecimal numbers are interpreted as negative values when they are in the range of 80 through FF and as positive values when they are in the range of 0 through 7F. In other words, if the most significant bit (leftmost bit) of a value is set to 1, the number is negative. If the leftmost bit is 0, the number is positive.



NOTE: For branch instructions, 0-7F are considered positive and 80-FF are considered negative.

You might think of 8-bit signed numbers as locations on a large number wheel rather than the usual number line. Then they would look like Figure 1-2.

If signed numbers are to be represented, the computer must have some way to tell them apart (positive or negative). Consider an 8-bit block of data as being composed of one sign bit and seven data bits.

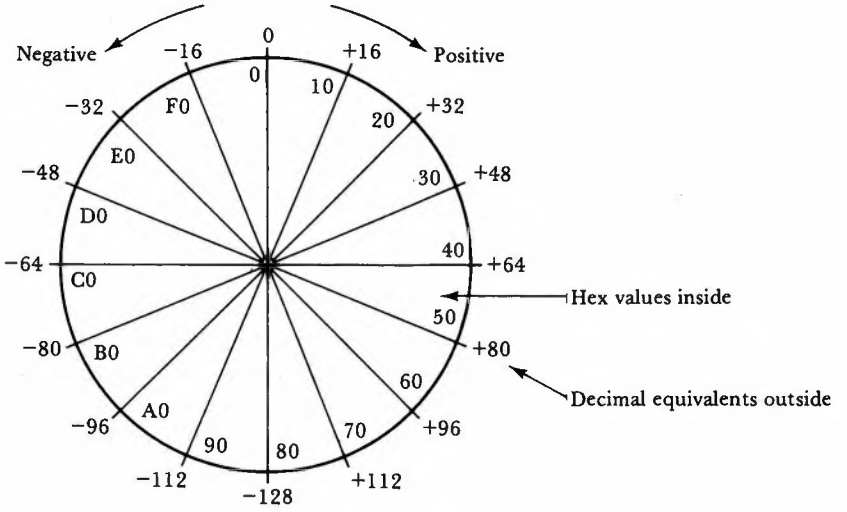
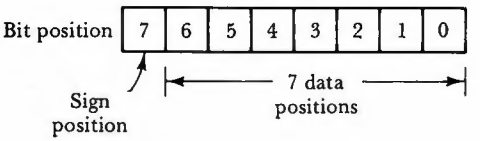


Figure 1-2. Signed Number Wheel



1. If the sign position holds a zero, the data is considered to be a positive number.

Examples

```
01111111 = +127 (64+32+16+8+4+2+1)
01111110 = +126 (64+32+16+8+4+2   )
01111101 = +125 (64+32+16+8+4   +1)

00000011 = +3   (               +2+1)
00000010 = +2   (               +2  )
00000001 = +1   (                 +1)
00000000 = +0   (                   )
```

zero is considered positive by branch instructions

2. If the sign position holds a one (1), the data is considered to be a negative number.

Examples

10000000 = −128
10000001 = −127
10000010 = −126

11111101 = −3
11111110 = −2
11111111 = −1

For our purposes, we must realize that certain branch instructions test the result of numbers to see whether they are negative or positive.

The values used as the second byte in relative branch instructions are shown in the following two tables.

Table 1-4. Branch Bytes for Forward Branches

Steps dec.	*Branch hex.*	*Steps dec.*	*Branch hex*	*Steps dec.*	*Branch hex*	*Steps dec.*	*Branch hex*
1	01	33	21	65	41	97	61
2	02	34	22	66	42	98	62
3	03	35	23	67	43	99	63
4	04	36	24	68	44	100	64
5	05	37	25	69	45	101	65
6	06	38	26	70	46	102	66
7	07	39	27	71	47	103	67
8	08	40	28	72	48	104	68
9	09	41	29	73	49	105	69
10	0A	42	2A	74	4A	106	6A
11	0B	43	2B	75	4B	107	6B
12	0C	44	2C	76	4C	108	6C
13	0D	45	2D	77	4D	109	6D
14	0E	46	2E	78	4E	110	6E
15	0F	47	2F	79	4F	111	6F
16	10	48	30	80	50	112	70
17	11	49	31	81	51	113	71
18	12	50	32	82	52	114	72
19	13	51	33	83	53	115	73
20	14	52	34	84	54	116	74
21	15	53	35	85	55	117	75
22	16	54	36	86	56	118	76
23	17	55	37	87	57	119	77
24	18	56	38	88	58	120	78
25	19	57	39	89	59	121	79

Steps dec.	*Branch hex.*	*Steps dec.*	*Branch hex*	*Steps dec.*	*Branch hex*	*Steps dec.*	*Branch hex*
26	1A	58	3A	90	5A	122	7A
27	1B	59	3B	91	5B	123	7B
28	1C	60	3C	92	5C	124	7C
29	1D	61	3D	93	5D	125	7D
30	1E	62	3E	94	5E	126	7E
31	1F	63	3F	95	5F	127	7F
32	20	64	40	96	60		

Table 1-5. Branch Bytes for Backward Branches

Steps dec.	*Branch hex*	*Steps dec.*	*Branch hex*	*Steps dec.*	*Branch hex*	*Steps dec.*	*Branch hex*
−1	FF	−33	DF	−65	BF	−97	9F
−2	FE	−34	DE	−66	BE	−98	9E
−3	FD	−35	DD	−67	BD	−99	9D
−4	FC	−36	DC	−68	BC	−100	9C
−5	FB	−37	DB	−69	BB	−101	9B
−6	FA	−38	DA	−70	BA	−102	9A
−7	F9	−39	D9	−71	B9	−103	99
−8	F8	−40	D8	−72	B8	−104	98
−9	F7	−41	D7	−73	B7	−105	97
−10	F6	−42	D6	−74	B6	−106	96
−11	F5	−43	D5	−75	B5	−107	95
−12	F4	−44	D4	−76	B4	−108	94
−13	F3	−45	D3	−77	B3	−109	93
−14	F2	−46	D2	−78	B2	−110	92
−15	F1	−47	D1	−79	B1	−111	91
−16	F0	−48	D0	−80	B0	−112	90
−17	EF	−49	CF	−81	AF	−113	8F
−18	EE	−50	CE	−82	AE	−114	8E
−19	ED	−51	CD	−83	AD	−115	8D
−20	EC	−52	CC	−84	AC	−116	8C
−21	EB	−53	CB	−85	AB	−117	8B
−22	EA	−54	CA	−86	AA	−118	8A
−23	E9	−55	C9	−87	A9	−119	89
−24	E8	−56	C8	−88	A8	−120	88
−25	E7	−57	C7	−89	A7	−121	87
−26	E6	−58	C6	−90	A6	−122	86
−27	E5	−59	C5	−91	A5	−123	85
−28	E4	−60	C4	−92	A4	−124	84
−29	E3	−61	C3	−93	A3	−125	83
−30	E2	−62	C2	−94	A2	−126	82
−31	E1	−63	C1	−95	A1	−127	81
−32	E0	−64	C0	−96	A0	−128	80

Summary

- The computer can be thought of as a gigantic array of electronic switches that are either open or closed.

- Binary numbers are used to communicate switch patterns to the computer. A 0 represents an open switch and a 1 represents a closed switch. A group of eight of these symbols communicates data to or from the computer.

1 0 0 1 1 1 0 0

- A binary number can be represented by two hex digits.

1 0 0 1 1 1 0 0

9 C

Hex symbols are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
A, B, C, D, E, and F.

- A machine language monitor allows you to enter machine codes and data into the computer's memory and registers. It also allows you to examine and change such data and to execute programs.

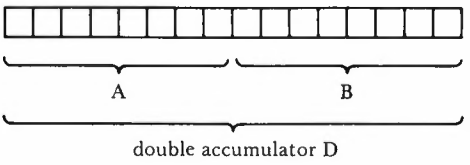
- The Color Computer uses the 6809 microprocessor. It has both 8-bit and 16-bit registers.

  The program counter is a 16-bit register that keeps track of the computer's position within a program.

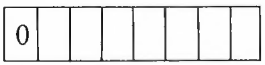
  Accumulator A is an 8-bit register used for temporary storage during data transfer. Arithmetic and logic operations also take place there.

  Accumulator B is an 8-bit register used in the same way as accumulator A.

  Accumulator D is a 16-bit register resulting from the combination of A and B.



- Branch instructions use signed numbers (positive and negative). A number is considered positive if its most significant bit (left-most) is a 0.



←0 through 7F in hex notation

A number is considered negative if its most significant bit is a 1.

←80 through FF in hex notation

Chapter Test

1. The binary digit used for switch closed is _____.
   The binary digit used for switch open is _____.
2. One byte is a block of how many bits? _____
3. What microprocessor does the Color Computer use? _____
4. Convert the following binary numbers to their hexadecimal equivalents.

   01110110 = _____ _____
   10011100 = _____ _____
   11101111 = _____ _____

5. Convert the following hexadecimal numbers to their binary equivalents: D3, 8C, 5A.
6. What size is accumulator A? _____ bits
7. What size is accumulator B? _____ bits
8. Describe double accumulator D. _____
   _____
9. What is the purpose of register P as used in the CBUG monitor?
   _____
   _____
10. If the second byte of a branch always (BRA) instruction is E3, in which direction will the branch be made, backward or forward?
    _____

Answers to Odd-numbered Exercises in Chaper Test

1. switched closed is 1
   switch open is 0
3. 6809 or 6809E
5. D3 = 11010011
   8C = 10001100
   5A = 01011010
7. 8 bits
9. P is the program counter. It points to the instruction that will be next executed. Thus it keeps track of where the computer is within the program.

Chapter 2

Sound

After running the sound program in Chapter 1, you would no doubt like to find out how to vary the tone and duration in order to create sounds of your own. Using Program 1 as a base, you can make a few changes to do this and we will show you how.

We'll also introduce the symbolism used by an assembler along with the machine language operation codes. This will make the instructions more understandable and also prepare you for the assembly language chapters that follow this one. To abbreviate the listing of this program, we'll give only the starting memory location for each instruction and put the complete instruction on one line.

Program 2—Making Your Own Sounds

Table 2-1. Program 2—Making Your Own Sounds

Memory	*Machine Codes*	*Assembler Symbols*	*Remarks*
0700	86 3F	LDA #$3F	load sound byte
0702	B7 FF 23	STA $FF23	turn on sound
0705	8E 00 FF	LDX #$FF	load duration
0708	C6 5F	LDB #$5F	tone value 5F
070A	F7 FF 20	STB $FF20	store tone
070D	12	NOP	time delay
070E	12	NOP	
070F	12	NOP	
0710	5C	INC B	increment B
0711	26 F7	BNE	branch (−9)
0713	30 1F	DEX	decrement X
0715	26 F1	BNE	branch (−15)
0717	39	RTS	return to CBUG

Note the symbols used in front of data in the Assembler Symbols column.

which precedes the LDA, LDX, and LDB values to the assembler indicates that the data to be loaded will be found immediately after the instruction.

$ indicates to the assembler that the data is in hexadecimal format.

Also note that each assembler instruction is an abbreviation of its function (see Remarks column):

LDA LoaD accumulator A
LDX LoaD X register
BNE Branch if result Not Equal to zero, etc.

Using CBUG, or some other monitor, load the program into your computer. With CBUG, the display looks as follows after the program has been entered.

```
CBUG:M 0700
0700    86  3F  B7  FF  23  8E  00  FF
0708    C6  5F  F7  FF  20  12  12  12
0710    5C  26  F7  30  1F  26  F1  39
0718   ■FF  FF  FF  FF  FF  FF  FF
```

cursor

Leave the memory mode (by pressing ENTER) and examine the registers to make sure the program counter is set to 0700.

```
0718  FF  FF  FF  FF  FF  FF  FF  FF
CBUG:R
A = 00  B = 00  D = 00  X = C051  Y = AAF1
U = 0000  P = 8302  S = 7F29  E----Z-C
CBUG:■
```

this must be changed

Type: C P

```
CBUG:C P
7F30  F1  00  00■83  02  00  FF  00
```

Type: 0700 and press ENTER

```
   :
7F30  F1  00  00  07  00  00  FF  00
CBUG:■
```

Type: J 0700 to execute the program. Your note sounds, then a return is made to CBUG.

```
   :
CBUG:J 0700
CBUG:■
```

Now, what if you want to change the sound? The tone of the note is controlled by the value following the instruction, Load Accumulator B (the value is 5F in the original program). The duration is controlled by the value loaded into the X register.

To change the note:
change the value at memory location 0709

To change the duration:
change the value at memory locations 0705 and 0706

As an example, we will change the value in 0709 to 30 and the value in 0706 to 80, leaving 0705 at 0.

```
   :
CBUG:J 0700
CBUG:M 0706
0700  86  3F  B7  FF  23  8E  00■FF
                              ↖— change this to 80
```

Type: 80

```
   :
CBUG: J 0700
CBUG:M 0706
0700    86   3F   B7   FF   23   8E   00   80↖
0708   ■C6   5F   F7   FF   20   12   12   12 ↗
                                        now changed
          ↖————— this one to be changed to 30
```

Type: ↵ or space bar

```
   :
   :
0708  C6■5F  F7  FF  20  12  12  12
```

Type: 30 and press ENTER (to leave memory examine/change)

```
   :
   :
0708  C6  30  F7  FF  20  12  12  12
CBUG:■
```

Now execute the program again. A lower tone is heard for a shorter time.

- Location 0709 may be varied from 0-FF.
- The combination of 0705 and 0706 holds a 2-byte count for the duration. This double size value may range from 0000 through FFFF.

Experiment by changing these values several times before going on to the discussion of the instructions used in Program 2.

New Instructions in Program 2

The following new instructions were used in Program 2.

Table 2-2. New Instructions in Program 2

Op Code	*Mnemonic*	*Bytes*	*Address Mode*	*Remarks*
C6	LDB	2	Immediate	accumulator B is loaded with data that follows
8E	LDX	3	Immediate	register X is loaded with following data
30 1F	DEX	2	Inherent	the X register is decremented by 1
39	RTS	1	Inherent	return is made to CBUG

NOTE: The instruction with Op code 30 1F really has a mnemonic of: LEAX −1,X. This will be discussed in Chapter 4. We used DEX because more programmers are familiar with it.

How Program 2 Works

The first two instructions put a value into memory location $FF23 to turn on the sound. The X register is then loaded with the count that determines how many times the sound loop will be executed. This controls the duration of the notes.

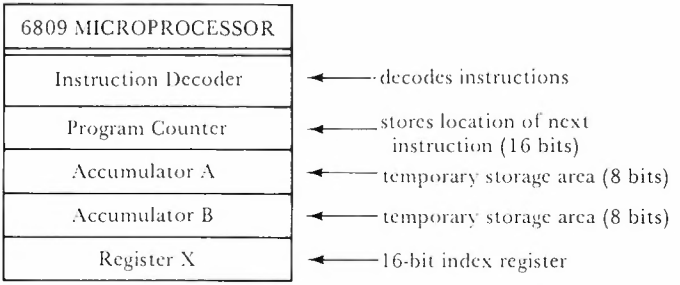
Accumulator B is loaded with a value (0-FF) that determines the tone. This value is stored in memory location $FF20, the sound generator. Three NOPs (No OPerations) follow. NOPs may be used as time delays or to save space for instructions or data that may be inserted at a later time. Accumulator B is incremented. If B is not zero, a branch is made backward to store the new value in $FF20. This continues until the value in accumulator B is zero.

The value in the X register is then decreased. If the new value in X is not zero, the computer branches back to load B with the original value and proceeds as before. Thus a nested loop is formed.

When the value in X reaches zero, the sound stops. The computer returns to CBUG. At this time, you could enter the memory examine/change mode (M) and change values for the duration and/or the tone.

A flowchart of the program is shown in Figure 2-1.

Let's take a look at some of the registers that you have used so far.



The 16-bit program counter acts as a program address pointer to assure that instructions are executed in the desired order. Instructions of a program are stored in consecutive locations in memory. They are made up of machine language operation codes and/or address bytes and numbers to be operated on. To control the desired sequence of operations of a program, the program counter is used as a pointer to designate the position in memory where the microprocessor will obtain each successive instruction. The program counter is incremented, after each instruction is "fetched," to point at the next instruction to be performed.

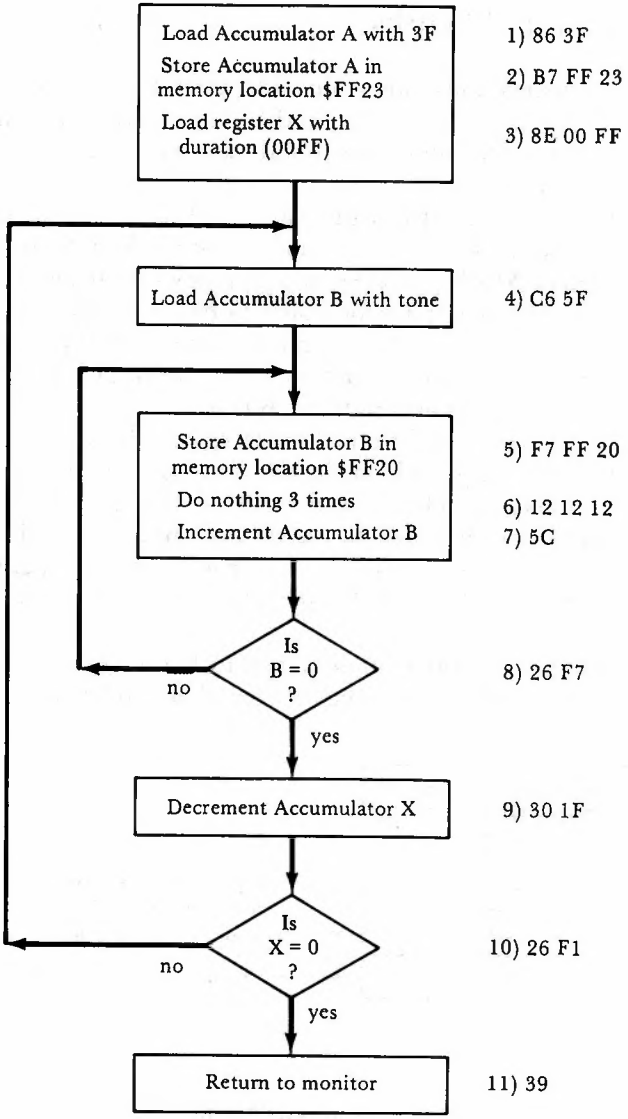
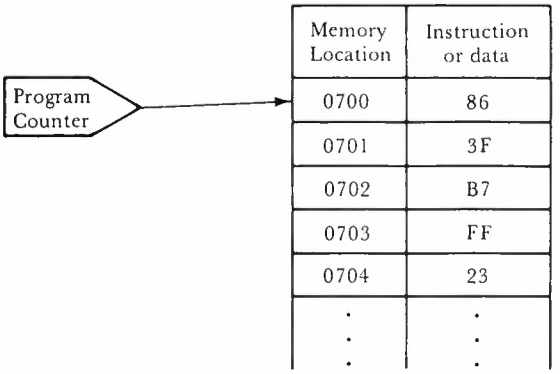


Figure 2-1. Flowchart of Program 2



The accumulators (A and B) are registers (storage places similar to memory) in which data is placed. They are used as temporary storage areas when moving data from one place to another. Arithmetic and logical operations on data also take place in the accumulators. Thus, they are used frequently, and many 6809 instructions involve them. They "accumulate" the results of successive operations on data that are requested by the instructions. Each accumulator holds eight bits of data.

The X register is also used for temporary data storage. It is sixteen bits wide and has the ability to be incremented and decremented by programmed instructions. When it is incremented, the value in the register is *increased* by one. When decremented, the value in the register is *decreased* by one. Therefore, the X register can be used as a counter (or a pointer) to store data into successive memory locations or to load data into successive memory locations. This is called indexing (hence it is often referred to as an index register). It can also be used as a counter (as in Program 2) to determine conditions for ending a series of repeated operations (a loop).

In Program 2, the X register was used to control the duration of the note. A value was loaded into X at memory locations 0705–0707. This value was decremented at 0713–0714 each time through the sound loop. When the value in X reached zero, the sound stopped. Thus X was used as a counter, going backwards from 00FF (or some other value that you inserted at 0706–0707) down to zero.

Execute Program 2 again. When you have finished, examine the registers by typing R after the CBUG prompt.

```
   :
CBUG:J 0700
CBUG:R
A = 00  B = 00  D = 00  X = C051  Y = AAF1
U = 0000  P = 0700  S = 7F29  E----Z-C
CBUG:■
```

We have discussed four of the registers displayed:

A is accumulator A = 00

B is accumulator B = 00

X is register X = C051

P is the program counter = 0700

Do the values in these registers give you any clues about the program? Not much. The problem is that CBUG resets some of the registers when a Return from Subroutine (RTS) instruction sends the computer back to the monitor at the end of the program.

To overcome this fact, let's put a Software Interrupt Instruction (SWI) in front of the return from subroutine instruction (memory 0717). The Op code for SWI is 3F.

Type: M 0717

```
   :
CBUG:M 0717
0710  5C  26  F7  30  1F  26  F1■39
                                  ← 3F to be inserted
```

Type: 3F then 39 and press ENTER

```
   :
CBUG:M 0717
0710  5C  26  F7  30  1F  26  F1  3F   ← software interrupt
0718  39  FF  FF  FF  FF  FF  FF  FF
CBUG:■
   ↑ return from subroutine now here
```

Now, type: ! ! is the CBUG command to take over the software interrupt

```
  :
0718  39  FF  FF  FF  FF  FF  FF  FF
CBUG:!
CBUG:■
```

As quoted from the CBUG Monitor Owner's Manual, "Until this command (!) is executed, it is undetermined what will happen when a SWI instruction (3F) is encountered by the 6809. This instruction sets the vector so that control will return to the monitor. Machine language debugging may then be accomplished by inserting SWIs into the program, at which point the monitor will be entered and the register contents dumped."

Now that you have inserted the software interrupt, run the program again.

```
  :
CBUG:!
CBUG:J 0700
A = 3F  B = 00  D = 00  X = 0000  Y = AAF1
U = 0000  P = 0718  S = 7F19  E----Z-C
CBUG:■
```

Notice the four registers now show the following:

A = 3F ← value originally loaded

B = 00 ← B has been incremented to zero

X = 0000 ← X has been decremented to zero

P = 0718 ← The program was interrupted by the SWI instruction at step 0717. Hence, the program counter is pointing to the next instruction (return from subroutine) at memory location 0718

Now, type: G

```
:
CBUG:G
CBUG:■
```

← the G command resumes the program at the instruction following the interrupt

Type: R to see the registers after the return from the subroutine

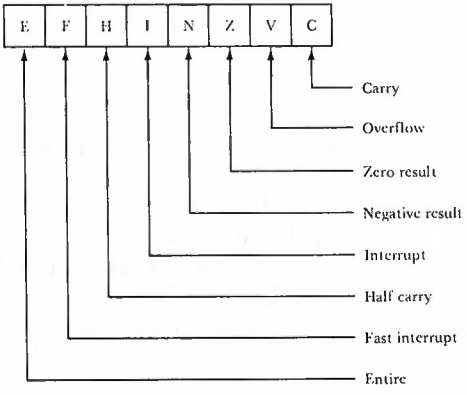
```
:
CBUG:G
CBUG:R
A = 00  B = 00  D = 00  X = C051  Y-AAF1
U = 0000  P = 0700  S = 7F29  E  ----Z-C
CBUG:■
```

A and B have been reset to zero; X was used by the monitor in resetting; P is set back to the beginning of the program.

Now, what about those funny symbols at the end of the register list (E----Z-C)? Read on!

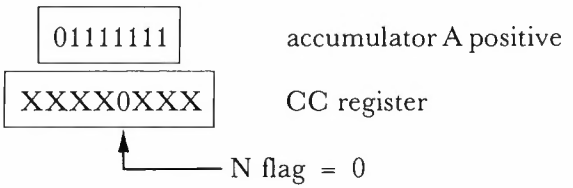
Condition Codes

The 6809 microprocessor has a special register called the Condition Codes register (CC register for short) that keeps track of such things as overflow, carry, negative result, zero result, etc. Each bit in this 8-bit register is assigned a special condition as shown.

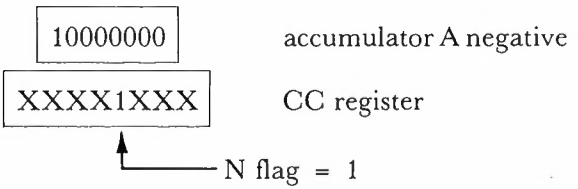


Individual bits of the Condition Codes register are used to keep track of specific effects that instructions have on the "status" of the computer. The presence or absence of an effect is shown by whether a particular bit has been set to one or reset to zero. These individual bits are also called flags. The conditions (or effects) are discussed in more detail as they are needed in understanding the instructions. For the present, we'll deal mainly with the Zero (Z) and the Negative (N) flags.

You have used two branch instructions whose action depends on the condition of the flags. Program 1 used the Branch if Plus (BPL; op code 2A). It followed the instruction, increment A. If the result in the A register is positive or zero, the N flag of the Condition Codes register would be zero.

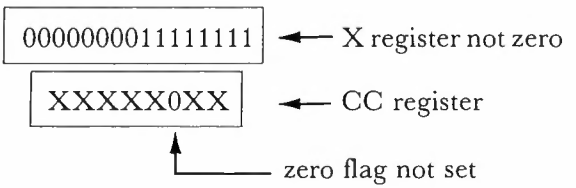


If the result of incrementing A is a negative number (80–FF), the N flag would be set to one.

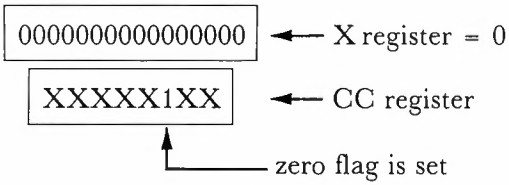


The branch would *not* be taken in the latter case because the value in the A register is negative (*not* positive).

In Program 2 (also in Program 1), you used the Branch if Not Equal to zero (BNE) instruction. The instruction preceding it in Program 2 was decrement the X register. If the result in the X register is not zero, the zero flag will not be set.



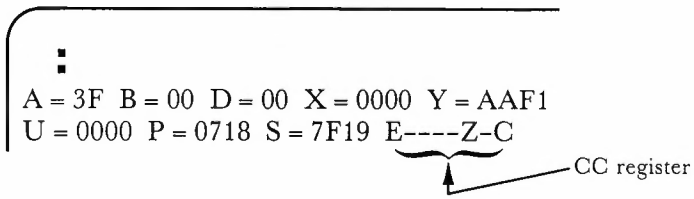
If the result in the X register is zero, the zero flag will be set to one.



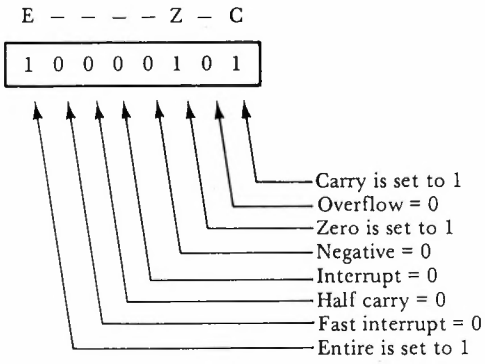
In the latter case, the branch would not be taken.

As you can see from the previous examples, the execution of a branch depends upon certain flags in the Condition Codes register. Some of the flags are changed when certain instructions are executed. Therefore, to branch or not to branch is determined by some instruction that was executed previous to the branch instruction.

Getting back to what you saw in the Condition Code register:



When you use the CBUG Monitor: note that if the letter of the condition flag is displayed, that flag is set to one; if a dash is displayed, the flag is zero. Thus



Using a Sound Table

If you want to play several notes in a row in a sound program, you can store a table of note values in memory. A different note can be accessed from the table each time the note playing portion of the pro-

gram is repeated. To do this, we can use another counting register. Fortunately, the 6809 CPU has a Y register that is very similar to the X register. In Program 3, we'll use the Y register to keep track of the notes being played.

Table 2-3. Program 3—Using a Sound Table

Memory	*Machine Codes*	*Assembler Codes*
0700	86 3F	LDA #$3F
0702	B7 FF 23	STA $FF23
0705	10 8E 07 50	LDY #$0750
0709	8E 00 80	LDX #$080
070C	E6 A0	LDB ,Y +
070E	C1 00	CMPB #00
0710	27 13	BEQ $13
0712	1F 98	TFR B,A
0714	F7 FF 20	STB #$FF20
0717	12	NOP
0718	12	NOP
0719	12	NOP
071A	5C	INC B
071B	26 F7	BNE $F7
071D	1F 89	TFR A,B
071F	30 1F	DEX
0721	26 F1	BNE $F1
0723	20 E4	BRA $E4
0725	39	RTS

Using CBUG, or some other monitor, load the program.

```
CBUG:M 0700
0700  86  3F  B7  FF  23  10  8E  07
0708  50  8E  00  80  E6  A0  C1  00
0710  27  13  1F  98  F7  FF  20  12
0718  12  12  5C  26  F7  1F  89  30
0720  1F  26  F1  20  E4  39■FF  FF
```

Press the down arrow key (↓) to access memory location 0750 and input data for notes.

```
:
0750  37  40  49  52  5B  64  6D  76   <-- 8 notes
0758  00  FF  FF  FF  FF  FF  FF  FF
      ^
      |
      00 is the end of the data table
```

Now, make sure the program counter is set to 0700.

```
CBUG:R
A = 00  B = 00  D = 00  X = C051  Y = AAF1
U = 0000  P = 8302  S = 7F29  = E----Z-C
CBUG:C P
```

type C P if you need to change the program counter—then change it to 0700

Then run the program. You should hear something that resembles a scale of notes. We don't claim to be musicians, so feel free to tinker with the data table to change the notes. We merely spaced the note values 9 apart.

Explanation of Program 3

The program breaks up into eight logical parts.

Part 1. 0700-0704 Sound is turned on by loading $3F into memory location $FF23, as in Programs 1 and 2.

Part 2. 0705-070B Register Y is loaded with the beginning address of the note table ($0750). Note that this is a 4-byte instruction. The duration is loaded into register X.

Part 3. 070C-0713 The B register is loaded with data from the memory location whose address is currently in the Y register. Then the Y register is increased by one. The data in B is compared to zero (070E-070F). If the difference is zero, a branch is made to the end of the program (all notes have been played) at part 7. If the difference is not zero, the value in B is transferred to A for future use and the computer goes on to step 4.

Part 4. 0714-071C This block plays the note as in Program 2.

Part 5. 071D–0722 The note value saved in A (part 3) is transferred back to B. The X register is decremented. If X is not zero, the note value is stuffed into $FF20 in part 4. If X is zero, the computer goes on to step 6.

Part 6. 0723–0724 A branch is made back to part 3 to get a new note.

Part 7. 0725 This return from subroutine instruction is reached from part 3 when all eight notes have been played. It returns to the monitor.

Part 8. 0750–0758 These are the note values. Be sure to load them before running the program.

New Instructions in Program 3

Table 2-4 shows the instructions appearing in this program that have not been used previously.

Table 2-4. New Instructions in Program 3

Op Code	*Mnemonic*	*Bytes*	*Address*	*Remarks*
10 8E	LDY	4	Immediate	16-bit register Y is loaded with 2 bytes that follow
E6 A0	LDB ,Y +	2	Indexed	B register is loaded from address held in Y: Y increased by 1
C1	CMPB	2	Immediate	B register compared to byte following op code
27	BEQ	2	Relative	Branch made if zero flag is set to 1; 2nd byte tells how far and what direction to go
1F 98	TFR B,A	2	Inherent	data copied from B to A

Suggestions for Playing with Program 3

The program may be altered in several ways for experimenting with sounds. Here are a few suggestions.

1. Change the values in the note table to produce a scale that is pleasing to your ear.

> Note: All musicians—send suggestions to:
> THE DYMAX GAZETTE
> P O Box 310
> Menlo Park, CA 94025

2. Extend the table to play any number of notes. A zero at the end of your values will tell the computer that you are finished.

Examples:

```
:
CBUG:M 0750
0750  37  40  49  52  5B  64  6D  76   <-- add more notes
0758  7F  88  91  9A  A3  AC  B5  00
```

or

```
:
CBUG:M 0750
0750  37  40  49  52  5B  64  6D  76   <-- play scale both ways
0758  6D  64  5B  52  49  40  37  00
```

3. Change the duration so that shorter notes are played. Provide plenty of notes in the table.

```
:
CBUG:M 0708
0708  50  8E  00  20  E6  A0  C1  00   <-- try 20 (or some other number less than 80)
CBUG:M 0750
0750  37  40  49  52  5B  64  6D  76   }
0758  7F  88  91  9A  A3  AC  B5  BE   } <-- lots of notes
0760  B5  AC  A3  9A  91  88  7F  76   }
0768  6D  64  5B  52  49  40  37  00   }
```

4. Change the program so that when a zero is encountered in the data table, the original table will be used again. Caution: You will have to press the RESET button to stop the program.
5. Use your imagination and experiment with your own alterations.

Saving Machine Language Programs on Tape

After you've perfected a machine language program, you can save it on tape and then load it back into the computer at a later time. The CBUG command to save such tapes is S. This is followed, in order, by the starting memory location of the program, the ending location of the program, the entry point of the program (quite often the same as the starting location), and the name of the program.

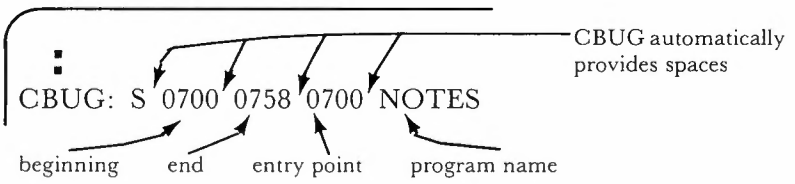
As an example, we'll use Program 3 in its original form with the sound table. Examining the memory for this program, we have

```
  :
0700  86  3F  B7  FF  23  10  8E  07
0708  50  8E  00  80  E6  A0  C1  00
0710  27  13  1F  98  F7  FF  20  12
0718  12  12  5C  26  F7  1F  89  30
0720  1F  26  F1  20  E4  39  FF  FF
```

We also have to put in the data table

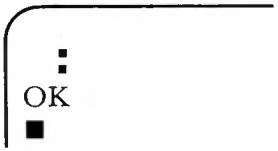
```
  :
0750  37  40  49  52  5B  64  6D  76
0758  00
```

After pressing the RECORD and PLAY buttons on the cassette recorder, we type the save command:



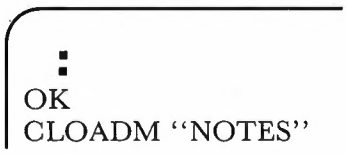
When you press the ENTER key, the recording is made. To be safe, the program should be saved at least twice.

The program is loaded back into the computer from BASIC. As a test, we turned off the computer to erase the program in memory. Then we turned it back on.

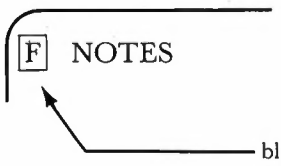


Rewind the tape to the point where the recording was made. Press the PLAY button on the recorder.

Type: CLOADM "NOTES"



Press ENTER



When the program has loaded, type: EXEC and press ENTER. The notes are played.

CBUG does not have to be present for the recorded program to load and execute. Each time the program ends, a return is made to BASIC. To execute the program again, type EXEC and press ENTER.

Summary

- Assembler symbols and instructions used are as follows:
  # data follows immediately
  $ hexadecimal value follows
  BEQ Branch if result EQuals zero
  BNE Branch if result is Not Equal to zero
  BRA BRanch Always
  CMPB CoMPare data in accumulator B
  DEX DEcrement the X register

INCB INCrement accumulator B
LDA LoaD accumulator A
LDB LoaD accumulator B
LDX LoaD register X
LDY LoaD register Y
NOP No OPeration
RTS ReTurn from Subroutine
STA STore accumulator A
STB STore accumulator B
SWI SoftWare Interrupt
TFR A, B TransFeR A to B
TFR B,A TransFeR B to A

- Registers used:
  The 16-bit program counteracts as a program address pointer to assure that instructions are executed in the desired order.
  Accumulators A and B are 8-bit registers used as temporary storage for data.
  The X register is a 16-bit register used as a counter to time the sound duration in Programs 2 and 3.
  The Y register is a 16-bit register used in Program 3 as a counter to index the values used in a data table.
- Condition code register bits:

E	F	H	I	N	Z	V	C

C carry
V overflow
Z zero result
N negative result
I interrupt disable
H half carry
F fast interrupt disable
E entire registers

Chapter Test

1. Describe the function of this assembler notation: LDA #$3F

_____

_____

2. Suppose accumulator A holds the value $3F, and the instruction implemented by the assembler notation STA $FF23 is executed. Show the bits as they would be stored.

3. What does the symbol # mean to the assembler?

   _____

   _____

4. What does the symbol $ mean to the assembler?

   _____

   _____

5. What is the range of values that may be stored in a memory location in hex notation? _____ to _____
6. What is the range of values that may be stored in the X or Y register? _____ to _____
7. What register is used to control the desired sequence of instructions of a program? _____
8. Name two index registers used in the 6809 microprocessor.

   _____ _____

9. What condition codes are used by the following instructions?
   BEQ _____
   BNE _____
   BPL _____
10. Describe the function of the assembler instruction: LDB ,Y +

    _____

    _____

Answers to Odd-Numbered Exercises in Chapter Test

1. Accumulator A would be loaded with the immediate (#) hex value ($) 3F.
3. The value immediately following is to be used as the instruction requests.
5. 00 to FF
7. P, the program counter
9. BEQ the zero flag
   BNE the zero flag
   BPL the negative flag

Chapter 3

Edit, Assemble, and ABUG

In the first two chapters, we demonstrated a machine language monitor to enter, modify, and execute machine language programs. We included mnemonic codes, as well, to prepare you for the use of an assembler. All the programs in the first two chapters were hand-assembled.

Hand-assembly is an uninteresting and tedious task that is very prone to small, but sometimes disastrous, errors. The length of instructions varies, and branch destinations must be calculated. Some instructions require data as operands while others require memory addresses or registers. It is easy to pick wrong op codes or addresses. It is also easy to transpose or mistype digits, etc. It would be much easier for us to assign the job of assembling a program to the computer.

One of the most powerful tools for machine language programming is an assembler. It can easily take care of assembling programs for us if we write the programs using assembly language instructions. The balance of this book is devoted to learning how to use an assembler and the assembly language form for the many addressing modes used.

We have chosen the SDS80C (Software Development System) from The Micro Works. One of the reasons for this choice is its convenience and ease of use. It is packaged in a cartridge that plugs into the cartridge slot of the TRS-80 Color Computer.

The Software Development System contains three separate programs, and it is simple to move back and forth from one program to another. The programs are as follows:

1. The Editor
2. The Assembler
3. The ABUG Monitor

The Editor Program is used to write and edit your assembly language programs. Assembly language is a shorthand that uses English-like abbreviations to represent instructions to the computer. It also uses numbers in decimal or hexadecimal form to provide data for programs.

The Assembler Program translates the abbreviations provided by the Editor into machine language codes and data that the computer can understand. It also takes care of assigning the instructions and data to their proper memory locations.

The ABUG Monitor is used to execute the machine language program that the Assembler produced. In addition, you can examine and modify data in memory or in registers.

You can see that the three programs are used in a logical order. If the program executed by ABUG is faulty, you may return to the Editor Program for changes. The process can be repeated until satisfactory results are obtained. Here is an oversimplified diagram of the order in which the three programs are used.

The Software Development System offers more options than are shown in Figure 3-1. We will discuss many of these options as we use them in developing assembly language programs.

Rather than jumping into technical explanations at this time, let's see how the Editor, Assembler, and ABUG Monitor work by going through a demonstration of a program from the Software Development System Owner's Manual.

Turn off your computer (if it is on), plug in the SDS80C cartridge, and turn the computer on. If you are using some other assembler, follow the directions in its owner's manual.

The Editor

The SDS80C comes up in the Editor Program. The screen is blank except for the top line. Two important numbers are shown. The number on the right side of the top line shows the number of bytes of memory available for the text buffer. The text buffer is an area of memory used to store the text that you are about to enter. The other

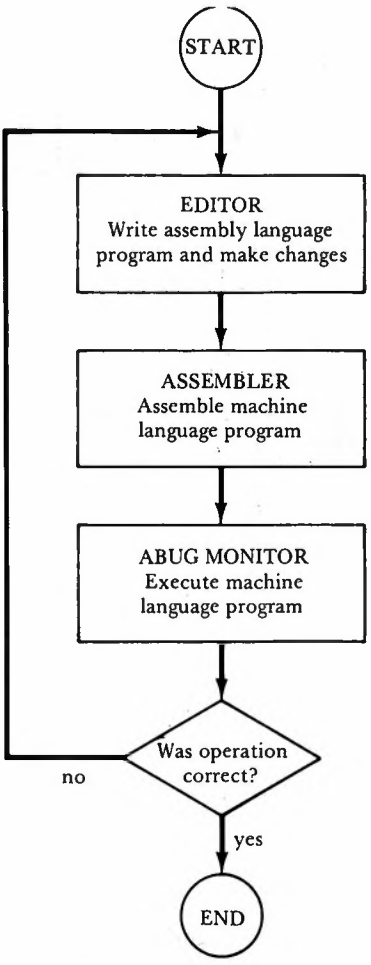
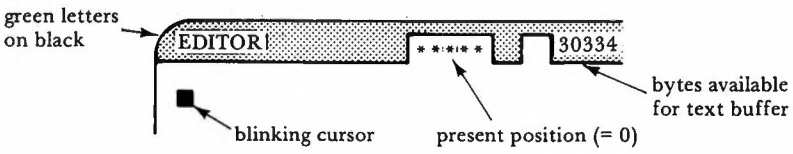
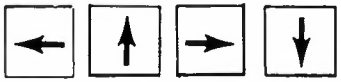


Figure 3-1. Flow of Software Development System

number shows your present position within a file. Because you haven't started yet, you are at the zero position. When you are at this position, a row of asterisks is displayed.



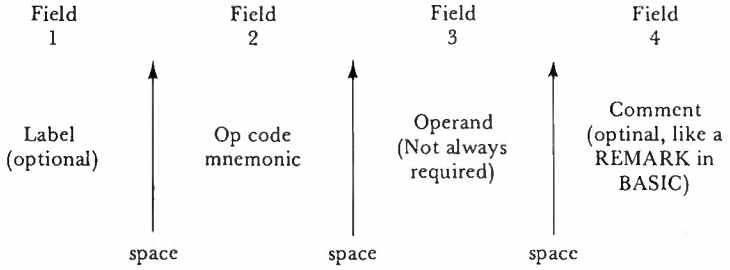
The Editor is screen oriented. What is seen on the screen is in the text buffer. The cursor may be moved about the file in the text buffer (as seen on the screen) by using the arrow keys:



The size of the source buffer is limited to the amount of memory in your computer. We are using a 32K computer. That's why 30,334 available bytes are shown on the screen. If the available bytes number decreases all the way to zero, the buffer is full. You would have to erase some of the text before you could put any more in.

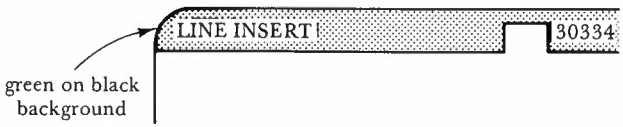
While in the Editor, all keys will automatically repeat while they are held down. Using the arrow keys in this way allows you to move rapidly throughout the file that you are editing.

Some editors use line numbers for the file being edited. The SBS80C system does not use line numbers; therefore, there are only four fields for each line of text.



To enter a program by the Editor, you first give the command to insert a line. Single letter commands are used. The letter L is used to insert lines.

Type: L



The Editor is now ready for the program to be entered. Here is the program that we will use for demonstration.

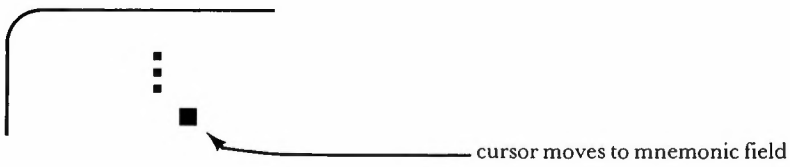
Table 3-1. Program 4—INVERT

Field 1 Label	*Field 2 Mnemonic*	*Field 3 Operand*	*Field 4 Comment*
	NAM	INVERT	
START	LDX	#$400	SCREEN ADDRESS
LOOP	LDA	,X	GET CHARACTER
	EORA	#$40	INVERT COLOR
	STA	,X+	SAVE; INCREMENT X
	CMPX	#$600	END OF SCREEN
	BLO	LOOP	DO WHOLE SCREEN
	SWI		CALL ABUG
	BRA	START	AND DO IT AGAIN
	END	START	

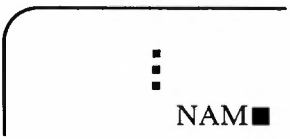
Notice that only the second and third lines have an entry in the Label field. To skip to the mnemonic field when there is no label, press the space bar.

Let's now get back to entering the program. You previously typed L and are ready to enter the first line. To enter the first line

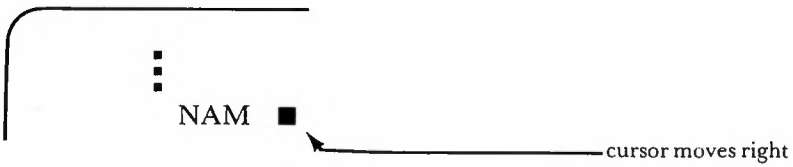
1. Press the space bar to move to the mnemonic field



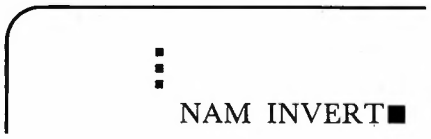
2. Type: NAM



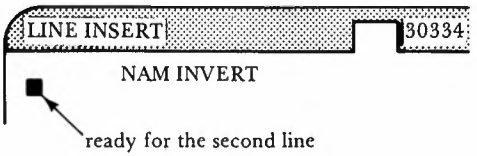
3. Press the space bar to leave the mnemonic field



4. Type: INVERT



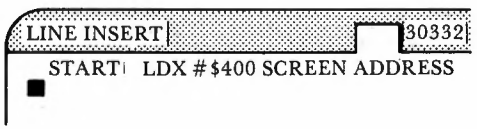
5. Press the ENTER key to enter the complete line



The first line displays the name of the program NAM INVERT. It will not be converted to machine code. It merely identifies the source program.

To enter the second line

1. Type: START ←in label field
2. Press the space bar
3. Type: LDX ←in mnemonic field
4. Press the space bar
5. Type: #$400 ←in operand field
6. Press the space bar
7. Type: SCREEN ADDRESS ←in comment field
8. Press ENTER ←terminates line 2

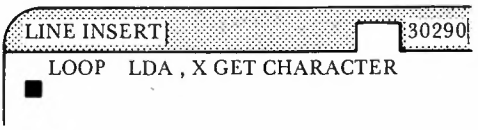


Notice the changes to the numbers in the top line:

30322 bytes available.

To enter the third line

1. Type: LOOP and press space bar ←in label field
2. Type: LDA and press space bar ←mnemonic field
3. Type: ,X and press space bar ←operand field
4. Type: GET CHARACTER ←in comment field
5. Press ENTER ←terminates line 3



To enter the fourth line

1. Press the space bar ←nothing in label field
2. Type: EORA and press space bar
3. Type: #$40 and press space bar
4. Type: INVERT COLOR
5. Press ENTER ←terminates line 4

Continue on. If there is nothing in the label field, be sure to press the space bar. Pressing the space bar terminates a field. When you get to the eighth line (SWI instruction)

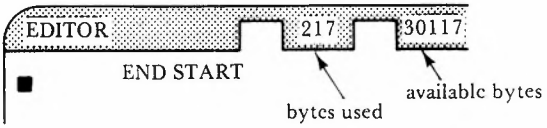
1. Press the space bar
2. Type: SWI and press the space bar
3. Type: CALL ABUG
4. Press ENTER

The assembler will recognize that the SWI instruction has no operand. Therefore, you do not have to type the extra space to move to the comment field.

When the complete program is in, the screen should look like this:

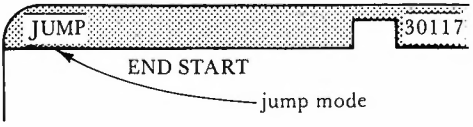
The BREAK key is used by the SDS80C to exit the INSERT LINES mode. A return is made to the command mode.

Press: BREAK to return to the command mode.

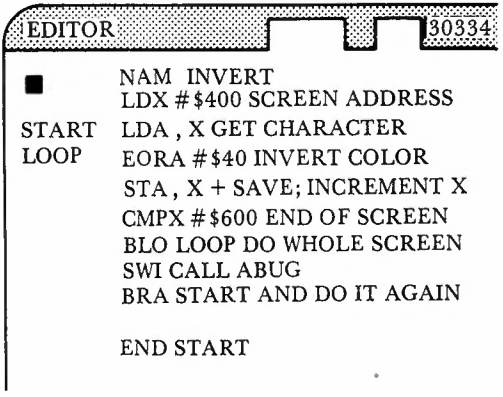


An easy way to view the entire assembly language program is to JUMP to the Beginning of the program by using the following two commands.

Press: J



Press: B (for beginning)



The program you have just entered with the editor is called the source program. It is made up of assembly language instructions. You are now ready to proceed to the assembler.

> The command for leaving the editor and entering the assembler is: @

The Assembler

After the program has been entered, the next step is performed by the assembler program. The assembler will convert the assembly language instructions into machine language codes and assign memory locations to the machine language codes. Before it performs its task, it allows you to select from several options. After pressing the @ key to enter the assembler, you are allowed to choose from these options:

L	produce a listing
S	produce a sorted symbol table
M	generate object code to memory
T	generate an object cassette tape
!	start listing in single step mode
3	send output to 32 column printer
4	send output to 40 column printer
8	send output to 80 column printer
=	do not assemble; go to ABUG instead

We will use the options L, S, and M this time to produce a listing on the video screen, produce a sorted symbol table, and put the object code (machine language codes) into memory. We'll also include the ! comment so that we can step through the program one line at a time.

The space bar controls the single step mode. Each time you press the space bar, a new line is displayed. Remember, the Editor has produced a source program; the Assembler converts the source program (assembly language) into an object program (machine language).

Type: L S M ! and press ENTER.

Then press the space bar until the top of the program is just under the command LSM!.

```
LSM!
0001 0767
        NAM INVERT
0002 0767 8E0400                ← machine codes produced by
START LDX #$400 SCREEN ADDRESS  ← assembler code
0003 076A A684
LOOP  LDA ,X GET CHARACTER
0004 076C 8840
        EORA #$40 INVERT COLOR
0005 076E A784
        STA ,X+ SAVE;INCREMENT X
0006 0770 8C0600
        CMPX #$600 END OF SCREEN
0007 0773 25F5
        BLO LOOP DO WHOLE SCREEN
0008 0775 3F
```

Press the space bar several times until the ABUG prompt appears.

```
0006 0770 8C0600
       CMPX #$600 END OF SCREEN
0007 0773 25F5
       BLO LOOP DO WHOLE SCREEN
0008 0775 3F
       SWI CALL ABUG
0009 0776 20EF
       BRA START AND DO IT AGAIN

0010 0778
       END START
LOOP 076A START 0767            ←—— symbol table

ABUG:■
```

Notice that the machine language program lines are numbered at the left (0001-0010). Following the line number is the memory location in which the machine codes have have placed. The machine code for the instructions and any data follow the memory location.

After each line of machine codes, the assembler instructions that produced the machine codes are shown. The last line of the display shows the symbol table consisting of the two labels used (LOOP and START) and the memory locations at which they occur in the program.

Here is a comparison of the assembler's source program and the machine language object program that was produced.

Table 3-2. Comparison of Assembly and Machine Languages

Source Program	*Object Program*
LDX #$400 SCREEN ADDRESS	0767 8E 04 00
LDA ,X GET CHARACTER	076A A6 84
EORA #$40 INVERT COLOR	076C 88 40
STA ,X + SAVE;INCREMENT X	076E A7 80
CMPX #$600 END OF SCREEN	0770 8C 06 00
BLO LOOP DO WHOLE SCREEN	0773 25 F5
SWI CALL ABUG	0775 3F
BRA START AND DO IT AGAIN	0776 20 EF
END START	0778 ← no machine code generated by the end instruction

If the single step command (!) had been omitted, the program would have scrolled past so fast that you would have only been able to read the last part of the program.

The ABUG Monitor

You have used the Editor Program to write the source program (assembly language) and the Assembler Program to assemble an object program (machine language). It's now time to use the ABUG Monitor to execute (or run) the object program.

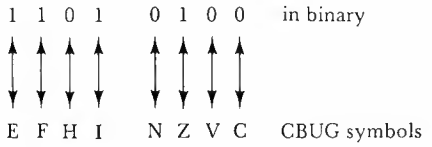
When the M option is used in the assembler to place the object program in memory, control is automatically passed to ABUG so that the object program may be executed.

After the program was assembled, you saw the ABUG prompt appear on the screen.

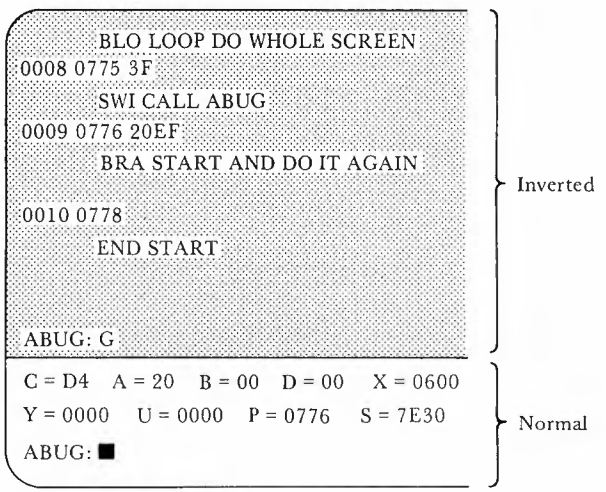
```
⋮
ABUG:■
```

One method that you may use to execute the program when in ABUG is to type G. When you type G, each location on the screen is inverted (appears as green on black) including the ABUG command G. Wow! That was fast. The bottom three lines are not inverted because they were printed after the program was interrupted by the SWI instruction. The first two of these lines show the condition codes that existed when the interrupt occurred.

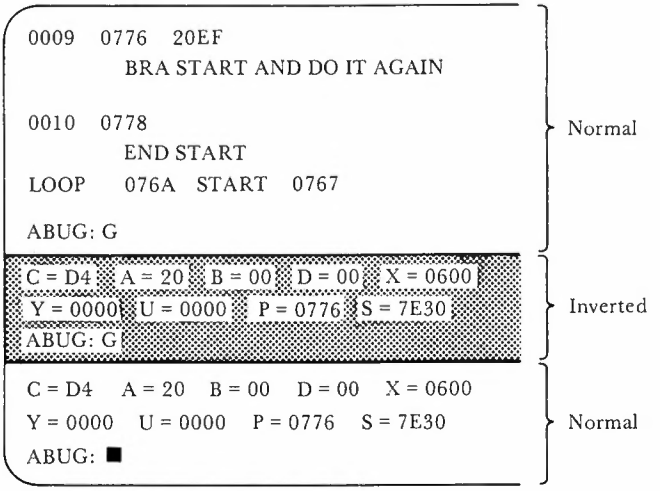
Notice that ABUG labels the condition codes register as register C (rather than EFHINZVC as the CBUG monitor did). C = D4 can be translated into binary digits as follows:



The display is as follows:



Now, press G again. You'll see another reversal.



This may be repeated as often as you like. Each time, all screen characters are inverted and three new lines appear at the bottom.

> The exit command from ABUG is the asterisk (*).

Press the asterisk when you are ready to stop experimenting with the program. The next section discusses the assembly instructions used.

Assembly Language Used in Program 4

Assembly language mnemonics are much easier to understand than the machine codes. The assembler takes most of the work out of preparing a machine language program. It assigns all the memory locations, determines the necessary machine codes, calculates the branches necessary, and presents you with a complete machine language program.

Here is an explanation of Program 4.

NAM INVERT	assigns the name "INVERT" to the program; this instruction is not translated into machine code

START	LDX #$400	loads the X register with $400; the # symbol indicates the immediate addressing mode; the label (START) is used as a reference for later branch instructions
LOOP	LDA ,X	load accumulator A from the address in X; indexed addressing mode; label (LOOP) is for branch reference
	EORA #$40	Exclusive OR accumulator A with $40; immediate addressing mode; this inverts the character to be displayed
	STA ,X +	store accumulator A in memory stored in X; increment X; indexed addressing mode
	CMPX #$600	compare value in X with $600; immediate addressing mode; $600 highest text screen memory + 1
	BLO LOOP	branch if value in X is lower than $600 back to instruction labeled LOOP; relative addressing mode
	SWI	software interrupt sends computer to ABUG; inherent addressing mode
	BRA START	branch always back to instruction labeled START; relative addressing mode
	END START	END is called a pseudo-operation and is not translated to machine code; tells the assembler where to stop

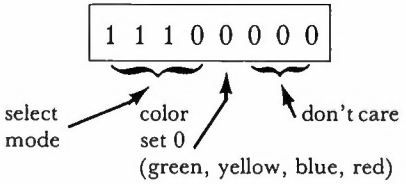
Designing a Graphics Program

Some planning is necessary to successfully develop a machine language program. This is especially true when using graphics. The graphics mode must be selected, the VDG (Video Display Generator) mode register must be set to match the graphics mode selected, the graphics screen should be cleared, and the color values must be loaded into the correct screen locations.

We'll develop a short program that will select a four-color graphic mode (called mode 6C) and display a red color bar near the center of the screen. In Extended Color BASIC, the graphics mode we will use would be set up by a PMODE 3,1 statement. In assembly language, it's a little more complicated.

To set mode 6C, three registers must be set as follows:

1. Location $FF22 is loaded with $E0



2. Locations $FFC3 and $FFC5 are set to 1 by storing any value you wish. The value is unimportant.

Four "pages" of memory are used in graphics mode 6C. The video memory starts at $0400. It will extend from there up to, but not including, $1C00.

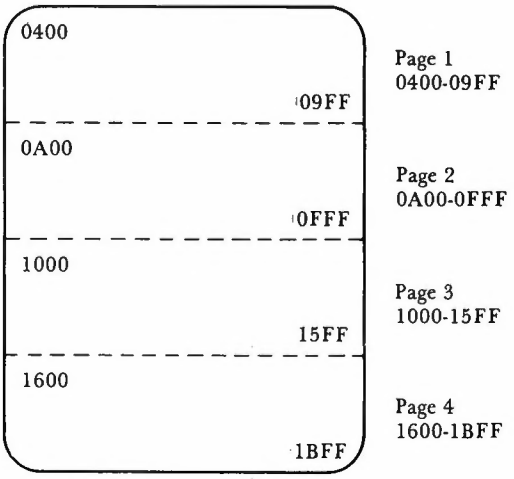
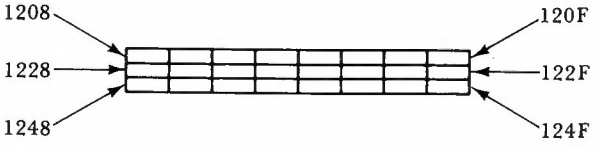
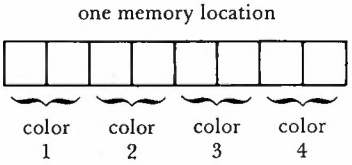


Figure 3-2. Memory Used for Graphics

We will put our color bar near the center of the screen. To make it highly visible, we'll use three rows of red color, eight bytes wide.



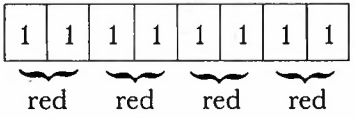
The colors are turned on by loading bytes into the video memory. In mode 6C, two bits are used to color one point. Therefore, one byte can set four adjacent color points.



The color codes for color set 0 are

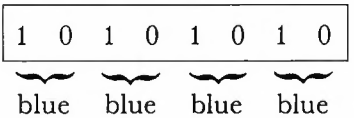
00	green
01	yellow
10	blue
11	red

We will be drawing a red color bar made up of bytes in the following form:



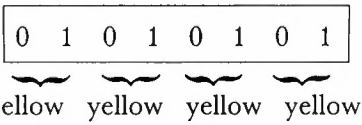
= $FF when loaded gives four red points

Four blue points would be



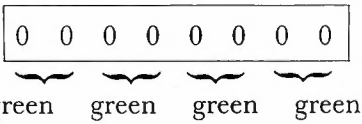
= $AA

Four yellow points



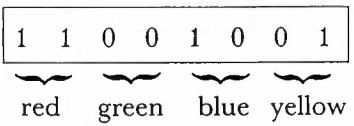
= $55

Four green points



= $00

You can mix colors within the byte also:



= $C9

You can see that to clear the screen (set all memory locations $0400-$1BFF to green) you must load lots of $00 bytes into memory locations. We will do just that in Program 5.

As your assembly language programs grow, you will find it to your advantage to save them on a cassette tape. Even though the assembler may assemble your source program into an object program, the final result may not be quite what you desire. After executing the program, you may want to alter the source program. To do this, you must go back to the Editor.

> Typing an asterisk (*) will return you to the Editor from ABUG.

After getting back to the Editor, you may find that the source program has disappeared. If your source program has been saved on tape, you may load it in and proceed with the alterations. If the source program is gone and it was not saved, you must enter it again from the keyboard.

> See Appendix A for saving and loading source and object programs.

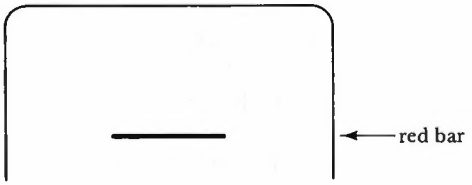
Enter Program 5 with your Editor, save it on tape, assemble it, and then execute it.

Program 5—Color Bar

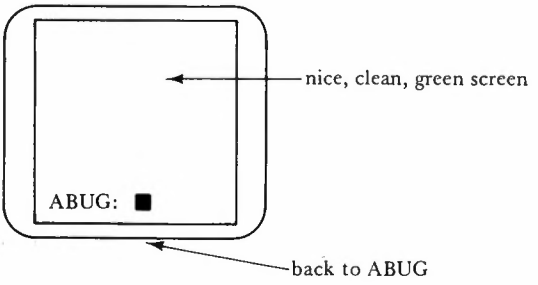
Block Number	*Assembly Language Program*	
1		ORG $1E00
		LDA #$E0 SELECT MODE
		STA $FF22
		STA $FFC3
		STA $FFC5
2		CLRA CLEAR SCREEN
		CLRB
		LDX #$400
	LOOP1	STD ,X + +
		CMPX #$1C00
		BLO LOOP1

```
3                LDX #$1208 FIRST LINE
                 JSR DISPLA DRAW IT
                 LDX #$1228 SECOND LINE
                 JSR DISPLA
                 LDX #$1248 THIRD LINE
                 JSR DISPLA

4       LOOP2    JSR $A1B1 POLL KEYBOARD
                 CMPA #$58 IS IT X?
                 BNE LOOP2 IF NOT, LOOK AGAIN

5                LDA #$60 LOAD SPACE
                 LDX #$400 TOP OF SCREEN
        LOOP3    STA ,X+ ONE AT A TIME
                 CMPX #$600 BOTTOM YET?
                 BLO LOOP3 IF NOT KEEP ON

6                LDA #$00 SET UP TEXT MODE
                 STA $FF22
                 STA $FFC2
                 STA $FFC4
                 RTS RETURN TO MONITOR

7       DISPLA   LDB #8 DISPLAY 8 BYTES
        GET      LDA #$FF
                 STA ,X+
                 DEC B
                 BNE GET
                 RTS
```

A red color bar should appear near the center of the screen.



To exit the program, press X.

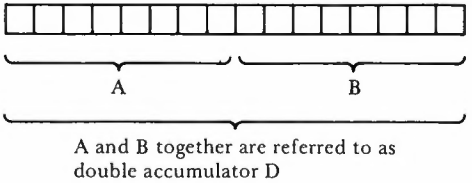


Now, let's see how it was done.

New Instructions Used in Program 5

Program 5 has been blocked into seven functional groups of instructions.

1. The first statement tells the assembler to put the ORiGin of the program at location #1E00 when it is assembled. The area used for graphics in this program extends through $1BFF. Therefore, the ORG pseudo-operation moves the program above the graphics screen memory. The other four instructions in block 1 select the graphics mode and the color set. This will be explained in more detail in Chapter 4.
2. This block clears the video screen. Registers A and B are cleared. Register X is loaded with the start address of graphics memory. Accumulator D is a 16-bit combination of accumulators A and B.



D is clear because both A and B were cleared. D is stored into the memory location in X and X + 1. The X register is then compared with $1C00 (where the program starts). A branch is made

back to loop 1 (the STD instruction) if the value in X is lower than $1C00 (BLO = Branch if LOwer).

3. The location where the first red line will be drawn ($1208) is loaded into the X register. The computer then jumps to a subroutine (block 7) to display the first line of the color bar. On return, the X register is loaded with $1228. This is one graphic line below the first one. The display subroutine then draws the second line. This is repeated once more for the third line at location $1248.

three lines drawn, but there
is no space between them

4. We have used a subroutine in the Color Computer ROM (location $A1B1) to scan the keyboard. A note of caution should be added. Whenever a routine is used from ROM, there are possibilities that some future version of Color Computer may have added or deleted certain ROM routines. The locations of some routines may also be moved in future versions. As a final precaution, consult the owner's manual for the ROM version you are using.

   Upon return from this subroutine, the value of any keystroke will appear in accumulator A. Therefore, we can compare the value in A with $58 (the ASCII code for the letter X). If not found, it branches back (BNE) to scan again. This allows you time to look at the finished red bar. Press the X key when through looking.

5. When the X key is pressed, the accumulator is loaded with $60 (ASCII code for space). The X register is used to index each memory location of text memory ($400-600). The space is loaded into each location in that range.

6. This section prepares the screen mode for text. This will be explained in Chapter 4.

7. This is the subroutine that puts eight bytes of graphics information into screen memory. To draw longer lines, you could increase the value loaded into accumulator B. To change colors, change the value loaded into accumulator A.

Summary

- An assembler is used to translate assembler mnemonics into machine language instructions that the computer can understand. It takes care of all the tedious details that were performed in hand-assembling the programs in the first two chapters. Assemblers are usually composed of three parts:
  1. an editor that is used to write and edit the source program in assembly language
  2. an assembler that translates the source program into an object program made up of machine language instructions and data
  3. a machine language monitor that allows you to execute the object program, examine memory and registers, and change the data in memory and registers

  The three parts, working together, allow you to develop machine language programs in a logical way with a minimum of detail.
- The editor maintains a data file in memory. The file is made up of assembly language instructions and data that you enter. Some editors, such as the SDS80C demonstrated in this chapter, display the amount of memory used by your file (in creating the source program) and the amount of memory still available for use. This is very valuable if you are creating long programs and/or you have a limited amount of memory in your computer. Your file may be edited while you are in the editor mode. Each program line consists of up to four fields, or areas, as follows:
  1. label—optional, often used as a reference for branch instructions
  2. op code—the assembler instruction (mnemonic)
  3. operand—some instructions require further information beyond the op code
  4. comment—optional, used to document your program with explanations that describe what the program is doing
- The assembler translates the source program into machine language codes and assigns memory locations for all instructions and data. If errors occur in your source program, a listing will show where the error occurred and what kind of error it is. You may then go back to the editor and make any necessary corrections. When the program is error free, the object program created by the assembler may be generated into memory in preparation for execution. Other options are normally available from the assembler, such as generating a cassette tape and sending the output to a printer.

- The machine language monitor allows you to execute the object program after the assembler generates it to memory. In addition, you can examine memory and registers and you can change data in memory and registers. Other options may be available.
- The amount of memory used to create graphics depends upon the mode that you select. In this chapter, we have used mode 6C, which uses screen memory from $400 through $1BFF.
- Several steps are necessary to prepare for a four-color graphic display in mode 6C. You must
  1. clear the area of memory that will be used for graphics
  2. load data into memory location $FF22 to select the mode and the color set to be used
  3. load or store data into memory locations $FFC3 and $FFC5—it doesn't matter what the data is

Chapter Test

1. Name the three programs of the Software Development System used in this chapter.
   a. _____
   b. _____
   c. _____
2. Describe the main function of each of your answers to exercise 1 of the chapter test.
   a. _____
   _____
   b. _____
   _____
   c. _____
   _____
3. When the computer is first turned on with the SDS80C in the cartridge slot, which one of the three programs is accessed first?
   _____
4. Two numbers are shown on the top line of the screen when the SDS80C is in the editor mode. What does each tell you?
   a. The number on the extreme right tells _____
   _____
   b. The other number tells _____
   _____

5. What limits the size of the source buffer in the editor mode?

   _____

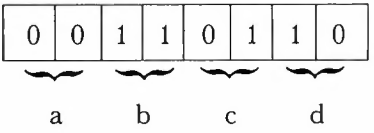
6. How do you move the cursor within your file when in the editor mode? _____

   _____

7. Name the four fields used when creating a source program.
   a. _____
   b. _____
   c. _____
   d. _____
8. How do you move from one field to another when writing the source program? _____

   _____

9. What key do you press to enter a line of text in the source program after it has been typed? _____
10. What key is pressed to exit the INSERT LINES mode of the SDS80C editor? _____
11. The assembly language program created in the editor is called the _____ program. The machine language program created by the assembler is called the _____ program.
12. What character is used to exit ABUG? _____
13. Name the colors that would be produced by the following data byte for mode 6C, color set 0.



   a. _____ c. _____
   b. _____ d. _____

Answers to Odd-Numbered Exercises in Chapter Test

1. a. editor
   b. assembler
   c. ABUG monitor
3. the editor

5. the amount of memory in your computer

7. a. label
   b. op code(mnemonic)
   c. operand
   d. comment

9. ENTER key

11. source, object

13. a.green b.red c.yellow d.blue

Chapter 4

Color Graphics

In order to create intelligent graphics on the Color Computer in machine language, it is necessary to understand how to use two important components of the computer.

1. One of these components is the dynamic Random Access Memory (RAM) controller chip (called SAM, Synchronous Address Multiplexer). This integrated circuit provides refresh and address multiplexing for the RAM. It also provides all of the system timing and device selection. The device selection is important in producing graphics.
2. The Video Display Generator (VDG) provides the interface to video and allows several different alphanumeric and graphic modes. The function of the VDG is controlled by one of two Peripheral Interface Adapters (PIAs). When this information is combined with RAM data, the VDG generates the composite video and color information for the video modulator circuitry.

In this chapter, we will discuss the use of the following four-color graphics modes.

Table 4-1. Four-Color Graphic Modes

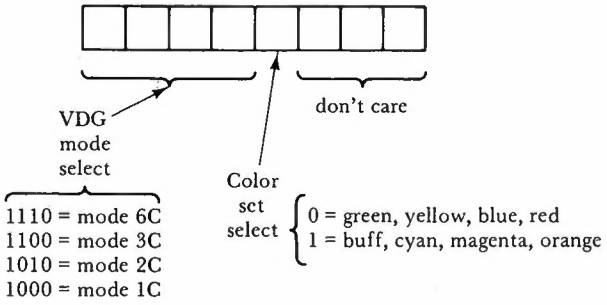
Resolution	*Graphics Element*	*Memory Used*	*Remarks*
128 × 192		$0400-$1BFF	This mode is called 6C. The screen display is 128 elements wide by 192 elements high. Each element is 2 units wide by 1 unit high and may be 1 of 4 colors. 6K of memory is used.
128 × 96		$0400-$0FFF	This mode is called 3C. The display is 128 elements wide by 96 elements high. Each element is 2 units wide by 2 units high and may be 1 of 4 colors. 3K of memory is used.
128 × 64		$0400-$0BFF	This mode is called 2C. The display is 128 elements wide by 64 elements high. Each element is 2 units wide by 3 units high and may be 1 of 4 colors. 2K of memory is used.
64 × 64		$0400-$07FF	This mode is called 1C. The display is 64 elements wide by 64 elements high. Each element is 4 units wide by 3 units high and may be 1 of 4 colors. 1K of memory is used.

Setting SAM, VDG, and Color Bytes

As was mentioned, the correct graphic mode must be selected by the SAM and VDG chips. This must be done before the data bytes are entered into the graphics memory. Go back and take another look at block one of Program 5—Color Bar. That program selected graphics mode 6C by storing the value $E0 into these memory locations

$FF22 $FFC3 $FFC5

Memory location $FF22 controls the VDG and selects the color set to be used. Individual bits make the selections as shown.



Certain registers must be set or cleared to match SAM with the VDG mode selected at $FF22. The registers are set or cleared in the following way.

A LOAD from or a STORE to instruction

$FFC0 clears V0
$FFC1 sets V0
$FFC2 clears V1
$FFC3 sets V1
$FFC4 clears V2
$FFC5 sets V2

Mode	*V2*	*V1*	*V0*	*Registers Set*
6C	1	1	0	Set $FFC5 and $FFC3
3C	1	0	0	Set $FFC5
2C	0	1	0	Set $FFC3
1C	0	0	1	Set $FFC1

Table 4-2 shows the instructions that are necessary to set up a four-color graphics mode.

Table 4-2. VDG and SAM Instructions for Four-Color Graphics

Mode	*Color Set*	*Instructions*	
6C	0	LDA#$E0	
		STA $FF22	←VDG mode, color set
		STA $FFC3	←set V1
		STA $FFC5	←set V2
	1	LDA #$EB	
		STA $FF22	
		STA $FFC3	
		STA $FFC5	
3C	0	LDA #$C0	
		STA $FF22	
		STA $FFC5	←set V2
	1	LDA #$C8	
		STA $FF22	
		STA $FFC5	
2C	0	LDA #$A0	
		STA $FF22	
		STA $FFC3	←set V1
	1	LDA #$A8	
		STA $FF22	
		STA $FFC3	
1C	0	LDA #$80	
		STA $FF22	
		STA $FFC1	←set V0
	1	LDA #$88	
		STA $FF22	
		STA $FFC1	

The data actually stored in $FFC1, $FFC3, and $FFC5 does not matter. But you must load or store data in them to set those registers necessary to match the desired VDG mode.

To clear one of the SAM registers that has been previously set, load or store some value to the even-numbered counterpart of the register that is set.

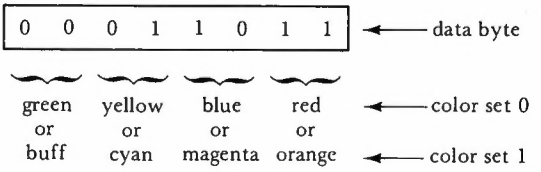
$FFC0 clears V0 (set by $FFC1)

$FFC2 clears V1 (set by $FFC3)

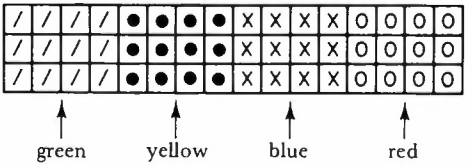
$FFC4 clears V2 (set by $FFC5)

For each of these four-color graphic modes, two bits of each data byte select a given color for one of four graphic elements.

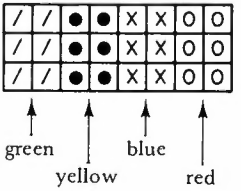
Examples



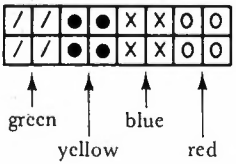
For mode 1C—color set 0, the above data byte would produce the following graphic elements:



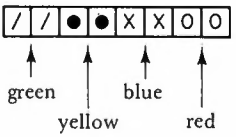
For mode 2C—color set 0, the data byte shown would produce these graphic elements:



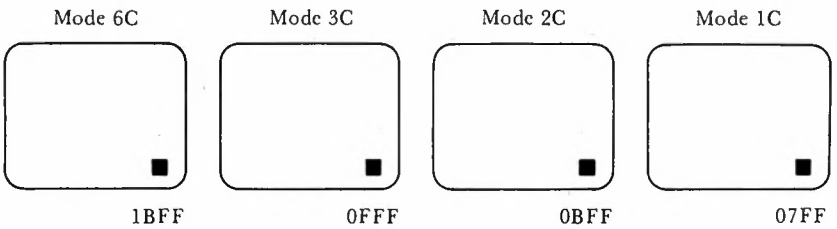
For mode 3C—color set 0, the data byte shown would produce:



For mode 6C—color set 0, the data byte shown would produce:



The area of memory where the data bytes are placed depends upon the graphic mode being used. The highest memory location will be different for each mode.



Putting a Graphics Program Together

To demonstrate the difference in resolution of the four-color graphic modes, we'll draw an orange rectangle on a buff background in the next program. Graphic modes 6C and 1C will be used to show the two extremes.

Orange and buff are members of color set 1. The two-bit code for orange is 11, and for buff it is 00. According to the data in Table 4-2, we must store $E8 into $FF22 and set V1 and V2 by "writing to" $FFC3 and $FFC5.

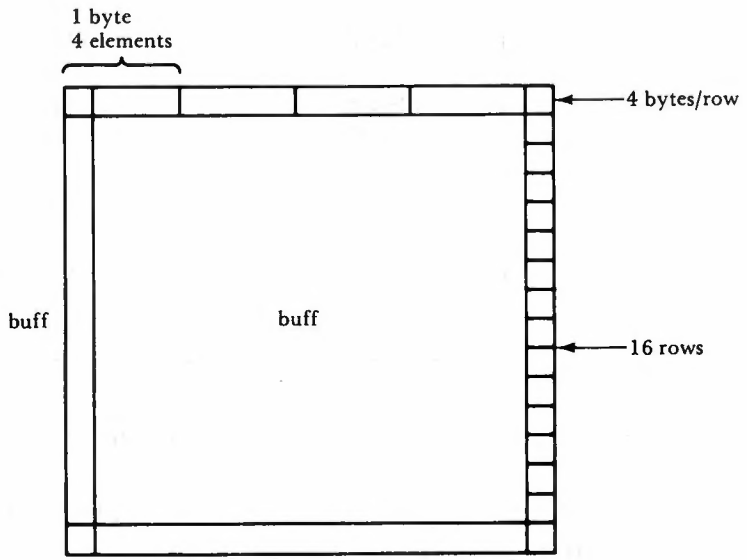
To design our rectangle, we must remember that one data byte will set four adjacent elements. Thus a data byte of

1	1	1	1	1	1	1	1

will color four adjacent elements (: one element for 6C):

4 orange elements

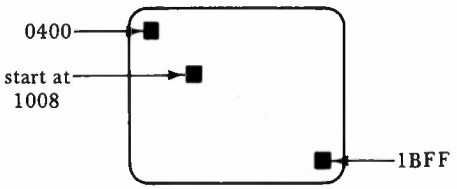
To draw an orange rectangle 16 elements wide by 16 elements high, we will use 4 bytes in each row with 16 rows.



The data would be as follows:

1st row	FF FF FF FF	←each element orange
2nd–15th row	C0 00 00 03	←1st and last elements orange
	↓ ↓ ↓ ↓	
16th row	FF FF FF FF	←each element orange

Next, we must consider where the rectangle should be placed on the screen. We'll put it somewhere above and to the left of center. For mode 6C, we have:



We want to store FF at 1008, 1009, 100A, and 100B

C0 at 1028 102B 03 at

1048 104B

1068 106B

1088 00 elsewhere 108B

10A8 10AB

10C8 10CB

10E8 10EB

1108 110B

1128 112B

1148 114B

1168 116B

1188 118B

11A8 11AB

11C8 11CB

FF at → 11E8, 11E9, 11EA, and 11EB

Program 6—Drawing an Orange Rectangle

Now that we know what we have to do, let's lay out a play to do it.

1. select the correct graphic mode
2. clear the graphics screen area
3. draw the display
4. use the X key on the keyboard to exit the graphics program
5. clear the text screen and go back to the monitor
6. provide data in a table

Following these steps, we write the program.

```
0)              ORG $1E00

1)              LDA #$E8        ;SELECT 6C
                STA $FF22
                STA $FFC3
                STA $FFC5

2)              CLRA            ;CLEAR SCREEN
                CLRB
                LDX #$400
          LOOP1 STD ,X++
                CMPX #$1C00
                BLO LOOP1

3)  a)          LDX #$1008
                LDY #TABLE
          FIRST LDA ,Y+         ;TOP ROW
                CMPA #$FF
                BNE DOWN
                STA ,X+
                BRA FIRST

    b)    DOWN  LDX #$1028
          NEXT  STA ,X+         ;MIDDLE ROWS
                LDA ,Y+
                CMPA #$C0
                BLO NEXT
                LEAX $1C,X      ;BUMP DOWN
                CMPX #$11E8
                BNE NEXT

    c)    LAST  STA ,X+         ;LAST ROW
                LDA ,Y+
                CMPA #$FF
                BEQ LAST

4)        XKEY  JSR $A1B1       ;LOOK FOR X KEY
                CMPA #$58
                BNE XKEY

5)              LDA #$60
                LDX #$400
          LOOP2 STA ,X+         ;CLEAR TEXT AREA
                CMPX #$600
                BLO LOOP2
                LDA #0          ;SET TEXT MODE
                STA $FF22
                STA $FFC2
                STA $FFC4
                RTS             ;RETURN TO MONITOR
```

```
6)          TABLE   FCB $FF,$FF,$FF,$FF
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $C0,0,0,3
                    FCB $FF,$FF,$FF,$FF
                    FCB 0,0,0,0
                    END
```

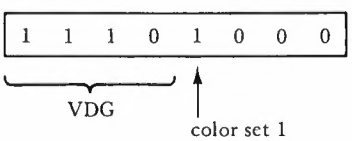
When you have entered the assembly language program, it would probably be a good idea to save the source program on a cassette tape before assembling it. Appendix A shows how to do this for the SDS80C system. If you have a different assembler, read its operator's manual for directions on saving to tape.

How Program 6 Works

This looks like a long program. However, you should keep in mind that long programs are just a series of short programs connected together. Each short part usually performs a certain function. If you wish, you can test each functional part before connecting them all together. Let's examine Program 6 by its functional parts.

1. Select the Correct Graphics Mode

   After the origin pseudo-op, mode 6C is selected by storing #E8 in memory location $FF22. We used $E0 in Program 5 to select color set 0. This time, we will use color set 1. Therefore, $E8 is stored. The same data is stored in $FFC5 and $FFC3 to set V2 and V1. No new instructions are used in this part.

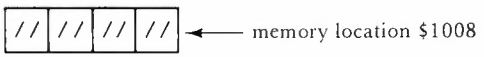


2. Clear the Graphic Screen Area

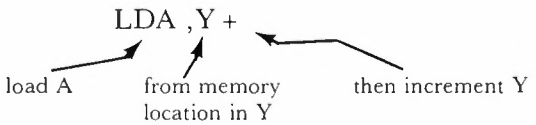
   This part is the same as that used in Program 5. No new instructions are used.

3. Draw the Display

   This part has been divided into three subsections. *Part 3a*—the top row of the rectangle is drawn. The X register is loaded with the screen memory location where the first data byte ($FF) will turn on four orange elements (LDX #$1008).

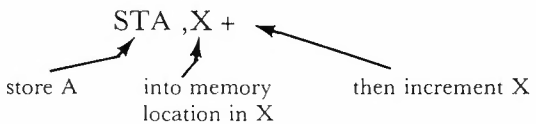


The Y register is loaded with the memory location of the first byte of the data table (LDY #TABLE). The assembler will decide where the table should be located (at the end of the program). The loop labeled FIRST is then executed. It loads accumulator A with a value from the memory location in register Y (the data table), and Y is incremented.



The value in A is then compared with $FF (CMPA #$FF).

If the value is FF, the branch (BNE DOWN) is not taken. The value in A is then stored in the screen memory contained in X, and X is incremented.



A branch is then taken back to the beginning of the loop (BRA FIRST).

If the value is not FF, a branch is made to part 3B. Notice the data table contains 4 FFs at the beginning. When C0 is loaded, the branch will be taken to part 3B.

Part 3b—this part is reached with $C0 in accumulator A. The instruction labeled DOWN loads the X register with $1028, which is one screen line below the first line used (DOWN LDX #$1028). The instruction labeled NEXT (STA ,X+) stores the value in A into the screen memory in X, and X is incremented. Accumulator A is then loaded with the next data value, and Y is incremented (LDA ,Y+).

The value in A is compared with $C0. If the value is lower than $C0 (either 00 or 03), a branch is made back to the instruction labeled NEXT (BLO NEXT).

If the value is equal to or greater than $C0, the branch is not taken. The X register is increased by $1C (LEAX $1C,X). This increases the value in the X register to point to the next screen line. A comparison is then made with the value in X and $11E8 to see if the last line of the rectangle has been reached (CMPX #$11E8). If not, a branch is made back (BNE NEXT) to store accumulator A into another screen location. If $11E8 is in the X register, the computer goes on to part 3c.

Part 3c—this part displays the last line (bottom) of the rectangle. The instruction labeled LAST stores the value in the memory location pointed to by X, and X is incremented (LAST STA ,X+). Accumulator A is then loaded with a new value from the location pointed to by Y, and Y is incremented (STA ,Y+). The value in A is compared with $FF.

If A holds FF, a branch is made back to LAST to store it (BEQ LAST). If A holds some other value, the branch is not taken. The computer goes on to part 4.

4. Look for X Key

   You've seen this section before. A jump is made into the ROM's POLCAT subroutine to look for an X keystroke. It then compares the value in A to see if it's $58 (ASCII "X"). If not, it looks again. If the X key has been pressed, the computer goes on to part 5.

5. Clear Text Screen and Go Back to Monitor

   This section was used in Program 5. It loads a space into all the text memory locations $400-05FF. The VDG and SAM chips are reset for text mode, and a return is made to the monitor.

6. Data Table

   This is the data table used to color the screen. Notice that it ends with a string of zeros to indicate that all data has been used. The data lines are each headed by FCB. This symbol indicates to the assembler that the values, separated by commas, are all single data bytes.

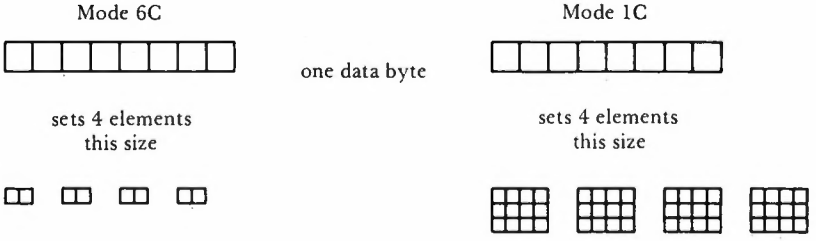
```
0001 0600                      ORG $1E00
0002 1E00 86E8                 LDA #$E8          SELECT 6C
0003 1E02 B7FF22               STA $FF22
0004 1E05 B7FFC3               STA $FFC3
0005 1E08 B7FFC5               STA $FFC5
0006 1E0B 4F                   CLRA              CLEAR SCREEN
0007 1E0C 5F                   CLRB
0008 1E0D 8E0400               LDX #$400
0009 1E10 ED81           LOOP1 STD ,X++
0010 1E12 8C1C00               CMPX #$1C00
0011 1E15 25F9                 BLO LOOP1
0012 1E17 8E1008               LDX #$1008
0013 1E1A 108E1E62             LDY #TABLE
0014 1E1E A6A0           FIRST LDA ,Y+           TOP ROW
0015 1E20 81FF                 CMPA #$FF
0016 1E22 2604                 BNE DOWN
0017 1E24 A780                 STA ,X+
0018 1E26 20F6                 BRA FIRST
0019 1E28 8E1028         DOWN  LDX #$1028
0020 1E2B A780           NEXT  STA ,X+           MIDDLE ROWS
0021 1E2D A6A0                 LDA ,Y+
0022 1E2F 81C0                 CMPA #$C0
0023 1E31 25F8                 BLO NEXT
0024 1E33 30881C               LEAX $1C,X        BUMP DOWN
0025 1E36 8C11E8               CMPX #$11E8
0026 1E39 26F0                 BNE NEXT
0027 1E3B A780           LAST  STA ,X+           LAST ROW
0028 1E3D A6A0                 LDA ,Y+
0029 1E3F 81FF                 CMPA #$FF
0030 1E41 27F8                 BEQ LAST
0031 1E43 BDA1B1         XKEY  JSR $A1B1
0032 1E46 8158                 CMPA #$58
0033 1E48 26F9                 BNE XKEY
0034 1E4A 8660                 LDA #$60
0035 1E4C 8E0400               LDX #$400
0036 1E4F A780           LOOP2 STA ,X+
0037 1E51 8C0600               CMPX #$600
0038 1E54 25F9                 BLO LOOP2
0039 1E56 8600                 LDA #$00
0040 1E58 B7FF22               STA $FF22
0041 1E5B B7FFC2               STA $FFC2
0042 1E5E B7FFC4               STA $FFC4
0043 1E61 39                   RTS
0044 1E62 FFFFFFFF       TABLE FCB $FF,$FF,$FF,$FF
0045 1E66 C0000003             FCB $C0,0,0,3
0046 1E6A C0000003             FCB $C0,0,0,3
0047 1E6E C0000003             FCB $C0,0,0,3
0048 1E72 C0000003             FCB $C0,0,0,3
0049 1E76 C0000003             FCB $C0,0,0,3
0050 1E7A C0000003             FCB $C0,0,0,3
0051 1E7E C0000003             FCB $C0,0,0,3
0052 1E82 C0000003             FCB $C0,0,0,3
```

```
0053 1E86 C0000003          FCB $C0,0,0,3
0054 1E8A C0000003          FCB $C0,0,0,3
0055 1E8E C0000003          FCB $C0,0,0,3
0056 1E92 C0000003          FCB $C0,0,0,3
0057 1E96 C0000003          FCB $C0,0,0,3
0058 1E9A C0000003          FCB $C0,0,0,3
0059 1E9E FFFFFFFF          FCB $FF,$FF,$FF,$FF
0060 1EA2 00000000          FCB 0,0,0,0
0061 1EA6                   END
```

Figure 4-1. Listing of Program 6

Program 7—Drawing an Orange Rectangle in Mode 1C

Let's compare the previous graphics mode (6C) with mode 1C, which has much lower resolution. It takes fewer data bytes to make the same size rectangle in mode 1C.



To draw a rectangle in mode 1C that is approximately the same size as we drew for 6C, our data table for mode 1C is much smaller.

```
FF FF
C0 03
C0 03
C0 03
C0 03
FF FF
```

Twelve bytes in mode 1C draw a rectangle of similar size as 64 bytes in mode 6C. Of course, the sides of the rectangle drawn in mode 1C are much thicker.

Compare Program 7 with Program 6. You will find that part 1 is changed to set up the screen for graphics mode 1C.

```
1.  ORG $1E00
    LDA #$88          select 1C
    STA $FF22
    STA $FFC1
```

Part 2 (clear screen) remains the same except that the graphics area for mode 1C begins at $400 and ends at $07FF (see Table 4-1).

```
2.          CLRA
            CLRB
            LDX #$400
    LOOP1   STD ,X++
            CMPX #$800
            BLO LOOP1
```

Part 3 has been changed to use accumulator B as a counter to store two bytes per screen line. This method of indexing screen lines is much more straightforward than the method used for Program 6. Also notice that the placement of data in screen memory has changed because less memory is used for this mode. The screen's center is in a different location.

```
3a.         LDX #$598
            LDY #TABLE
            LDB #2
    FIRST   LDA ,Y+           top row
            STA ,X+
            DEC B
            BNE FIRST
```

```
3b.         LDX #$598
   DOWN     LDB #2
   NEXT     LDA ,Y+          middle rows
            STA ,X+
            DEC B
            BNE NEXT
            LEAX $E,X
            CMPX #$5D8
            BNE DOWN

3c.         LDB #2
   LAST     LDA ,Y+          last row
            STA ,X+
            DEC B
            BNE LAST
```

Parts 4 and 5 remain the same as for Program 6, and the change in data is reflected in part 6.

```
6.  TABLE   FCB $FF,$FF
            FCB $C0,3
            FCB $C0,3
            FCB $C0,3
            FCB $C0,3
            FCB $FF,$FF
            FCB 0,0
            END
```

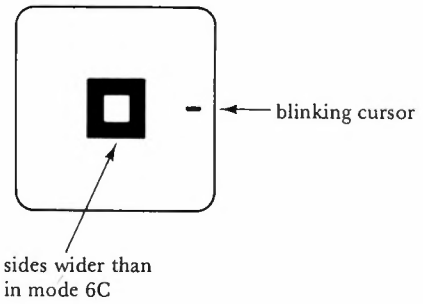
Enter the source program and assemble it. We used the Listing, Symbol table, and Printer options of the SDS80C system to send the listing to the printer. The result is shown in Figure 4-2.

```
0001 0600                       ORG $1E00
0002 1E00 8688                  LDA #$88          SELECT 1C
0003 1E02 B7FF22                STA $FF22
0004 1E05 B7FFC1                STA $FFC1
0005 1E08 4F                    CLRA
0006 1E09 5F                    CLRB
0007 1E0A 8E0400                LDX #$400
0008 1E0D ED81          LOOP1   STD ,X++
0009 1E0F 8C0800                CMPX #$800
```

```
0010 1E12 25F9              BLO LOOP1
0011 1E14 8E0588            LDX #$588
0012 1E17 108E1E5C          LDY #TABLE
0013 1E1B C602              LDB #2
0014 1E1D A6A0      FIRST   LDA ,Y+          TOP ROW
0015 1E1F A780              STA ,X+
0016 1E21 5A                DECB
0017 1E22 26F9              BNE FIRST
0018 1E24 8E0598            LDX #$598        MIDDLE ROWS
0019 1E27 C602      DOWN    LDB #2
0020 1E29 A6A0      NEXT    LDA ,Y+
0021 1E2B A780              STA ,X+
0022 1E2D 5A                DECB
0023 1E2E 26F9              BNE NEXT
0024 1E30 300E              LEAX $E,X
0025 1E32 8C05D8            CMPX #$5D8
0026 1E35 26F0              BNE DOWN
0027 1E37 C602              LDB #2           LAST ROW
0028 1E39 A6A0      LAST    LDA ,Y+
0029 1E3B A780              STA ,X+
0030 1E3D 5A                DECB
0031 1E3E 26F9              BNE LAST
0032 1E40 BDA1B1    XKEY    JSR $A1B1
0033 1E43 8158              CMPA #$58
0034 1E45 26F9              BNE XKEY
0035 1E47 8660              LDA #$60
0036 1E49 8E0400            LDX #$400
0037 1E4C A780      LOOP2   STA ,X+          CLEAR TEXT
0038 1E4E 8C0600            CMPX #$600
0039 1E51 25F9              BLO LOOP2
0040 1E53 8600              LDA #$00
0041 1E55 B7FF22            STA $FF22
0042 1E58 B7FFC0            STA $FFC0
0043 1E5B 39                RTS
0044 1E5C FFFF      TABLE   FCB $FF,$FF
0045 1E5E C003              FCB $C0,3
0046 1E60 C003              FCB $C0,3
0047 1E62 C003              FCB $C0,3
0048 1E64 C003              FCB $C0,3
0049 1E66 FFFF              FCB $FF,$FF
0050 1E68 0000              FCB 0,0
0051 1E6A                   END

DOWN    1E27  FIRST   1E1D  LAST    1E39  LOOP1   1E0D
LOOP2   1E4C  NEXT    1E29  TABLE   1E5C  XKEY    1E40
```

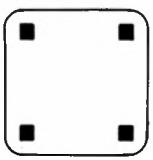
Figure 4-2. Listing of Program 7

When the program is assembled correctly, execute the object program. Here is how our screen looked.



Screen Memory for Four-Color Graphic Modes

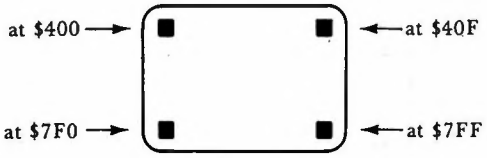
By now you may be slightly confused about where to put data to display something on the video screen because placement changes for each mode. We'll try to straighten this out with the next program. This program places one byte of graphic data in each of the four corners of the screen for each of the four-color graphic modes.



The size of the displayed points will vary in size as you go from one mode to another. The four corners are displayed first for mode 1C. They stay on the screen until you press the X key. Then the corners are displayed for mode 2C. When you press the X key again, the corners for mode 3C are displayed. Pressing the X key once more displays the corners for mode 6C. One more press of the X key will return you to the monitor.

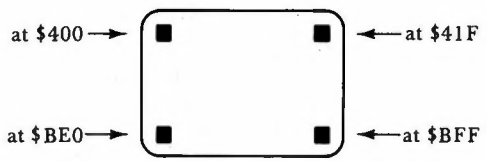
Once more, we'll break the program up into functional parts.

Part 1 displays the four corners for mode 1C.




```
ORG $1E00
LDA #$88
STA $FF22
STA $FFC1       ←turn on V0
JSR CLEARG
LDA #$FF        ←red
STA $400
STA $40F        ←four corners
STA $7F0
STA $7FF
JSR XKEY
JSR CLEART
STA $FFC0       ←turn off V0
```

Part 2 displays the four corners for mode 2C.

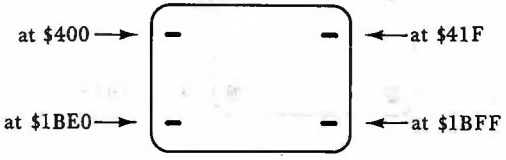



```
LDA #$A8
STA $FF22
STA $FFC3       ←turn on V1
JSR CLEARG
LDA #$FF
STA $400
STA $41F
STA $BE0
STA $BFF
JSR XKEY
JSR CLEART
STA $FFC2       ←turn off V1
```

Part 3 displays the four corners for mode 3C.


```
LDA #$C8
STA $FF22
STA $FFC4        ←turn on V2
JSR CLEARG
LDA #$FF
STA $400
STA $41F
STA $FE0
STA $FFF
JSR XKEY
JSR CLEART
STA $FFC4        ←turn off V2
```

Part 4 displays the four corners for mode 6C.




```
LDA #$E8
STA $FF22
STA $FFC3
STA $FFC5
JSR CLEARG
LDA #$FF
STA $400
STA $41F
STA $1BE0
STA $1BFF
```

```
JSR XKEY
JSR CLEART
STA $FFC2
STA $FFC4
RTS
```

Part 5 consists of subroutines used in each of the first four parts.

Clear graphic area

```
CLEARG    CLRA
          CLRB
          LDX #$400
LOOP1     STD ,X++
          CMPX #$1C00
          BLO LOOP1
          RTS
```

Look for X key

```
XKEY      JSR $A1B1
          CMPA #$58
          BNE XKEY
          RTS
```

Clear text area

```
CLEART LDA #$60
       LDX #$400
LOOP2  STA ,X+
       CMPX #$600
       BLO LOOP2
       STA #$00
       STA $FF22
       RTS
       END
```

A listing of the program is shown in Figure 4-3. You should notice that all the graphic modes have used memory location $400 as the beginning of the display (upper left corner). The beginning location can be changed, as you will see in Chapter 5. This allows changing graphic pages just as you were able to do in BASIC.

```
0001 0600                ORG $1E00
0002 1E00 8688           LDA #$88
0003 1E02 B7FF22         STA $FF22
0004 1E05 B7FFC1         STA $FFC1
0005 1E08 BD1E8F         JSR CLEARG
0006 1E0B 86FF           LDA #$FF
0007 1E0D B70400         STA $400
0008 1E10 B7040F         STA $40F
0009 1E13 B707F0         STA $7F0
0010 1E16 B707FF         STA $7FF
0011 1E19 BD1E9C         JSR XKEY
0012 1E1C BD1EA4         JSR CLEART
0013 1E1F B7FFC0         STA $FFC0
0014 1E22 86A8           LDA #$A8
0015 1E24 B7FF22         STA $FF22
0016 1E27 B7FFC3         STA $FFC3
0017 1E2A BD1E8F         JSR CLEARG
0018 1E2D 86FF           LDA #$FF
0019 1E2F B70400         STA $400
0020 1E32 B7041F         STA $41F
0021 1E35 B70BE0         STA $BE0
0022 1E38 B70BFF         STA $BFF
0023 1E3B BD1E9C         JSR XKEY
0024 1E3E BD1EA4         JSR CLEART
0025 1E41 B7FFC2         STA $FFC2
0026 1E44 86C8           LDA #$C8
0027 1E46 B7FF22         STA $FF22
0028 1E49 B7FFC5         STA $FFC5
0029 1E4C BD1E8F         JSR CLEARG
0030 1E4F 86FF           LDA #$FF
0031 1E51 B70400         STA $400
0032 1E54 B7041F         STA $41F
0033 1E57 B70FE0         STA $FE0
0034 1E5A B70FFF         STA $FFF
0035 1E5D BD1E9C         JSR XKEY
0036 1E60 BD1EA4         JSR CLEART
0037 1E63 B7FFC4         STA $FFC4
0038 1E66 86E8           LDA #$E8
0039 1E68 B7FF22         STA $FF22
0040 1E6B B7FFC3         STA $FFC3
0041 1E6E B7FFC5         STA $FFC5
0042 1E71 BD1E8F         JSR CLEARG
0043 1E74 86FF           LDA #$FF
0044 1E76 B70400         STA $400
0045 1E79 B7041F         STA $41F
0046 1E7C B71BE0         STA $1BE0
0047 1E7F B71BFF         STA $1BFF
0048 1E82 BD1E9C         JSR XKEY
0049 1E85 BD1EA4         JSR CLEART
0050 1E88 B7FFC2         STA $FFC2
0051 1E8B B7FFC4         STA $FFC4
0052 1E8E 39             RTS
```

```
0053 1E8F 4F            CLEARG CLRA
0054 1E90 5F                   CLRB
0055 1E91 8E0400               LDX #$400
0056 1E94 ED81          LOOP1  STD ,X++
0057 1E96 8C1C00               CMPX #$1C00
0058 1E99 25F9                 BLO LOOP1
0059 1E9B 39                   RTS
0060 1E9C BDA1B1        XKEY   JSR $A1B1
0061 1E9F 8158                 CMPA #$58
0062 1EA1 26F9                 BNE XKEY
0063 1EA3 39                   RTS
0064 1EA4 8660          CLEART LDA #$60
0065 1EA6 8E0400               LDX #$400
0066 1EA9 A780          LOOP2  STA ,X+
0067 1EAB 8C0600               CMPX #$600
0068 1EAE 25F9                 BLO LOOP2
0069 1EB0 8600                 LDA #$00
0070 1EB2 B7FF22               STA $FF22
0071 1EB5 39                   RTS
0072 1EB6                      END

CLEARG 1E8F   CLEART 1EA4   LOOP1   1E94   LOOP2   1EA9
XKEY   1E9C
```

Figure 4-3. Four Corners for Four-Color Graphic Modes

Table 4-3 shows all of the 6809 assembler instructions used through the first four chapters. Also listed are the programs in which they appeared.

Table 4-3. Instructions Used in First Four Chapters

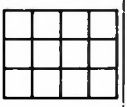
Instruction and Addressing Mode		*Program Where Used*							
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
BEQ	relative			X			X		
BLO	relative				X	X	X	X	X
BNE	relative	X	X	X		X	X	X	X
BPL	relative	X							
BRA	relative	X		X	X		X		
CLRA	inherent	X				X	X	X	X
CLRB	inherent					X	X	X	X
CMPA	immediate					X	X		X

Table 4-3. (*continued*)

Instruction and Addressing Mode		*Program Where Used*							
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
CMPB	immediate			X					
CMPX	immediate				X	X	X	X	X
DECB	inherent					X		X	
DEX	inherent		X	X					
END	pseudo-op				X	X	X	X	X
EORA	immediate				X				
FCB	pseudo-op						X	X	
INCA	inherent	X							
INCB	inherent	X	X	X					
JSR	extended					X	X		X
LDA	immediate	X	X	X		X	X	X	X
LDA ,X+	indexed				X				
LDA ,Y+	indexed						X	X	
LDB	immediate		X			X		X	
LDB ,Y+	indexed			X					
LDX	immediate		X	X	X	X	X	X	X
LDY	immediate			X					
LEAX n,X	indexed		X	X			X	X	
NOP	inherent	X	X	X					
ORG	pseudo-op					X	X	X	X
RTS	inherent		X	X		X	X		X
STA	extended	X	X	X		X	X	X	X
STA ,X+	indexed				X	X	X	X	X
STB	extended	X	X	X					
STD ,X++	indexed					X	X	X	X
SWI	inherent				X				
TFR A,B	inherent	X		X					
TFR B,A	inherent			X					

Summary

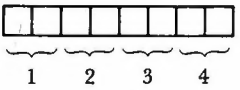
- The four-color graphic modes all use different amounts of memory because of different resolutions. The resolution depends on the size of the graphic elements displayed.

Mode	*Resolution*	*Element*	*Memory Used*
6C	128 × 192		$400-1BFF 6K
3C	128 × 96		$400-FFF 3K
2C	128 × 64		$400-BFF 2K
1C	64 × 64		$400-7FF 1K

- To select a given graphic mode, several registers must be set.

Mode	*Set Registers*	*Data in $FF22*
6C	$FFC5, $FFC3	E0-color set 0 E8-color set 1
3C	$FFC5	C0-color set 0 C8-color set 1
2C	$FFC3	A0-color set 0 A8-color set 1
1C	$FFC1	80-color set 0 88-color set 1

- One data byte sets four graphic elements to colors specified by two bits of the bytes.



Bits	*Color Set 0*	*Color Set 1*
00	green	buff
01	yellow	cyan
10	blue	magenta
11	red	orange

- Data bytes for coloring graphic elements may be accessed from a table and displayed by using index registers.

Example:

LDX #1008	load index register X with starting address of screen memory
LDY #TABLE	load index register Y with starting address of beginning of table
LDA ,Y+	load accumulator A from address in Y register; then increment Y
STA ,X+	store accumulator A into screen address contained in X register; then increment X

- Fewer data bytes are necessary for low resolution than for high resolution graphics, but the lines displayed using low resolution are larger.

Chapter Test

1. Draw the graphic elements for each of the following:
   a. mode 1C b. mode 2C c. mode 3C d. mode 6C

2. Tell the mode and color set determined by setting memory location $FF22 to the following values.
   a. E0 (hex) = mode _____ color set _____
   b. 10101000 (binary) = mode _____ color set _____
3. Tell the mode determined by storing data to the following.
   a. STA $FFC5 mode _____
   b. STA $FFC5 and $FFC3 mode _____
   c. STA $FFC1 mode _____
4. When leaving a graphic mode, certain registers must be cleared. Storing data in the following registers clears which mode?
   a. STA $FFC0 clears mode _____
   b. STA $FFC4 clears mode _____
5. What color is designated by the following bit pairs for color set 0?
   a. 01 color = _____
   b. 10 color = _____
   c. 00 color = _____
   d. 11 color = _____

6. What four-color elements would be set by these data bytes?
   a. E3 _____ _____ _____ _____
   b. 3C _____ _____ _____ _____
7. Give the data byte to set these four graphic elements.

green	blue	blue	red

= data byte _____

8. What is the memory location (approximately) of the center of the screen in each of these modes?
   a. 6C _____ b. 3C _____
   c. 2C _____ d. 1C _____
9. What are the graphic memory locations for the four corners of the display in mode 3C?
   a. upper left _____ b. upper right _____
   c. lower left _____ d. lower right _____
10. Write a program to display a solid block of yellow color.

Answers to Odd-Numbered Exercises in Chapter Test

1. a. mode 1C b. mode 2C c. mode 3C d. mode 6C

3. a. mode 3C
   b. mode 6C
   c. mode 1C

5. A.YELLOW B.BLUE C.GREEN D.RED

7. 00101011 or 2B (hex)

9. a. $0400 b. $041F
   c. $0FE0 d. $0FFF

Chapter 5

Animation

In this chapter, we'll build on the knowledge of graphics that you acquired in Chapter 4. You know how to put a simple figure on the screen. Now we'll investigate some ways to simulate motion of a displayed figure.

Two-Color Graphics

You have used the four-color graphic modes. We'll now use some of the two-color modes so that you can see how they differ from four-color graphics. A description of two-color graphic modes is shown in Table 5-1.

Table 5-1. Two-Color Graphic Modes

Resolution	*Graphic Element*	*Memory Used*	*Remarks*
256×192		0400-1BFF	Mode 6R; highest resolution of all; each element is one unit wide and one unit high; 6K of memory used
128×192		0400-0FFF	Mode 3R; each element is two units wide and one unit high; 3K of memory used
128×96		0400-09FF	Mode 2R; each element is two units wide and two units high; 1.5K of memory used
128×64		0400-07FF	Mode 1R; each element is two units wide and three units high; 1K of memory used

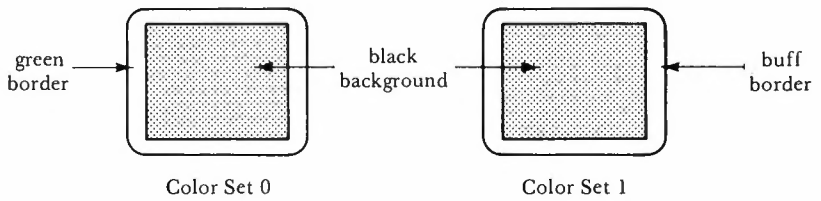
For each of these modes, green and black are the colors used in color set 0.

Color Set 0 { 0 = black
1 = green

For each of these modes, buff and black are used in color set 1.

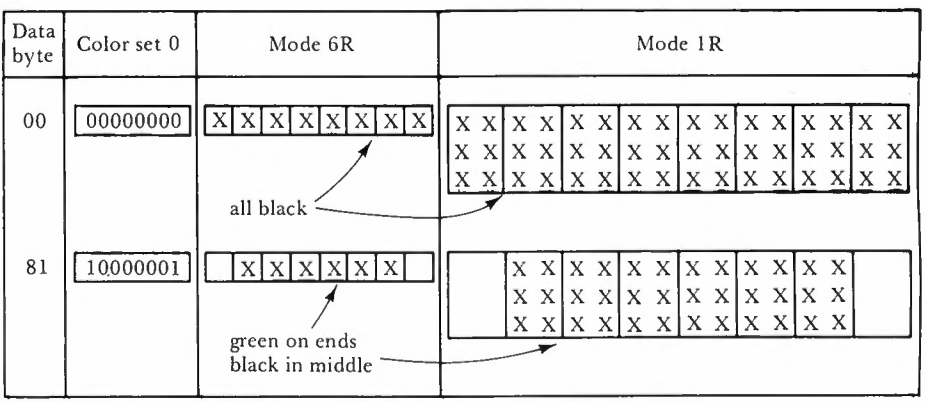
Color Set 1 { 0 = black
1 = buff

An empty screen display for all these modes would be



Each byte of data placed in screen memory will designate the color of eight consecutive graphic elements.

Examples



Of course, you must select the correct values for SAM and VDG as you did for the four-color modes. These instructions to do this are shown in Table 5-2.

Table 5-2. VDG and SAM Instructions for Two-Color Graphics

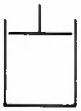
Mode	*Color Set*	*Instructions*
6R	0 green/ black	LDA F0 STA $FF22 STA $FFC5 STA $FFC3
	1 buff/ black	LDA #$F8 STA $FF22 STA $FFC5 STA $FFC3
3R	0 green/ black	LDA #$D0 STA $FF22 STA $FFC5 STA $FFC1
	1 buff/ black	LDA #$D8 STA $FF22 STA $FFC5 STA $FFC1
2R	0 green/ black	LDA #$B0 STA $FF22 STA $FFC3 STA $FFC1
	1 buff/ black	LDA #$B8 STA $FF22 STA $FFC3 STA $FFC1
1R	0 green/ black	LDA #$90 STA $FF22 STA $FFC1
	1 buff/ black	LDA #$98 STA $FF22 STA $FFC1

Because only two colors are used in each of these modes, each bit of a data byte designates the color of one graphic element. Therefore, one byte of data can designate the color of eight adjacent elements.

Using Mode 1R

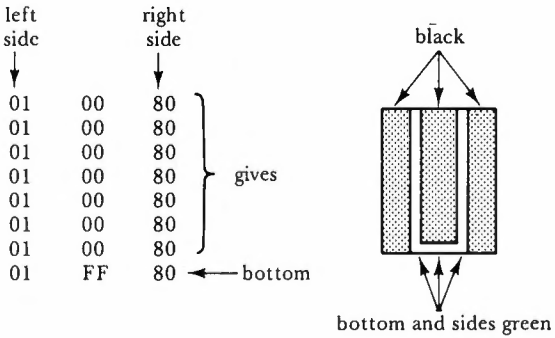
Now that you know how to draw a rectangle, we'll use that as a basis for the next program. We'll draw a rectangle on the video screen that has no top.

This represents a cross section of a cylinder. Inside the cylinder, we'll draw a piston.



Because the figure is simple, we'll use one of the low resolution modes, 1R. Each element will be this size

To plan our data bytes, we made a large sketch on paper as shown in Figure 5-1. To draw the cylinder, which will be stationary, we will use the following data bytes. The black matches the background. Therefore, only the green bottom and sides of the cylinder will appear.



The piston and its shaft will be drawn near the top of the cylinder. In this program, we'll merely blink this part of the figure on (green) and off (black). In Progam 10, we'll make it move up and down inside the cylinder.

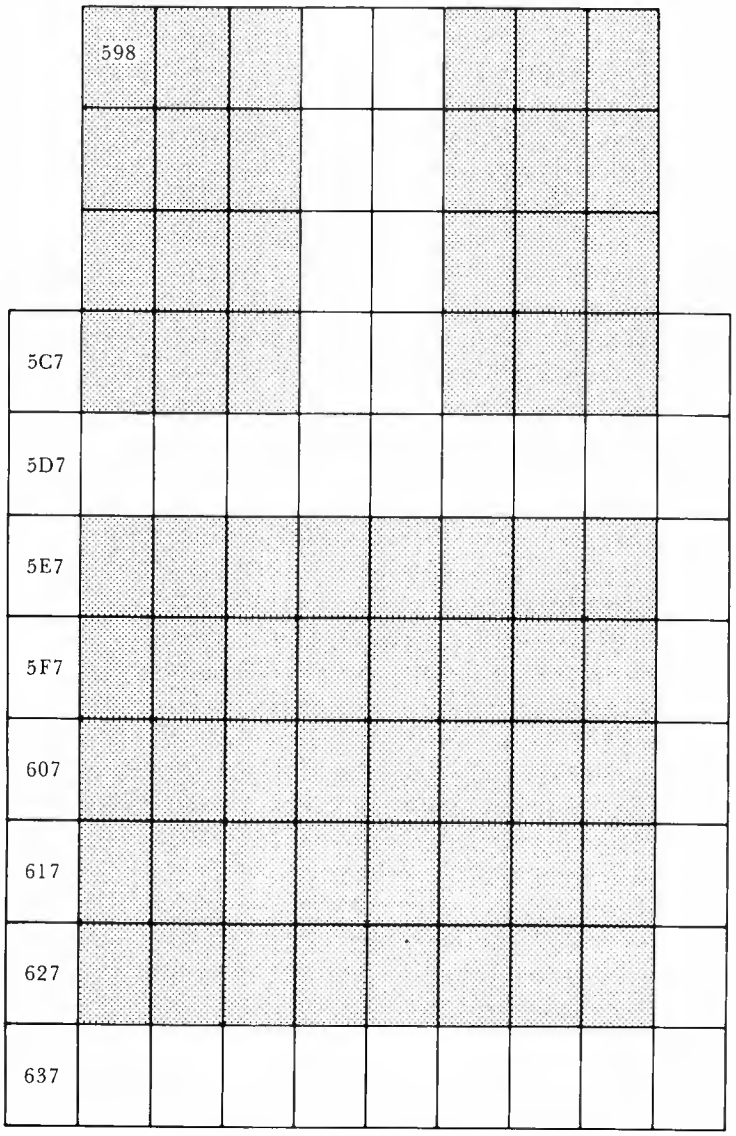
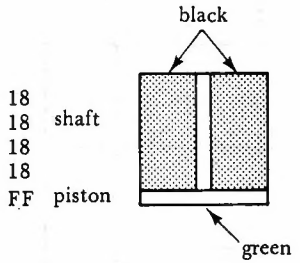
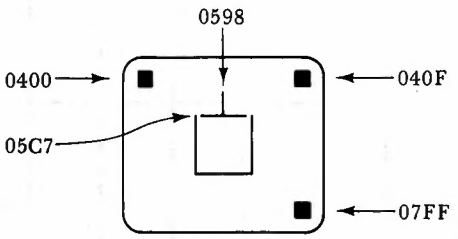


Figure 5-1. Planning the Valve and Cylinder

The following bytes are used for the pistons:



We'll place our figure near the center of the screen.



Parts for Program 9

We'll use many parts that will be similar to previous programs. The correct mode must always be set up, the color set must be selected, the screen must be cleared, the graphics must be drawn, and a way should be provided to stop the program.

Part 1—Set Up

```
        ORG  $1E00
        LDA  #$90
        STA  $FF22
        STA  $FFC1
```

Part 2—Clear Screen

```
        CLRA
        CLRB
        LDX  #$400
LOOP1   STD  ,X++
        CMPX #$1C00
        BLO  LOOP1
```

Part 3—Draw Cylinder

```
        LDX #$5C7
        LDY #TABLE1
DOWN    LDB #3
RIGHT   LDA ,Y+
        STA ,X+
        DECB
        BNE RIGHT
        LEAX $D,X
        CMPX #$647
        BNE DOWN
```

Part 4—Draw Piston

```
DRAW    LDY #TABLE2
        LDX #$598
        LDB #5
PIST    LDA ,Y+
        STA ,X+
        LEAX $F,X
        DECB
        BNE PIST
        JSR DELAY
```

Part 5—Erase Piston

```
        CLRA
        LDX #$598
        LDB #5
ERASE   STA ,X+
        LEAX #F,X
        DECB
        BNE ERASE
        JSR DELAY
```

Part 6—Test for X Key

```
        LDA #$FE
        STA $FF02
        LDA $FF00
        CMPA #$F7
        BNE DRAW
```

Part 7—Go to Text and Monitor

```
TEXT    LDA #$60
        LDX #$00
LOOP2   STA ,X+
        CMPX #$600
        BL0 LOOP2
        CLRA
        STA  $FF22
        STA  $FFC0
        RTS
```

Part 8—Delay Subroutine

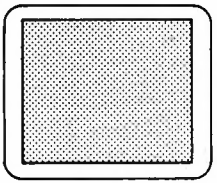
```
DELAY   LDS #$F000
COUNT   NOP
        NOP
        NOP
        DEX
        BNE COUNT
        RTS
```

Part 9—Data Tables

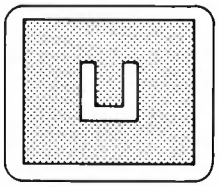
```
TABLE1  FCB 1,0,$80,1,0,$80
        FCB 1,0,$80,1,0,$80
        FCB 1,0,$80,1,0,$80
        FCB 1,0,$80,1,$FF,$80
TABLE2  FCB $18,$18,$18,$18,$FF
        END
```

How Program 9 Works

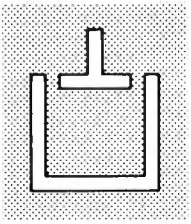
Parts 1 and 2 perform the same function as in previous programs. Graphic mode 1R is selected by storing $90 (see Table 5-2) in locations $FF22 and $FFC1. Part 2 clears the screen to display a green border with a black background.



Part 3 uses the data in Table 1 (see Part 9—Data Tables above) to draw the cylinder. Three bytes are used for each screen line with bytes 1, 0, and $80 (seven times) forming the sides. Data bytes 1, $FF, and $80 form the bottom of the cylinder.

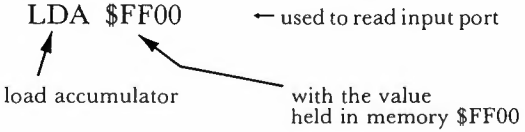


Part 4 draws the piston from the data in Table 2 above. One byte is used for each screen line with $18 (four times) forming the shaft and $FF forming the bottom of the piston. A jump to a time delay subroutine is made so that you will have time to see the figure.



Part 5 erases the piston, and the time delay subroutine is used again.

Part 6 scans the keyboard for the X key. We have written a subroutine to do this so that you can see how the color computer "reads" the keyboard. Figure 5-2 shows how the keyboard matrix is wired. You can see that each key is connected to one bit of Input Port $FF00 and to one bit of the Output Port $FF02. Notice that the X key is connected to bit 3 of the input port and to bit 0 of the output port. The LDA (Load Accumulator A) instruction appears in a new form in this part. Extended addressing is used.



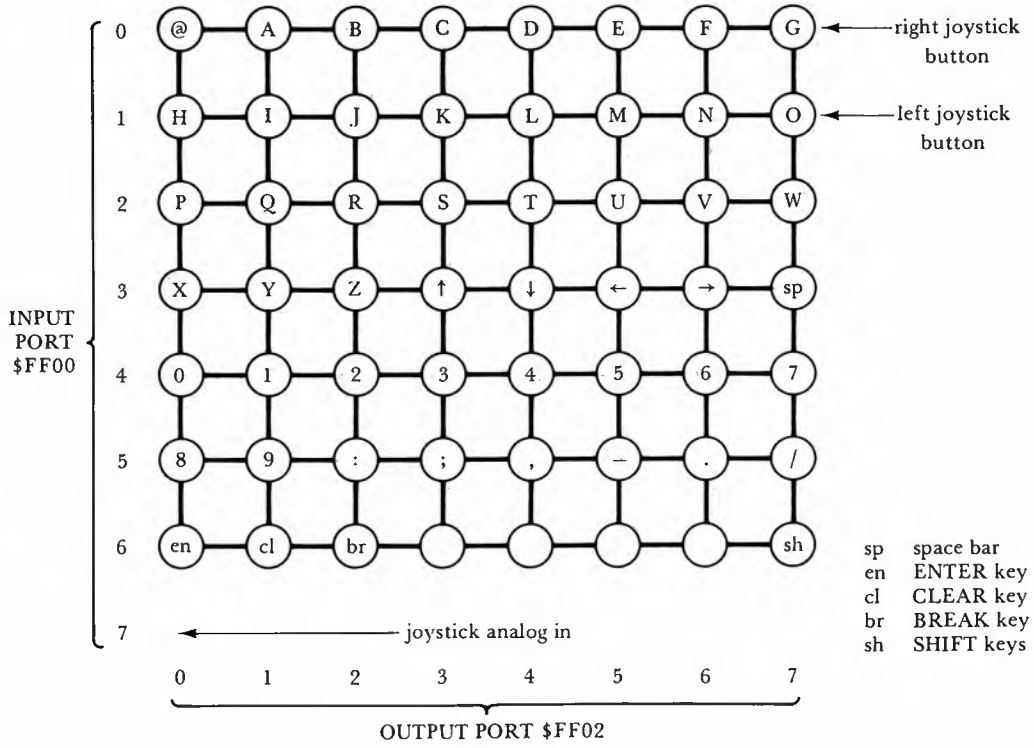


Figure 5-2. Color Computer Keyboard Matrix

To see if the X key is pressed, write a 0 to bit 0 of Output Port $FF02, then read Input Port $FF00. If bit 3 of $FF02 is a 0, the X key is down.

> Keyboard input and output bits are enabled by a 0 and disabled by a 1.

Therefore

```
LDA #$FE      write a 0 to bit 0
STA $FF02     of output port
```

7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0

```
LDA $FF00     read input port
CMPA #$F7     is bit 3 = 0?
```

7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1

```
BNE DRAW      if X key not down, go back to draw piston
```

Other keys could be tested in a similar way.

Part 7 is the same procedure used in previous programs to clear the text memory area and to reset VDG and SAM for the text mode. It ends with a return from subroutine that sends the computer back to the monitor.

Part 8 is a time delay that slows down the graphics so that they are visible. You can shorten the delay by removing the NOPs or by decreasing the value loaded into the X register.

Part 9 contains the data to draw the cylinder (Table 1) and the piston (Table 2). The pseudo-op FCB indicates that the items are single bytes.

When the program is loaded, assembled, and executed, the cylinder appears and the piston is alternately displayed and erased. Try it a few times before going on to Program 10, which will move the piston up and down. Remember to press the X key to stop the program.

Program 10—Moving the Piston

There are several ways that the piston can be made to appear to move up and down. In Program 10, the animation is achieved by simply erasing the piston (as in Program 9) and redrawing it in a new position.

Parts 1, 2, 3, 6, 7, 8, and 9 remain the same as in Program 9.

Parts 4 and 5 are changed and subroutines (parts 8a and 8b) are added for drawing and erasing the piston.

Part 1—Set Up
Part 2—Clear Screen } same as Program 9
Part 3—Draw Cylinder
Part 4—Draw and Erase Piston

```
& 5     DRAW    LDX #$598
                JSR ON      <---- draw piston
                LDX #$598
                JSR OFF     <---- erase piston
                LDX #$5B8
                JSR ON      <---- draw piston
                LDX #$5B8
                JSR OFF     <---- erase piston
                LDX #$5D8
                JSR ON                 :
                LDX #$5D8              :
                JSR OFF              etc.
                LDX #$5B8
                JSR ON
                LDX #$5B8
                JSR OFF
```

Parts 6, 7, and 8 are the same as in Program 9. Part 6 tests for the X key. Part 7 prepares the screen for text before returning to the monitor. Part 8 is the time delay.

We have added parts 8a and 8b, the subroutines that draw and erase the piston.

Part 8a—Draw the Piston

```
ON      LDY #TABLE2
        LDB #5
PIST    LDA ,Y+
        STA, X+
        LEAX $F,X
        DECB
        BNE PIST
        JSR DELAY
        RTS
```

Part 8b—Erase the Piston

```
OFF     CLRA
        LDB #5
ERASE   STA ,X+
        LEAX $F,X
        DECB
        BNE ERASE
        RTS
```

Part 9 is the same data table used in Program 9. A complete listing of source and object programs is given in Figure 5-3.

When Program 10 is executed, the piston is drawn and erased so quickly that the piston appears to move up and down in the cylinder.

```
0001 0600                      ORG $1E00
0002 1E00 8690                 LDA #$90
0003 1E02 B7FF22               STA $FF22
0004 1E05 B7FFC1               STA $FFC1
0005 1E08 4F                   CLRA
0006 1E09 5F                   CLRB
0007 1E0A 8E0400               LDX #$400
0008 1E0D ED81          LOOP1  STD ,X++
0009 1E0F 8C1C00               CMPX #$1C00
0010 1E12 25F9                 BLO LOOP1
0011 1E14 8E05C7               LDX #$5C7
0012 1E17 108E1EA4             LDY #TABLE1
0013 1E1B C603          DOWN   LDB #3
0014 1E1D A6A0          RIGHT  LDA ,Y+
0015 1E1F A780                 STA ,X+
0016 1E21 5A                   DECB
0017 1E22 26F9                 BNE RIGHT
0018 1E24 300D                 LEAX $D,X
0019 1E26 8C0647               CMPX #$647
0020 1E29 26F0                 BNE DOWN
0021 1E2B 8E0598        DRAW   LDX #$598
0022 1E2E BD1E86               JSR ON          ;DRAW PISTON
0023 1E31 8E0598               LDX #$598
0024 1E34 BD1E99               JSR OFF         ;ERASE PISTON
0025 1E37 8E05B8               LDX #$5B8
0026 1E3A BD1E86               JSR ON          ;DRAW PISTON
0027 1E3D 8E05B8               LDX #$5B8
0028 1E40 BD1E99               JSR OFF         ;ERASE PISTON
0029 1E43 8E05D8               LDX #$5D8
0030 1E46 BD1E86               JSR ON          ;    ETC.
0031 1E49 8E05D8               LDX #$5D8
0032 1E4C BD1E99               JSR OFF
0033 1E4F 8E05B8               LDX #$5B8
```

```
0034 1E52 BD1E86              JSR ON
0035 1E55 8E05B8              LDX #$5B8
0036 1E58 BD1E99              JSR OFF
0037 1E5B 86FE                LDA #$FE
0038 1E5D B7FF02              STA $FF02
0039 1E60 B6FF00              STA $FF00
0040 1E63 81F7                CMPA #$F7
0041 1E65 26C4                BNE DRAW
0042 1E67 8660          TEXT  LDA #$60
0043 1E69 8E0400              LDX #$400
0044 1E6C A780          LOOP2 STA ,X+
0045 1E6E 8C0600              CMPX #$600
0046 1E71 25F9                BLO LOOP2
0047 1E73 4F                  CLRA
0048 1E74 B7FF22              STA $FF22
0049 1E77 B7FFC0              STA $FFC0
0050 1E7A 39                  RTS

0051 1E7B 8E4000        DELAY LDX #$4000
0052 1E7E 12            COUNT NOP
0053 1E7F 12                  NOP
0054 1E80 12                  NOP
0055 1E81 301F                DEX
0056 1E83 26F9                BNE COUNT
0057 1E85 39                  RTS
0058 1E86 108E1EBC      ON    LDY #TABLE2
0059 1E8A C605                LDB #5
0060 1E8C A6A0          PIST  LDA ,Y+
0061 1E8E A780                STA ,X+
0062 1E90 300F                LEAX $F,X
0063 1E92 5A                  DECB
0064 1E93 26F7                BNE PIST
0065 1E95 BD1E7B              JSR DELAY
0066 1E98 39                  RTS
0067 1E99 RF            OFF   CLRA
0068 1E9A C605                LDB #5
0069 1E9C A780          ERASE STA ,X+
0070 1E9E 300F                LEAX $F,X
0071 1EA0 5A                  DECB
0072 1EA1 26F9                BNE ERASE
0073 1EA3 39                  RTS
0074 1EA4 0100800100  TABLE1  FCB 1,0,$80,1,0,$80
0075 1EAA 0100800100          FCB 1,0,$80,1,0,$80
0076 1EB0 0100800100          FCB 1,0,$80,1,0,$80
0077 1EB6 01008001FF          FCB 1,0,$80,1,$FF,$80
0078 1EBC 18181818FF  TABLE2  FCB $18,$18,$18,$18,$FF
0079 1EC1                     END

COUNT   1E7E  DELAY   1E7B  DOWN    1E1B  DRAW    1E2B
ERASE   1E9C  LOOP1   1E0D  LOOP2   1E6C  OFF     1E99
ON      1E86  PIST    1E8C  RIGHT   1E1D  TABLE1  1EA4
TABLE2  1EBC  TEXT    1E67
```

Figure 5-3. Listing of Program 10

Paging Screen Memory

All the programs that you have used so far have started the screen memory at address $400. This is the normal location set when the color is first turned on. However, this location can be changed by writing to the display offset registers at locations $FFC6 through $FFD3.

If all the display offset registers are cleared, the display will begin at $0000. The offsets are described in the Radio Shack Service Manual for the Color Computer (Catalog Number 26-3001/3002) as F0, F1, F2, F3, F4, F5, and F6. To set or clear the offsets, write to the appropriate registers, shown in Table 5-3.

Table 5-3. Display Offset Registers

Write to Register	*Sets Offset*	*Clears Offset*	*Offset Added*
FFC6	—	F0	—
FFC7	F0	—	$200
FFC8	—	F1	—
FFC9	F1	—	$400
FFCA	—	F2	—
FFCB	F2	—	$800
FFCC	—	F3	—
FFCD	F3	—	$1000
FFCE	—	F4	—
FFCF	F4	—	$2000
FFD0	—	F5	—
FFD1	F5	—	$4000
FFD2	—	F6	—
FFD3	F6	—	$8000

The values added by the offset registers that are set are summed to form the total offset for the beginning address of the display.

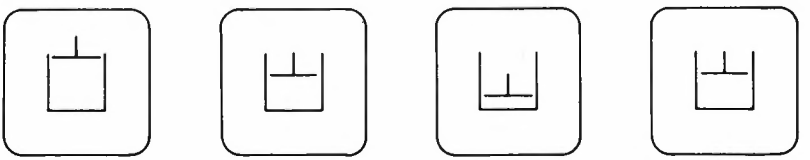
When the computer is turned on, the normal offset is $400. This means that F1 is set and all the other offsets (F0, F2, F3, F4, F5, and F6) are clear. Table 5-4 shows the offset registers that should be set to start the display at any one-half K offset up to $4000. By setting a combination of offset registers, you can page through all of the RAM memory.

Table 5-4. Display Offsets 0-$4000

Starting Display Address	*Write to Address(es)*
0000	none
0200	$FFC7
0400	$FFC9
0600	$FFC7, $FFC9
0800	$FFCB
0A00	$FFCB, $FFC7
0C00	$FFCB, $FFC9
0E00	$FFCB, $FFC9, $FFC7
1000	$FFCD
1200	$FFCD, $FFC7
1400	$FFCD, $FFC9
1600	$FFCD, $FFC9, $FFC7
1800	$FFCD, $FFCB
1A00	$FFCD, $FFCB, $FFC7
1C00	$FFCD, $FFCB, $FFC9
1E00	$FFCD, $FFCB, $FFC9, $FFC7
2000	$FFCF
2200	$FFCF, $FFC7
2400	$FFCF, $FFC9
2600	$FFCF, $FFC9, $FFC7
2800	$FFCF, $FFCB
2A00	$FFCF, $FFCB, $FFC7
2C00	$FFCF, $FFCB, $FFC9
2E00	$FFCF, $FFCB, $FFC9, $FFC7
3000	$FFCF, $FFCD
3200	$FFCF, $FFCD, $FFC7
3400	$FFCF, $FFCD, $FFC9
3600	$FFCF, $FFCD, $FFC9, $FFC7
3800	$FFCF, $FFCD, $FFCB
3A00	$FFCF, $FFCD, $FFCB, $FFC7
3C00	$FFCF, $FFCD, $FFCB, $FFC9
3E00	$FFCF, $FFCD, $FFCB, $FFC9, $FFC7
4000	$FFD0

Program 11—Animation by Paging

Program 11 will produce a display similar to that of Program 10. However, instead of erasing and drawing a new figure each time, the figures will be drawn in different blocks of memory. The offset registers will then be used to display one of these blocks at a time. The blocks will be displayed in order (1, 2, 3, 4), then loop back to repeat (1, 2, 3, 4, 1, 2, 3, 4, 1,...., etc.). We'll use the offsets so that block 1 will be from $600 through $9FF, block 2 from $A00 through $DFF block 3 from $E00 through $11FF, and block 4 from $1200 through $15FF.



The cylinders must be drawn at the same relative position in each block of screen memory.

Part 1—Set Up

```
        ORG  $1E00
ROW     RMB  1
        LDA  #$98          ←color set 1, mode 1R
        STA  $FF22
        STA  $FFC1
```

Part 2—Clear Screen

```
        CLRA
        CLRB
LOOP1   LDX  #$600
        STD  ,X++          ←clear graphics area
        CMPX #$1600        for four blocks of memory
        BLO  LOOP1
```

Part 3—Draw Cylinder in Each Block of Memory

```
        LDX #$706
        JSR CYL         ←1st block
        LDX #$B06
        JSR CYL         ←2nd block
        LDX #$F06
        JSR CYL         ←3rd block
        LDX #$1306
        JSR CYL         ←4th block
```

Part 4—Draw Piston in Each Block of Memory

```
        LDX #$6D7
        JSR PIST        ←1st block
        LDX #$AE7
        JSR PIST        ←2nd block
        LDX #$EF7
        JSR PIST        ←3rd block
        LDX #$1307
        JSR PIST        ←4th block
```

Part 5—Change Display Blocks and Test for X

```
        STA $FFC7       ←turn on block 1
AGIN    JSR DELAY           (FFC7, FFC9 set)
        STA $FFC8       ←turn off block 1
        STA $FFCB       ←turn on block 2
        JSR DELAY           (FFCY, FFCB set)
        STA $FFC9       ←turn on block 3
        JSR DELAY           (FFC7, FFC9, FFCB set)
        STA $FFC8       ←turn off block 3
        STA $FFCA
        STA $FFCD       ←turn on block 4
        JSR DELAY           (FFC7, FFCD set)
        LDA #$FE        ←test for X
        STA $FF02
        LDA $FF00
        CMPA #$F7
        BEQ TEXT
        STA $FFCC       ←turn off block 4
```

```
        STA $FFC9       ←turn on block 1
        BRA AGIN            (FFC7, FFC9 set)
                        ←repeat the cycle
```

Part 6—Go to Text and Monitor

```
TEXT    LDA #$60
        LDX #$400
LOOP2   STA ,X+
        CMPX #$600
        BLO LOOP2
        CLRA
        STA $FF22   }   ←get ready for text mode
        STA $FFC0   }
        STA $FFC6   }   ←adjust offsets
        STA $FFCC   }
        STA $FFC9   }
        RTS
```

Part 7—Draw a Cylinder

```
CYL     LDA #8
        STA ROW
        LDY #TABLE1
DOWN    LDB #3
RIGHT   LDA ,Y+
        STA ,X+
        DECB
        BNE RIGHT
        LEAX $D,X
        DEC ROW
        BNE DOWN
        RTS
```

Part 8—Draw a Piston

```
PIST    LDY #TABLE2
        LDB #5
RPT     LDA ,Y+
        STA ,Y+
        LEAX $F,X
        DECB
        BNE RPT
        RTS
```

Part 9—Delay

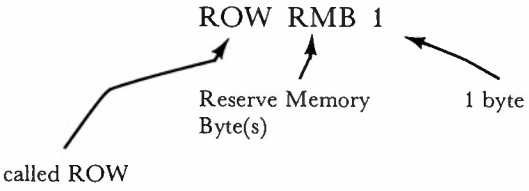
```
DELAY   LDX #$4000
COUNT   NOP
        NOP
        NOP
        DEX
        BNE COUNT
        RTS
```

Part 10—Data Tables

```
TABLE1  FCB 1,0,$80,1,0,$80
        FCB 1,0,$80,1,0,$80
        FCB 1,0,$80,1,0,$80
        FCB 1,0,$80,1,$FF,$80
TABLE2  FCB $18,$18,$18,$18,$FF
        END
```

Description of Program 11

Parts 1 and 2 are similar to previous programs. In part 1, a space (called ROW) is reserved in memory to store the number of rows used to draw the cylinder.



In part 2, we have only cleared that area of memory that will be used for graphics ($600-$1600) in this program.

Part 3 has been changed to call a subroutine to draw the cylinder in each block of memory. The X register is set to the first location of each drawing.

Part 4 draws the piston by using a subroutine. The X register is again used to set the first location of each drawing.

Part 5 is used to access each screenful of memory in succession. A time delay subroutine is used to hold the display briefly for viewing.

The offset for block 1 ($600) requires that offset registers F0 and F1 be set. Because F1 is set normally, you only have to set F0 by writing to $FFC7 (see Tables 5-3 and 5-4). A time delay follows. Block 2 starts at $A00. This requires that F0 and F2 be set. Therefore, you must clear F1 by writing to $FFC8. F0 was set in block 1. F2 is set by writing to $FFCB. A time delay follows. Block 3 starts at $E00. This requires that F0, F1, and F2 be set. You already have F0 and F2 set from block 2. Therefore, set F1 by writing to $FFC9. A time delay follows. The last block begins at $1200. This requires that F0 and F3 be set. To turn off F1 and F2, write to $FFC8 and $FFCA. F3 is set by writing to $FFCD. A time delay follows.

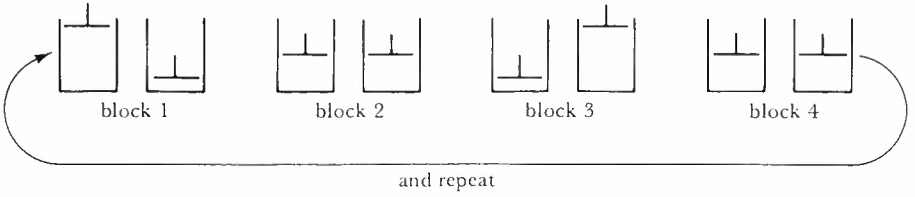
A text is then made to see if the X key has been pressed. If it has, an exit is made to part 6 to prepare for a return to the monitor. If the X key has not been pressed, the computer will go back to display block 1 again. To do this, F3 must be turned off and F1 must be set. Writing to $FFCC turns off F3. Writing to $FFC9 sets F1. Then the blocks are displayed in order until the X key is pressed.

Part 6 prepares for a return to the monitor as before, except some additional store instructions are used to return to the normal text offset at $400.

Parts 7 and 8 draw the cylinders and pistons in much the same way as before.

Part 9 is the same time delay routine and part 10 is made up of the data used to draw the figures.

As we said earlier, Programs 10 and 11 accomplish the same thing, but in different ways. Program 11 is more adaptable to further changes. You might want to try displaying two cylinders at once with each piston in a different position.



If you're really ambitious, you might try to display four cylinders at once. This would require six blocks of memory to show each piston in a different position.

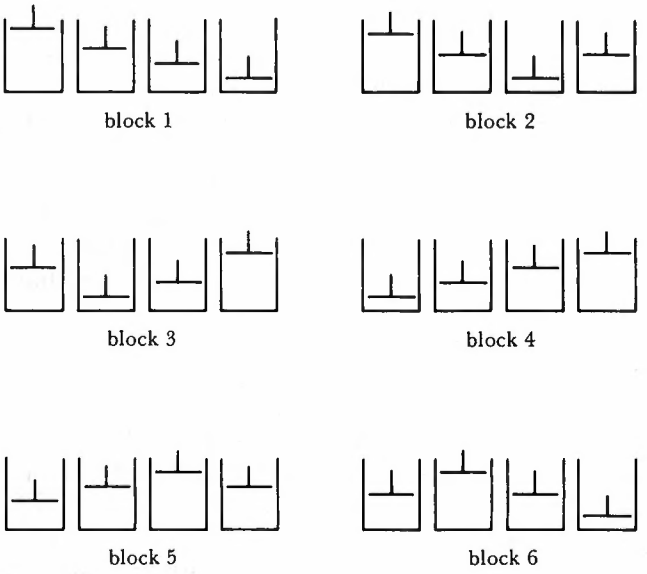


Table 5-5 shows instructions that were first used in this chapter.

Table 5-5. New Instructions in This Chapter

Instruction and Addressing Mode	*Program and Use*
LDA extended	Program 9; accumulator A is loaded with the data in the extended address Example: LDA $FF00
FCB pseudo-op	Program 9; used to indicate single byte items in a data table Example: FCB 1,0,$80,1,0,$80
RMB n	Program 11; used to reserve the specified number (n) of bytes Example: RMB 1

Summary

In this chapter you worked with two-color graphic modes and learned two methods to animate graphic displays.

- Each of the graphic modes discussed uses different amounts of memory and displays different degrees of resolution.

Mode	*Resolution*	*Element*	*Memory Used*
6R	256 × 192		6K
3R	128 × 192		3K
2R	128 × 96		1.5K
1R	128 × 64		1K

- Certain registers must be set to designate a given graphic mode.

Mode	*Set Registers*	*Data in $FF22*
6R	$FFC3, $FFC5	F0 for color set 0 F8 for color set 1
3R	$FFC1, $FFC5	D0 for color set 0 D8 for color set 1
2R	$FFC1, $FFC3	B0 for color set 0 B8 for color set 1
1R	$FFC1	90 for color set 0 98 for color set 1

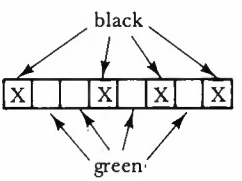
- One data byte sets eight adjacent graphic elements. If a given bit in the data byte is 0, the color of the corresponding element is black. If a given bit is 1, the color of the corresponding element is green in color set 0 or buff in color set 1.

Examples

Color set 1, mode 6R

Data byte = 6A

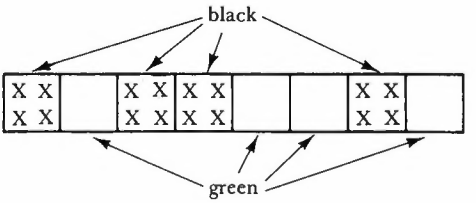
0 1 1 0 1 0 1 0 gives



Color set 1, mode 2R

Data byte = 4B

0 1 0 0 1 1 0 1 gives



- One method used to animate a figure is to draw it, erase it, and redraw it in a new position.
- A second method of animation is to draw the figure in different positions in different blocks of memory. Each different block is displayed by altering the display offset registers. Setting combinations of these registers allows you to set the starting address of the display anywhere in memory.

Offset Register	*Write to Address*		*Offset Added*
	To Clear	*To Set*	
F0	FFC6	FFC7	$200
F1	FFC8	FFC9	$400
F2	FFCA	FFCB	$800
F3	FFCC	FFCD	$1000
F4	FFCE	FFCF	$2000
F5	FFD0	FFD1	$4000
F6	FFD2	FFD3	$8000

- The keyboard may be scanned by writing to Output Port $FF02 and reading Input Port $FF00. Keyboard inputs and outputs are enabled by 0 and disabled by 1.

Chapter Test

1. What two-color graphic mode displays the following size elements?

   a.  _____

   b.  _____

2. How much memory fills the video screen for each of the following two-color graphic modes?

   a. 6R _____

   b. 2R _____

   c. 1R _____

3. What colors are used for the following two-color graphic mode color sets?

   a. color set 0 _____ and _____

   b. color set 1 _____ and _____

4. What modes are selected by storing the following values in $FF22?
   a. $F0 mode _____
   b. $B0 mode _____

5. What graphic mode and color set is set up by the following instructions?

```
LDA #$D8
STA $FF22
STA $FFC5
STA $FFC1
```

   a. graphic mode _____
   b. color set _____

6. Write instructions similar to those in test exercise 5 that would set up mode 2R, color set 0.

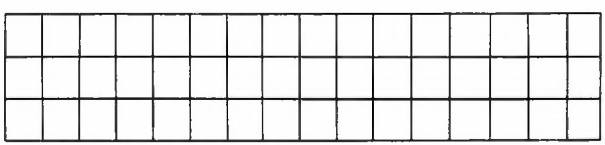
_____
_____
_____
_____

7. A graphic mode and color set are selected as shown. The data byte shown is loaded. Fill in the colors where they would be displayed by using the symbols:

X for black
0 for green
/ for buff
} ← all do not have to be used

```
Set up:     LDA #$90
            STA $FF22
            STA $FFC1

Data Byte:  LDA #$5A
            STA $608
```

$608 →



8. What are the numbers of the keyboard ports?
   a. Input Port _____
   b. Output Port _____
9. Use Figure 5-2 to give the key that is searched for by the following instructions.

```
LDA #$F7
STA $FF02
LDA $FF00
CMPA #$FB
```

   The key is _____
10. Suppose the display offset has been fixed at $400 and you want to change the offset to $900 (choose from: F0, F1, F2, F3, F4, F5, and F6).
    a. What offset registers should be cleared?

    _____

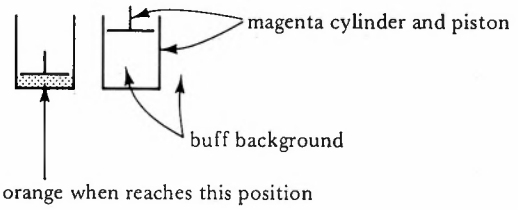
    b. What offset registers should be set?

    _____

11. What does the RMB represent in part 1 of Program 11?

    _____

12. Write a program using a four-color graphic mode to draw the motion of a two-cylinder engine using orange to show the firing at the bottom of each piston stroke.



Answers to Odd-Numbered Exercises in Chapter Test

1. a. 3R b. 1R
3. a. green and black b. buff and black
5. a. graphic mode 3R b. color set 1

7. $608

```
XX00XX0000XX00XX
XX00XX0000XX00XX
XX00XX0000XX00XX
```

9. The key is S.

11. Reserve Memory Bytes(s)

Chapter 6

Sound and Graphics

You can now put figures on the screen and move them around. Earlier, you learned how to make computer sounds. In this chapter, we'll put sound and graphics together. Before we do that however, you should experiment systematically with the Color Computer's sound capabilities.

Experimenting with Sound

To experiment, we have modified a program from *The Facts**, a book that describes the internal components of the Color Computer and how they are used in the system.

Our program allows you to input a two-digit hexadecimal value for the number of cycles used to create the sound and a second two-digit hexadecimal value for a time delay. By experimenting with these two values, you can vary the sound that is produced.

Program 12—Sound Explorer

Equate statements are used to give names to certain ROM routines used. Whenever you use the routines, you can refer to them by their equated names rather than their memory locations. Names are much easier to remember than numerical locations. Equate statements are not a part of the actual program.

* The Facts, Spectral Associates, 141 Harvard Ave., Tacoma, WA 98466

```
POLCAT   EQU $A1B1  keyboard scan routine
PRINIT   EQU $A30A  screen print routine
DA       EQU $FF20  address of digital to analog converter
AUDON    EQU $A976  enable the sound analog multiplexer
SELMUX   EQU $A9A2  connect d/a to multiplexer
WAIT     EQU $A7D3  BASIC's delay routine
```

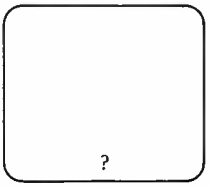
Another statement used is RMB (Reserve Memory Byte). It reserves one or more memory bytes for storage of data. Any reference to the name by which the memory is reserved will use that memory. In our program we will use the following:

```
DELAY RMB 1   (reserve one memory byte called DELAY)
```

The start of the program will be set by the pseudo-operation ORG (ORiGin) as follows:

```
ORG $3000
```

After all the equate statements and pseudo-operations are taken care of, the program begins by prompting you for the number of cycles desired. To do this, the screen is cleared and a question mark appears near the bottom of the screen.



A subroutine called QUEST performs the task of getting your two-digit hex input (01-FF), converting the input's ASCII code to hexadecimal form and storing it in register D. QUEST is used in the same way to acquire the time delay value.

```
JSR QUEST      get cycles
TFR D,Y        put it in Y
JSR QUEST      get delay
TFR B,DELAY    put it in memory (DELAY)
```

Two subroutines, AUDON and SELMUX, are used to enable the sound and select the correct values for the multiplexer.

```
JSR AUDON
CLRB
JSR SELMUX
```

The main part of the program then uses the values for the number of cycles, the delay, and a data table of amplitudes (loaded into accumulator A, one at a time) to produce the sound. Figure 6-1 shows a complete listing of the source and object programs.

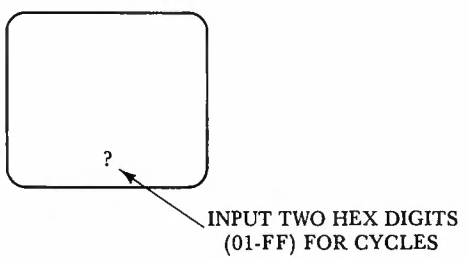
```
0001 0600                POLCAT EQU $A1B1
0002 0600                PRINIT EQU $A30A
0003 0600                DA     EQU $FF20
0004 0600                AUDON  EQU $A976
0005 0600                SELMUX EQU $A9A2
0006 0600                WAIT   EQU $A7D3
0007 0600                DELAY  RMB 1
0008 0601                       ORG $3000
0009 3000 8660           START  LDA #$60
0010 3002 8E0400                LDX #$400
0011 3005 A780           CLEAR  STA ,X+
0012 3007 8C0600                CMPX #$600
0013 300A 25F9                  BLO CLEAR
0014 300C 8660                  LDA #$60
0015 300E B70400                STA $400
0016 3011 BD3044                JSR QUEST
0017 3014 1F02                  TFR D,Y
0018 3016 BD3044                JSR QUEST
0019 3019 F70600                STB DELAY
0020 301C BDA976                JSR AUDON
0021 301F 5F                    CLRB
0022 3020 BDA9A2                JSR SELMUX
0023 3023 338D0052       LOOP1  LEAU TABLE,PCR
0024 3027 A6C0           LOOP   LDA ,U+
0025 3029 2712                  BEQ LOOP2
0026 302B 48                    LSLA
0027 302C 48                    LSLA
0028 302D 8A02                  ORA #2
0029 302F B7FF20                STA DA
0030 3032 4F                    CLRA
0031 3033 F60600                LDB DELAY
0032 3036 1F01                  TFR D,X
0033 3038 BDA7D3                JSR WAIT
0034 303B 20EA                  BRA LOOP
0035 303D 313F           LOOP2  LEAY -1,Y
0036 303F 26E2                  BNE LOOP1
0037 3041 20BD                  BRA START
0038 3043 3F             QUIT   SWI

0039 3044 863F           QUEST  LDA #'?
0040 3046 BDA30A                JSR PRINIT
0041 3049 BDA1B1                JSR POLCAT
0042 304C 8123                  CMPA #'#
```

```
0043 304E 27F3                BEQ QUIT
0044 3050 BDA30A              JSR PRINIT
0045 3053 8140                CMPA #$40
0046 3055 2502                BLO SHIFT
0047 3057 8B09                ADDA #9
0048 3059 48          SHIFT   LSLA
0049 305A 48                  LSLA
0050 305B 48                  LSLA
0051 305C 48                  LSLA
0052 305D 1F89                TFR A,B
0053 305F BDA1B1              JSR POLCAT
0054 3062 8123                CMPA #'#
0055 3064 27DD                BEQ QUIT
0056 3066 BDA30A              JSR PRINIT
0057 3069 8140                CMPA #$40
0058 306B 2502                BLO MSK
0059 306D 8B09                ADDA #9
0060 306F 840F        MSK     ANDA ##F
0061 3071 3404ABE0            ABA
0062 3075 1F89                TFR A,B
0063 3077 4F                  CLRA
0064 3078 39                  RTS
0065 3079 0101        TABLE   FCB 1,1
0066 307B 020406              FCB 2,4,6
0067 307E 090D11              FCB 9,13,17
0068 3081 161C23              FCB 22,28,35
0069 3084 2C3F00              FCB 44,63,0
0070 3087                     END

AUDON  A976  CLEAR  3005  DA     FF20  DELAY  0600
LOOP   3027  LOOP1  3023  LOOP2  303D  MSK    306F
POLCAT A1B1  PRINIT A30A  QUEST  3044  QUIT   3043
SELMUX A9A2  SHIFT  3059  START  3000  TABLE  3079
WAIT   A7D3
```

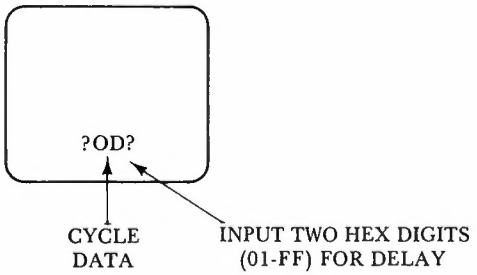
Figure 6-1. Sound Explorer

Using the Sound Explorer

As you can see from the listing, the program begins at memory location $3000. The first question mark prompts you for the number of



cycles. Remember, a two-digit hex input is required (01-FF). Your input will appear following the first question mark. A second question mark then appears. This is prompting you to input the time delay value. Again, this should be a two-digit hexadecimal value (01-FF).



The sound will then be made. Following the sound, the screen will clear, and the process is repeated for a new sound.

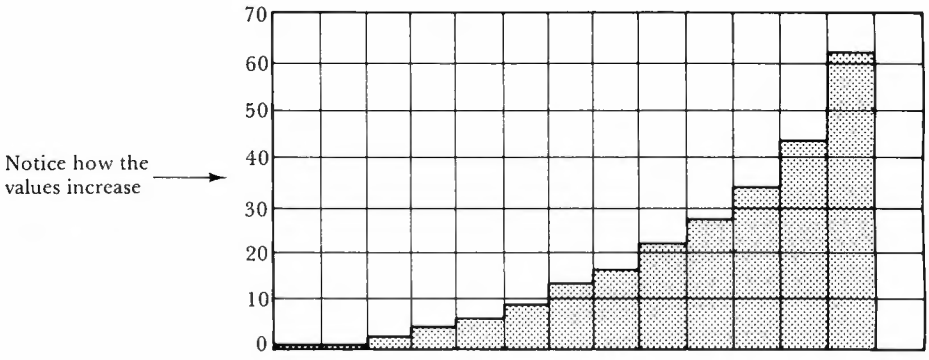
Table 6-1 gives some suggested values to input. Try those and then make up some of your own.

Table 6-1. Experimental Sound Values

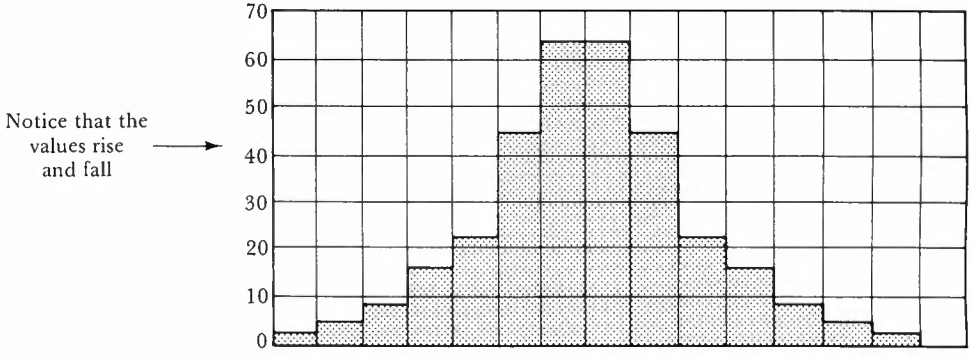
Cycles	*Delay*	*Results*
05	05	a short "pip"
05	10	lower "pip"
05	20	
05	40	
05	60	
05	90	
05	C0	
05	F0	very low
10	40	longer sound with 40 tone
20	40	
40	40	
60	40	
90	40	
C0	40	
F0	40	very long

for you to try →

Other variations can be produced by changing the amplitude values in TABLE. Here is the one used in the program.



Here is one variation that you might try.



Try some other variations of your own.

Adding Sound to Programs

Now, we'll use the sound routine with some very simple graphics. In this program, sound will be added to a variation of the Color Bar program that you ran in Chapter 3.

Program 13—Sound Bars

That program drew three red lines to form a color bar near the center of the screen. After the red bar is displayed, we'll add a tone. The bar will then be drawn using blue, and a higher tone will be made. Finally, the bar will be drawn in yellow, and a still higher tone will be made.

Two subroutines form the backbone of the program. DISPLA draws the color bars, and SOUND plays the tones. The color, number of cycles, and delay are selected from a table called COLOR. The values for the sound amplitudes are selected from the table called TABLE. Both the COLOR table and the TABLE table may be changed as you desire. We have selected our values for demonstration purposes only. A listing of the Sound Bars program is shown in Figure 6-2.

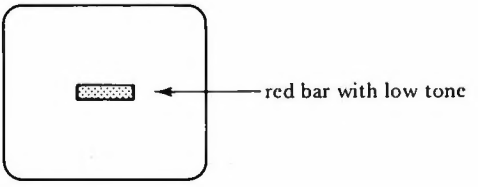
```
0001 0600                DA      EQU $FF20
0002 0600                AUDON   EQU $A976
0003 0600                SELMUX  EQU $A9A2
0004 0600                WAIT    EQU $A7D3
0005 0600                DELAY   RMB 1
0006 0601                CYCLE   RMB 1
0007 0602                        ORG $1E00
0008 1E00 86E0           START   LDA #$E0
0009 1E02 B7FF22                 STA $FF22
0010 1E05 B7FFC3                 STA $FFC3
0011 1E08 B7FFC5                 STA $FFC5
0012 1E0B B7FFC7                 STA $FFC7
0013 1E0E 4F                     CLRA
0014 1E0F 5F                     CLRB
0015 1E10 8E0600                 LDX #$600
0016 1E13 ED81           NUM1    STD ,X++
0017 1E15 8C1E00                 CMPX #$1E00
0018 1E18 25F9                   BLO NUM1
0019 1E1A 338D004A               LEAU COLOR,PCR
0020 1E1E 8E1208         NUM2    LDX #$1208
0021 1E21 BD1E5C                 JSR DISPLA
0022 1E24 8E1228                 LDX #$1228
0023 1E27 BD1E5C                 JSR DISPLA
0024 1E2A 8E1248                 LDX #$1248
0025 1E2D BD1E5C                 JSR DISPLA
0026 1E30 3341                   LEAU 1,U
0027 1E32 A6C0                   LDA ,U+
0028 1E34 B70600                 STA DELAY
0029 1E37 A6C0                   LDA ,U+
0030 1E39 B70601                 STA CYCLE
0031 1E3C BD1E74                 JSR SOUND
0032 1E3F 20DD                   BRA NUM2
```

```
0033 1E41 8660          QUIT    LDA #$60
0034 1E43 8E0400                LDX #$400
0035 1E46 A780          NUM3    STA ,X+
0036 1E48 8C0600                CMPX #$600
0037 1E4B 25F9                  BLO NUM3
0038 1E4D 8600                  LDA #0
0039 1E4F B7FF22                STA $FF22
0040 1E52 B7FFC2                STA $FFC2
0041 1E55 B7FFC4                STA $FFC4
0042 1E58 B7FFC6                STA $FFC6
0043 1E5B 3F                    SWI
0044 1E5C C608          DISPLA  LDB #8
0045 1E5E A6C4          GET     LDA ,U
0046 1E60 27DF                  BEQ QUIT
0047 1E62 A780                  STA ,X+
0048 1E64 5A                    DECB
0049 1E65 26F7                  BNE GET
0050 1E67 39                    RTS

0051 1E68 FF5020        COLOR   FCB $FF,$50,$20
0052 1E6B AA3040                FCB $AA,$30,$40
0053 1E6E 551580                FCB $55,$15,$80
0054 1E71 000000                FCB 0,0,0
0055 1E74 3456          SOUND   PSHS U,X,D
0056 1E76 BDA976                JSR AUDON
0057 1E79 5F                    CLRB
0058 1E7A BDA9A2                JSR SELMUX
0059 1E7D 338D001E      LOOP1   LEAU TABLE,PCR
0060 1E81 A6C0          LOOP    LDA ,U+
0061 1E83 2712                  BEQ LOOP2
0062 1E85 48                    LSLA
0063 1E86 48                    LSLA
0064 1E87 8A02                  ORA #2
0065 1E89 B7FF20                STA DA
0066 1E8C 4F                    CLRA
0067 1E8D F60600                LDB DELAY
0068 1E90 1F01                  TFR D,X
0069 1E92 BDA7D3                JSR WAIT
0070 1E95 20EA                  BRA LOOP
0071 1E97 7A0601        LOOP2   DEC CYCLE
0072 1E9A 26E1                  BNE LOOP1
0073 1E9C 3556                  PULS U,X,D
0074 1E9E 39                    RTS
0075 1E9F 010102        TABLE   FCB 1,1,2
0076 1EA2 040609                FCB 4,6,9
0077 1EA5 0D1116                FCB 13,17,22
0078 1EA8 1C232C                FCB 28,35,44
0079 1EAB 3F00-                 FCB 63,0
0080 1EAD                       END

AUDON   A976    COLOR   1E68    CYCLE   0601    DA      FF20
DELAY   0600    DISPLA  1E5C    GET     1E5E    LOOP    1E81
LOOP1   1E7D    LOOP2   1E97    NUM1    1E13    NUM2    1E1E
NUM3    1E46    QUIT    1E41    SELMUX  A9A2    SOUND   1E74
START   1E00    TABLE   1E9F    WAIT    A7D3
```

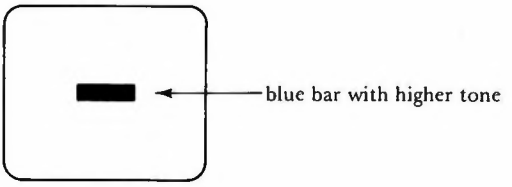
Figure 6-2. Sound Bars

When the program is run, the following sequence of pictures and sounds are produced.

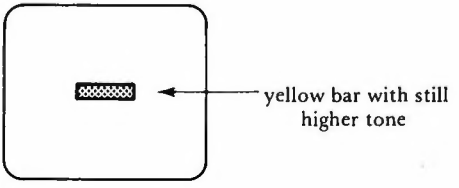
1.



2.



3.



The color for the bar and the delay and cycles for the sound are obtained from the table labeled COLOR. Typical values from the table are as follows:

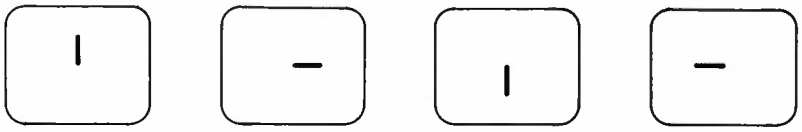
```
COLOR   FCB $FF,$50,$20
```

red color delay cycles

The bars are drawn by the DISPLA subroutine using the amplitude values from TABLE.

Program 14—Rotating Bar

In this program, we'll make a color bar rotate through four different positions. As the bar rotates to a new position, a sound will be made.



The color used for the first rotation is red, the second is blue, and the third is yellow.

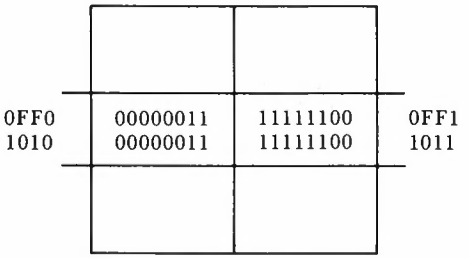
A small area near the center of the screen is used in graphics mode 6C. The memory locations used are:

0F30	0F31
0F50	0F51
0F70	0F71
0F90	0F91
0FB0	0FB1
0FD0	0FD1
0FF0	0FF1
1010	1011
1030	1031
1050	1051
1070	1071
1090	1091
10B0	10B1
10D0	10D1

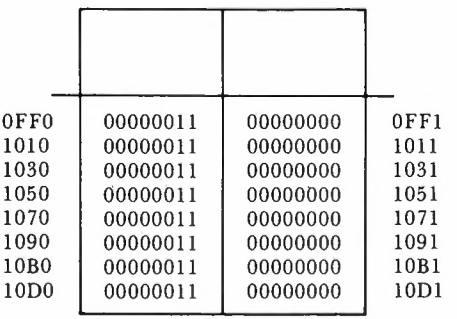
The first position is colored red by loading the data $0300 into the screen area: $0F30,$0F31; $0F50,$0F51; $0F70,$0F71; $0F90, $0F91; $0FB0,$0FB1; $0FD0,$0FD1; $0FF0, $0FF1; and $1010, 1011.

	0 *3*	*0* *0*	← Hex
0F30	00000011	00000000	} binary
0F50	00000011	00000000	
0F70	00000011	00000000	
0F90	00000011	00000000	
0FB0	00000011	00000000	
0FD0	00000011	00000000	
0FF0	00000011	00000000	
1010	00000011	00000000	

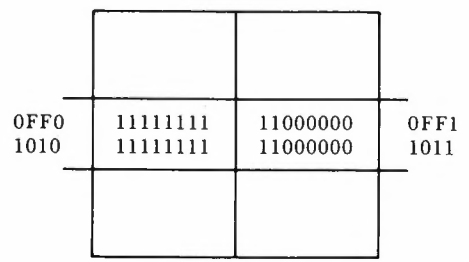
The second position is colored red by loading the data $03 into the screen locations $0FF0 and $1010 and the data $FC into screen locations $0FF1 and $1011.



The third position is drawn in a similar way, but the data ($0300) is loaded into screen locations $0FF0,$0FF1; $1010,$1011; $1030, $1031; $1050, $1051; $1070,$1071; $1090,$1091; $10B0,$10B1; and $10D0,$10D1.



The fourth, and last, position is displayed by loading $FF into $0FF0 and $1010 and loading $C0 into $0FF1 and $1011.



The color data is loaded from TABLE1 of the program. TABLE2 contains the screen addresses into which the color data is stored. The color data used for the first revolution (just described) displays a red

figure. The program changes the color to blue for the second revolution, and yellow for the third revolution.

After three revolutions, the program returns to the monitor. You could have the pattern repeat over and over again by replacing the SWI instruction to a branch always to the section marked GO, as:

```
BRA GO
```

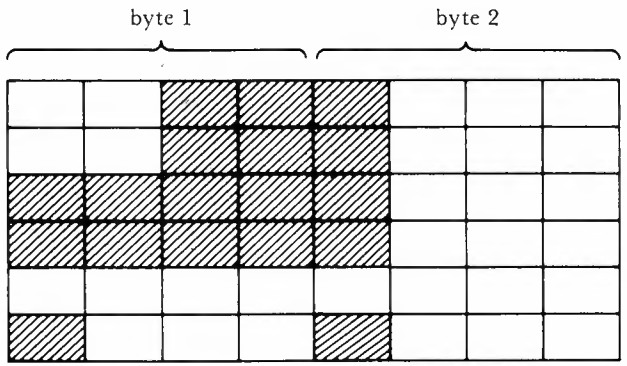
The source and object programs are shown in Figure 6-3.

```
0001 0600                    DA      EQU $FF20
0002 0600                    AUDON   EQU $A976
0003 0600                    SELMUX  EQU $A9A2
0004 0600                    WAIT    EQU $A7D3
0005 0600                    CYCLE   RMB 1
0006 0601                    COUNT   RMB 1
0007 0602                    INDEX   RMB 1
0008 0603                            ORG $2000
0009 2000 86E0               START   LDA #$E0
0010 2002 B7FF22                     STA $FF22
0011 2005 B7FFC3                     STA $FFC3
0012 2008 B7FFC5                     STA $FFC5
0013 200B B7FFC7                     STA $FFC7
0014 200E 4F                         CLRA
0015 200F 5F                         CLRB
0016 2010 8E0600                     LDX #$600
0017 2013 ED81               NUM1    STD ,X++
0018 2015 8C1E00                     CMPX #$1E00
0019 2018 25F9                       BLO NUM1
0020 201A 8603               GO      LDA #3
0021 201C B70602                     STA INDEX
0022 201F 8E20BE                     LDX #TABLE1
0023 2022 108E20D6           ROUND   LDY #TABLE2
0024 2026 8608                       LDA #8
0025 2028 B70601                     STA COUNT
0026 202B BD20B4                     JSR DRAW
0027 202E BD2071                     JSR SOUND
0028 2031 BD20A0                     JSR CLEAR
0029 2034 8602                       LDA #2
0030 2036 B70601                     STA COUNT
0031 2039 BD20B4                     JSR DRAW
0032 203C BD2071                     JSR SOUND
0033 203F BD20A0                     JSR CLEAR
0034 2042 8608                       LDA #8
0035 2044 B70601                     STA COUNT
0036 2047 BD20B4                     JSR DRAW
0037 204A BD2071                     JSR SOUND
0038 204D BD20A0                     JSR CLEAR
0039 2050 8602                       LDA #2
0040 2052 B70601                     STA COUNT
```

```
0041 2055 BD20B4              JSR DRAW
0042 2058 BD2071              JSR SOUND
0043 205B BD20A0              JSR CLEAR
0044 205E 7A0602              DEC INDEX
0045 2061 26BF                BNE ROUND
0046 2063 4F                  CLRA
0047 2064 B7FF22              STA $FF22
0048 2067 B7FFC2              STA $FFC2
0049 206A B7FFC4              STA $FFC4
0050 206D B7FFC6              STA $FFC6
0051 2070 3F                  SWI
0052 2071 3476          SOUND PSHS U,X,Y,D
0053 2073 8640                LDA #$40
0054 2075 B70600              STA CYCLE
0055 2078 BDA976              JSR AUDON
0056 207B 5F                  CLRB
0057 207C BDA9A2              JSR SELMUX
0058 207F 338D007B      LOOP1 LEAU TABLE3,PCR
0059 2083 A6C0          LOOP  LDA ,U+
0060 2085 2711                BEQ LOOP2
0061 2087 48                  LSLA
0062 2088 48                  LSLA
0063 2089 8A02                ORA #2
0064 208B B7FF20              STA DA
0065 208E 4F                  CLRA
0066 208F C610                LDB #$10
0067 2091 1F01                TFR D,X
0068 2093 BDA7D3              JSR WAIT
0069 2096 20EB                BRA LOOP
0070 2098 7A0600        LOOP2 DEC CYCLE
0071 209B 26E2                BNE LOOP1
0072 209D 3576                PULS U,X,Y,D
0073 209F 39                  RTS
0074 20A0 3410          CLEAR PSHS X
0075 20A2 8E0F30              LDX #$0F30
0076 20A5 4F                  CLRA
0077 20A6 5F                  CLRB
0078 20A7 ED84          LOOP3 STD ,X
0079 20A9 308820              LEAX $20,X
0080 20AC 8C10E0              CMPX #$10E0
0081 20AF 25F6                BLO LOOP3
0082 20B1 3510                PULS X
0083 20B3 39                  RTS
0084 20B4 EC81          DRAW  LDD ,X++
0085 20B6 EDB1          LOOP4 STD [,Y++]
0086 20B8 7A0601              DEC COUNT
0087 20BB 26F9                BNE LOOP4
0088 20BD 39                  RTS
0089 20BE 030003FC03    TABLE1 FCB $3,0,3,$FC,3,0,$FF,0
0090 20C6 020002AB02           FCB 2,0,2,$AB,2,0,$AA,0
0091 20CE 0100015401           FCB 1,0,1,$54,1,0,$55,0
```

```
0092 20D6 0F300F500F   TABLE2 FCB  $F,$30,$F,$50,$F,$70
0093 20DC 0F900FB00F           FCB  $F,$90,$F,$B0,$F,$D0
0094 20E2 0FF01010             FCB  $F,$F0,$10,$10
0095 20E6 0FF01010             FCB  $F,$F0,$10,$10
0096 20EA 0FF0101010           FCB  $F,$F0,$10,$10,$10
0097 20EF 3010501070           FCB  $30,$10,$50,$10,$70
0098 20F4 109010B010           FCB  $10,$90,$10,$B0,$10
0099 20F9 D00FF01010           FCB  $D0,$F,$F0,$10,$10
0100 20FE 0101         TABLE3  FCB  1,1
0101 2100 020406               FCB  2,4,6
0102 2103 090D11               FCB  9,13,17
0103 2106 161C23               FCB  22,28,35
0104 2109 2C3F00               FCB  44,63,0
0105 210C                      END

AUDON   A976   CLEAR  20A0   COUNT  0601   CYCLE  0600
DA      FF20   DRAW   20B4   GO     201A   INDEX  0602
LOOP    2083   LOOP1  207F   LOOP2  2098   LOOP3  20A7
LOOP4   20B6   NUM1   2013   ROUND  2022   SELMUX A9A2
SOUND   2071   START  2000   TABLE1 20BE   TABLE2 20D6
TABLE3  20FE   WAIT   A7D3
```

Figure 6-3. Program 14—Rotating Bar



00	11	10	01
green	red	blue	yellow

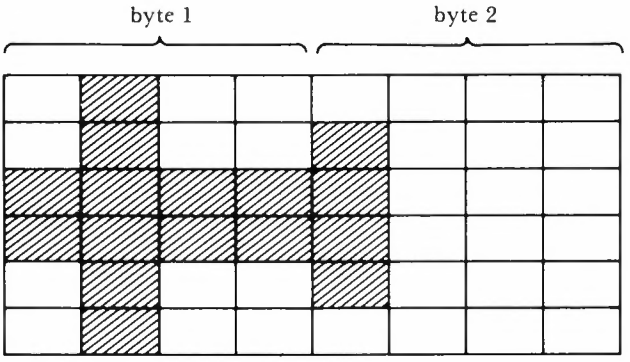
red	blue	yellow
0F C0	0A 80	05 40
0F C0	0A 80	05 40
FF C0	AA 80	55 40
FF C0	AA 80	55 40
00 00	00 00	00 00
C0 C0	80 80	40 40

Figure 6-4. Data Bytes Defining Trucks

Animation with Sound

Program 14 mixed the drawing of colored bars with sound and combined some animation. The next program increases the complexity of the motions. We'll draw trucks and airplanes and add sound as the objects move across the screen.

The high-resolution color mode 6C is used. The trucks are defined by 12 bytes of data as shown in Figure 6-4. We will use three different colors (red, blue, and yellow). The airplanes are defined in a similar way as shown in Figure 6-5.



00 green	11 red	10 bluc	01 yellow

red	bluc	yellow
30 00	20 00	10 00
30 C0	20 80	10 40
FF C0	AA 80	55 40
FF C0	AA 80	55 40
30 C0	20 80	10 40
30 00	20 00	10 00

Figure 6-5. Data Bytes Defining Airplanes

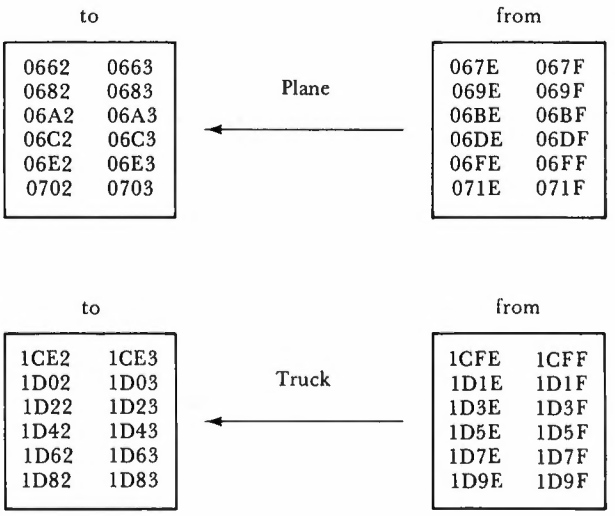
Between each movement of the figures (from right to left) across the screen, a sound will be made to simulate the noise of the motors. Then the screen is erased before the next movement of the figures.

```
0001 0600                DA      EQU $FF20
0002 0600                AUDON   EQU $A976
0003 0600                SELMUX  EQU $A9A2
0004 0600                WAIT    EQU $A7D3
0005 0600                        ORG $2000

0006 2000 86E0           START   LDA #$E0
0007 2002 B7FF22                 STA $FF22
0008 2005 B7FFC3                 STA $FFC3
0009 2008 B7FFC5                 STA $FFC5
0010 200B B7FFC7                 STA $FFC7
0011 200E BD203C                 JSR ERASE
0012 2011 8603           SET     LDA #3
0013 2013 B720A6                 STA COLOR
0014 2016 108E20AA               LDY #TABLE
0015 201A 8620           GO      LDA #$20
0016 201C B720A9                 STA INDEX
0017 201F 8E1CFE                 LDX #$1CFE
0018 2022 BD204D                 JSR MOVE
0019 2025 31A818                 LEAY $18,Y
0020 2028 7A20A6                 DEC COLOR
0021 202B 26ED                   BNE GO
0022 202D 8600           QUIT    LDA #$0
0023 202F B7FF22                 STA $FF22
0024 2032 B7FFC2                 STA $FFC2
0025 2035 B7FFC4                 STA $FFC4
0026 2038 B7FFC6                 STA $FFC6
0027 203B 3F                     SWI
0028 203C 3410           ERASE   PSHS X
0029 203E 4F                     CLRA
0030 203F 5F                     CLRB
0031 2040 8E0600                 LDX #$600
0032 2043 ED81           NUM1    STD ,X++
0033 2045 8C1E00                 CMPX #$1E00
0034 2048 25F9                   BLO NUM1
0035 204A 3510                   PULS X
0036 204C 39                     RTS
0037 204D 8606           MOVE    LDA #6
0038 204F B720A7                 STA COUNT
0039 2052 BD205E                 JSR DRAW
0040 2055 BD203C                 JSR ERASE
0041 2058 7A20A9                 DEC INDEX
0042 205B 26F0                   BNE MOVE
0043 205D 39                     RTS
0044 205E ECA1           DRAW    LDD ,Y++
0045 2060 ED84                   STD ,X
0046 2062 ECA1                   LDD ,Y++
0047 2064 ED89E980               STD -$1680,X
0048 2068 308820                 LEAX $20,X
0049 206B 7A20A7                 DEC COUNT
0050 206E 26EE                   BNE DRAW
0051 2070 31A8E8                 LEAY -$18,Y
0052 2073 3089FF3F               LEAX -$C1,X
0053 2077 3476           SOUND   PSHS U,X,Y,D
0054 2079 8603                   LDA #3
0055 207B B720A8                 STA CYCLE
```

```
0056 207E BDA976              JSR AUDON
0057 2081 5F                  CLRB
0058 2082 BDA9A2              JSR SELMUX
0059 2085 338D0069    LOOP1   LEAU TABLE2,PCR
0060 2089 A6C0        LOOP    LDA ,U+
0061 208B 2711                BEQ LOOP2
0062 208D 48                  LSLA
0063 208E 48                  LSLA
0064 208F 8A02                ORA #2
0065 2091 B7FF20              STA DA
0066 2094 4F                  CLRA
0067 2095 C640                LDB #$40
0068 2097 1F01                TFR D,X
0069 2099 BDA7D3              JSR WAIT
0070 209C 20EB                BRA LOOP
0071 209E 7A20A8      LOOP2   DEC CYCLE
0072 20A1 26E2                BNE LOOP1
0073 20A3 3576                PULS U,X,Y,D
0074 20A5 39                  RTS
0075 20A6             COLOR   RMB 1
0076 20A7             COUNT   RMB 1
0077 20A8             CYCLE   RMB 1
0078 20A9             INDEX   RMB 1
0079 20AA 0FC02000    TABLE   FCB $F,$C0,$20,0
0080 20AE 0FC02080            FCB $F,$C0,$20,$80
0081 20B2 FFC0AA80            FCB $FF,$C0,$AA,$80
0082 20B6 FFC0AA80            FCB $FF,$C0,$AA,$80
0083 20BA 00002080            FCB 0,0,$20,$80
0084 20BE C0C02000            FCB $C0,$C0,$20,0
0085 20C2 0A801000            FCB $A,$80,$10,0
0086 20C6 0A801040            FCB $A,$80,$10,$40
0087 20CA AA805540            FCB $AA,$80,$55,$40
0088 20CE AA805540            FCB $AA,$80,$55,$40
0089 20D2 00001040            FCB 0,0,$10,$40
0090 20D6 80801000            FCB $80,$80,$10,0
0091 20DA 05403000            FCB 5,$40,$30,0
0092 20DE 054030C0            FCB 5,$40,$30,$C0
0093 20E2 5540FFC0            FCB $55,$40,$FF,$C0
0094 20E6 5540FFC0            FCB $55,$40,$FF,$C0
0095 20EA 000030C0            FCB 0,0,$30,$C0
0096 20EE 40403000            FCB $40,$40,$30,0
0097 20F2 0101        TABLE2  FCB 1,1
0098 20F4 020406              FCB 2,4,6
0099 20F7 090D11              FCB 9,13,17
0100 20FA 161C23              FCB 22,28,35
0101 20FD 2C3F00              FCB 44,63,0
0102 2100                     END

AUDON   A976  COLOR  20A6  COUNT  20A7  CYCLE  20A8
DA      FF20  DRAW   205E  ERASE  203C  GO     201A
INDEX   20A9  LOOP   2089  LOOP1  2085  LOOP2  209E
MOVE    204D  NUM1   2043  QUIT   202D  SELMUX A9A2
SET     2011  SOUND  2077  START  2000  TABLE  20AA
TABLE2  20F2  WAIT'  A7D3
```

Figure 6-6. Truck 'n Plane

In Program 15 (Figure 6-6), the MOVE and DRAW routines are used to move a truck and plane from right to left across the screen. The truck moves along the bottom of the screen, and the plane moves along the top of the screen.



The top two lines of the DRAW section move the truck, one line at a time.

```
LDD  ,Y++      ←get shape for line from table
STD  X         ←put it in location stored in X
```

The next two lines move the plane, one line at a time.

```
LDD  ,Y++      ←get shape for line from table
STD  -$1680    ←put in -1680 locations from truck
```

The following line then increases the value in X to prepare for a new line of the figures.

```
LEAX $20,X     ←increase count in X by $20
```

After the figures are drawn, the sound routine is used to simulate the engines.

Feel free to experiment with the parameters used in the program. You can probably come up with a more realistic sound for the engines than the ones we have used.

General Information on Programs in this Chapter

We have used color mode 6C in the programs in this chapter to provide high-resolution four-color graphics with color set 0 (green, yellow, blue, and red). Other modes or colors may be used by modifying a few program lines to select the desirable parameters as described in Chapters 4 and 5.

The necessary parameters for mode 6C, color set 0, are as follows:

```
LDA #$E0      ←color set 0, mode 6C
STA $FF22
STA $FFC3
STA $FFC5
```

We also provided for starting the graphics at $600, which is above the normal text area, by writing to location $FFC7.

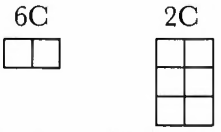
```
STA $FFC7     ←adds $200 offset to the normal starting point
                ($400) of the text screen
```

When you want to return to a text screen from graphics, you must "turn off" the registers that were set for the graphics mode. In our examples for mode 6C, this was done by

```
LDA #0    }
STA $FF22 }  ←turns off previous $E0
STA $FFC2     ←turns off effect of STA $FFC3
STA $FFC4     ←        "            $FFC5
STA $FFC6     ←        "            $FFC7
```

The data necessary to draw graphics will also change for each graphics mode. Chapter 4 explained the color patterns produced by given data bytes. As an example, here are the necessary changes to the Sound Bars program to convert it to mode 2C, color set 1.

```
START   LDA #$A8    }
        STA $FF22   }  <-- setting up the mode
        STA $FFC3   }      and color set
        STA $FFC7      <-- add $200 offset
        :
NUM2    LDX #$1208
        JSR DISPLA
        LDX #$1228  }
        JSR DISPLA  }  <-- OMIT these lines since
        LDX #$1248  }      mode 2C draws 3 times as
        JSR DISPLA  }      mode 6C
        :
NUM3    :
        STA $FF22      <-- turn off graphics parameters
        STA $FFC2
        STA $FFC6
```



Try these changes to Sound Bars and note the difference in the colors. They should now be orange, magenta, and cyan on a buff background.

Summary

In this chapter you learned how to make sounds by using a program that combined an amplitude table with parameters for cycles and delays. You then used this program to add sounds to animated graphics.

The following new instructions were used:

Instruction	*Program Where Used and Description*
TFR D,Y	12 used to transfer data from the register D to register Y
LEAU TABLE,PCR	12 calculates distance (in bytes) to location of TABLE and inserts these values in the object code
LDA ,U+	12 loads accumulator A from stack, increases stack pointer by 1
LSLA	12 shifts all bits in A one place to the left, most significant bit is lost

ORA #n	12 logical OR with accumulator with the value n
TFR D,X	12 transfer data from register D to register X
LEAY n,Y	12 increment the value in Y register by the value n
LDA #'?	12 load accumulator A with the ASCII code of the question mark
CMPA #'#	12 compare value in A with the ASCII code of the number sign
ADDA #n	12 add the value n to accumulator A
ANDA #n	12 logical AND accumulator A with n
ABA	12 add accumulator B to accumulator A
LEAU n,U	13 increment stack pointer by n
PSHS U,X,D	13 push values in U,X and D onto the stack
PULS U,X,D	13 pull values of stack and place in U,X, and D
STD ,Y + +	14 store the most significant byte of D into the memory location stored where Y is pointing and the least significant byte of D into the succeeding location, increment Y by 2
LDD ,Y + +	15 load register D from the location stored in Y and Y + 1, increment Y by 2
STD −$1680,X	15 store the value in D into $1680 less than the location held in X

Chapter Test

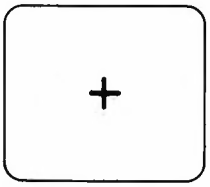
1. Describe the purpose of the statement
   DELAY RMB 1 _____
   _____
   _____

2. Describe the purpose of the subroutines
   a. AUDON _____
   _____
   b. SELMUX _____
   _____

3. Program 13 displayed red, then blue, then yellow bars with a tone following each display. A one line change would make the colors orange, magenta, and cyan. Show the necessary change.

4. Rewrite Program 13 to display vertical bars of color.

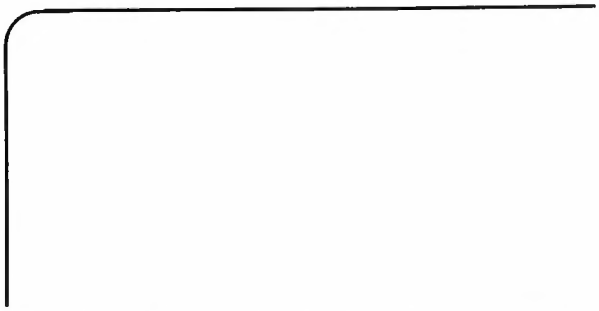
5. Study the data used to create the rotating color bar in Program 14. Give the data bytes (using red) that would form the following figure:



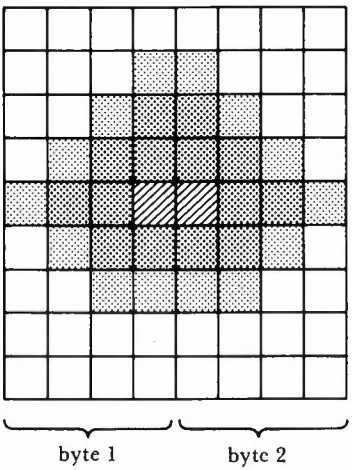
Hint: Combine the four bars of Program 14

Address	*Data*	*Data*	*Address*
0F30			0F31
0F50			0F51
0F70			0F71
0F90			0F91
0FB0			0FB1
0FD0			0FD1
0FF0			0FF1
1010			1011
1030			1031
1050			1051
1070			1071
1090			1091
10B0			10B1
10D0			10D1

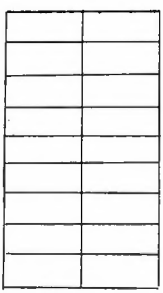
6. Revise Program 14 to display the data that you supplied in test exercise 5. Then see if you can change the color to blue and yellow.



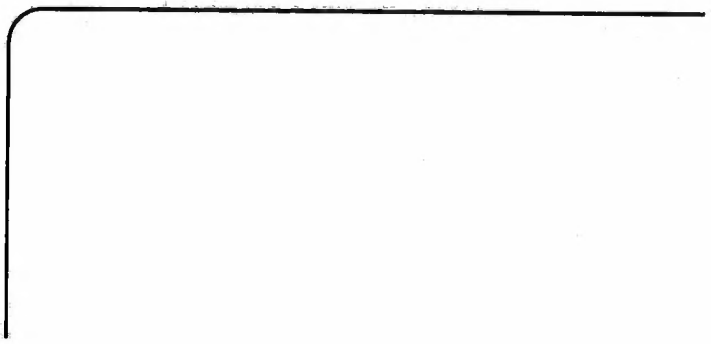
7. Give the data bytes for the following figure where:



Use hex digits.

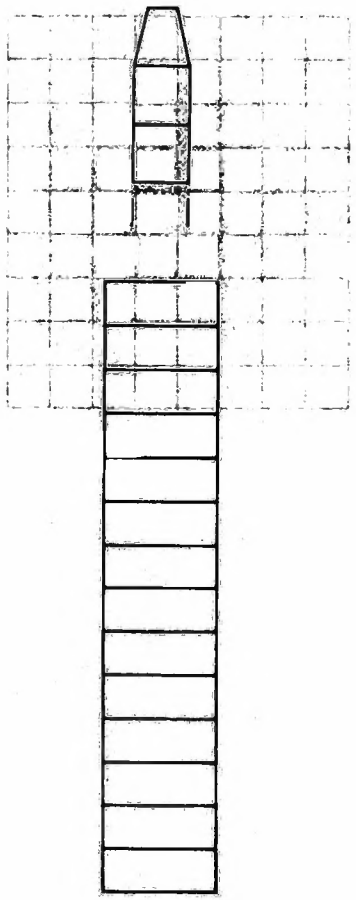


8. Revise Prgram 15 to use the figure defined by your data of test exercise 7.

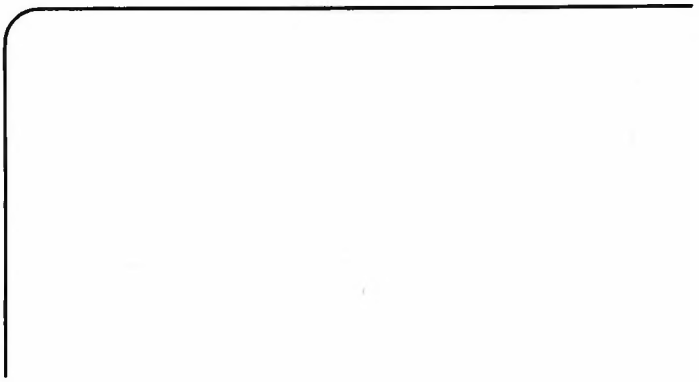


9. Design a rocket similar to the one below. Fill in the data bytes that would be used to display it. Keep it one byte wide.

Data bytes



10. Use your data of test exercise 9 to write a program that sends the rocket from the bottom to the top of the screen.



Answers to Odd-Numbered Exercises in Chapter Test

1. DELAY RMB 1 reserves one byte in memory for the value named DELAY. This value may then be referred to in the program by its name.

3. START LDA #$E8

5.

0F30	3	0	0F51
0F50	3	0	0F51
0F70	3	0	0F71
0F90	3	0	0F91
0FB0	3	0	0FB1
0FD0	3	0	0FD1
0FF0	FF	FF	0FF1
1010	FF	FF	1011
1030	3	0	1031
1050	3	0	1051
1070	3	0	1071
1090	3	0	1091
10B0	3	0	10B1
10D0	3	0	10D1

7.

00	00
02	80
09	60
25	58
97	D6
25	58
0A	A0
00	00
00	00

9. Yours may be entirely different. This is just a sample.

```
28
96
96
AA
96
96
AA
96
96
AA
00
C3
C3
C3
```

Chapter 7

Joystick Animation

In previous programs, we programmed the computer to move figures about the screen. In this chapter, we'll give you more control of the action. You will learn how to move a figure around the screen by maneuvering the joysticks. You will use many of the techniques learned in Chapters 5 and 6. In addition, you'll learn how to read the joysticks when using an assembler and how to convert the joystick reading to a screen location.

We'll introduce the use of decimal numbers in the operand of assembly instructions. You'll also learn how to use interrupts to time the action that is taking place on the video screen.

Designing a Joystick Program

To design a joystick program, you should use what you learned in the first six chapters. Plan what you want to do before you start writing a program.

Step 1

The joystick can be made to move an object about the screen. A flying saucer seems like a logical choice for the object. If you use graphics mode 6C, you can have a small saucer of the color of your choice. Keep it small so that you can move it more easily and quickly. You used an airplane in Program 15 of Chapter 6, but we chose a saucer this time because it is more symmetrical and can be moved in any direction without redrawing it.

The data defining the saucer is shown in Table 7-1; the shape produced by the data is shown in Figure 7-1.

Table 7-1. Data for Flying Saucer

Hex Values	*Binary Values*
00 00	00000000 00000000
0A A0	00001010 10100000
AA AA	10101010 10101010
AA AA	10101010 10101010
08 20	00001000 00100000
00 00	00000000 00000000

Byte 1	*Byte 2*
00 00 00 00	00 00 00 00
00 00 10 10	10 10 00 00
10 10 10 10	10 10 10 10
10 10 10 10	10 10 10 10
00 00 10 00	00 10 00 00
00 00 00 00	00 00 00 00

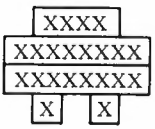


Figure 7-1. Flying Saucer Shape Table

In the program, this shape table will take the form:

```
TABLE   FCB 0,0,$A,$A0
        FCB $AA,$AA,$AA,$AA
        FCB 8,$20,0,0
```

Step 2

The joysticks must be read in order to move the saucer. To get the data, we will use a ROM subroutine that we will call JOYST. Its entry point is $A00A. The horizontal value of the right joystick is stored by the computer in memory location $15A, the vertical value in $15B.

After obtaining the data for the joysticks, these values are stored in temporary locations called VERTJ and HORIJ. We will use the following program segment to read and store the joystick values.

```
JSR JOYST     ←read joystick
LDA $15A      ←get horizontal data
STA HORIJ     ←store it
LDA $15B      ←get vertical data
STA VERTJ     ←store it
```

Step 3

Another important part of the program will be the conversion of the joystick readings to the placement of the saucer on the video screen. We'll use the vertical motion of the stick to control the vertical position of the saucer and the horizontal motion of the stick to control the horizontal position of the saucer.

We'll offset the screen to start at $600 as before. Now compare the joystick and screen values.

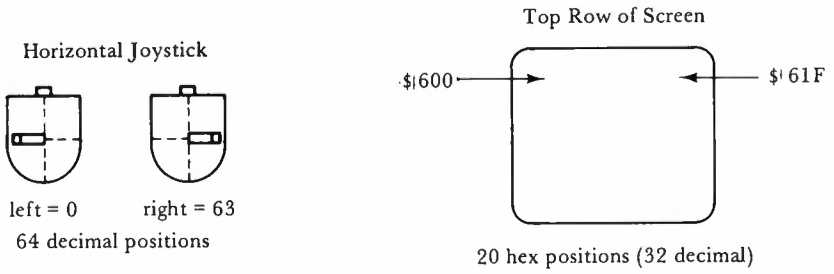


Figure 7-2. Horizontal Joystick and Screen Positions

If we divide the joystick reading by 2, we will get the correct value to position the saucer at any given horizontal video position.

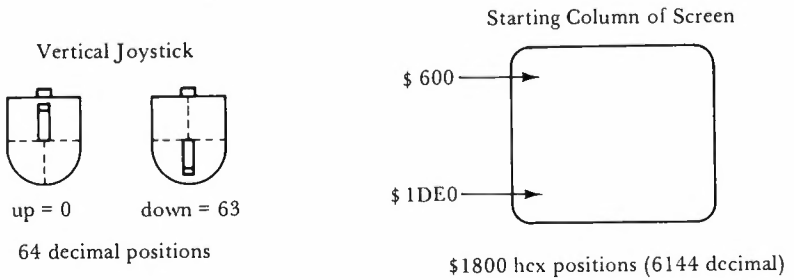


Figure 7-3. Vertical Joystick and Screen Positions

If we multiply the vertical joystick position by 96, we will get the correct value to position the saucer at any given video row. The correct screen position is then obtained by adding the horizontal offset and the vertical offset to the beginning location ($600) of the video screen. The screen calculations will take place in a subroutine labeled SCREEN.

```
SCREEN  LDA VERTJ       get vertical position
        CMPA #61        keeps saucer on the screen
        BLO CALC        if lower than 61, go on
        LDA #61         if equal or greater, use 61
CALC    LDB #96
        MUL             multiply vertical by 96
        TFR D,Y         save result in Y
        LDB HORIJ       get horizontal position
        LSRB            shift right to divide by 2
        CMPB #30        make sure it stays on screen
        BLO DO          if lower, go do it
        LDB #30         if not lower, make it 30
DO      CLRA            D now holds horizontal offset
        LEAX D,Y        add D and Y, put in X
        LEAX VIDBEG,X   add beginning of screen
        RTS             return to main program
```

Step 4

The old saucer position must be erased before the saucer is placed in a new position. This is done by the following:

```
        BSR SCREEN      get address of old figure
        LDA #6          6 rows define the figure
ERASE   CLR ,X+         clears 1 row (2 bytes)
        CLR ,X
        LEAX 31,X       move to new row
        DECA            decrement row count
        BNE ERASE       go back if not done
```

The saucer shape is accessed from the table in the same manner that you have previously used. The complete program is shown in Figure 7-4.

```
0001 0600                       JOYST   EQU $A00A
0002 0600                       KEYIN   EQU $A000
0003 0600                       VIDBEG  EQU $600
0004 0600                               ORG VIDBEG+$1800
0005 1E00 86E0          START   LDA #$E0
0006 1E02 B7FF22                STA $FF22
0007 1E05 B7FFC3                STA $FFC3
0008 1E08 B7FFC5                STA $FFC5
0009 1E0B B7FFC7                STA $FFC7
0010 1E0E 4F                    CLRA
0011 1E0F 5F                    CLRB
0012 1E10 8E0600                LDX #VIDBEG
0013 1E13 ED81          LOOP    STD ,X++
0014 1E15 8C1E00                CMPX #VIDBEG+$1800
0015 1E18 25F9                  BLO LOOP
0016 1E1A F71E8F                STB HORIJ
```

```
0017 1E1D F71E90              STB VERTJ
0018 1E20 AD9FA000    MOVE    JSR [KEYIN]
0019 1E24 8158                CMPA #'X
0020 1E26 260F                BNE GO
0021 1E28 8600                LDA #0
0022 1E2A B7FF22              STA $FF22
0023 1E2D B7FFC2              STA $FFC2
0024 1E30 B7FFC4              STA $FFC4
0025 1E33 B7FFC6              STA $FFC6
0026 1E36 3F                  SWI
0027 1E37 AD9FA00A    GO      JSR [JOYST]
0028 1E3B 8D31                BSR SCREEN
0029 1E3D 8606                LDA #6
0030 1E3F 6F80        ERASE   CLR ,X+
0031 1E41 6F84                CLR ,X
0032 1E43 30881F              LEAX 31,X
0033 1E46 4A                  DECA
0034 1E47 26F6                BNE ERASE
0035 1E49 B6015A              LDA $15A
0036 1E4C B71E8F              STA HORIJ
0037 1E4F B6015B              LDA $15B
0038 1E52 B71E90              STA VERTJ
0039 1E55 8D17                BSR SCREEN
0040 1E57 338D0036            LEAU TABLE,PCR
0041 1E5B 8606                LDA #6
0042 1E5D B71E8E              STA COUNT
0043 1E60 ECC1        DISPLA  LDD ,U++
0044 1E62 ED84                STD ,X
0045 1E64 308820              LEAX 32,X
0046 1E67 7A1E8E              DEC COUNT
0047 1E6A 26F4                BNE DISPLA
0048 1E6C 20B2                BRA MOVE
0049 1E6E B61E90      SCREEN  LDA VERTJ
0050 1E71 813D                CMPA #61
0051 1E73 2502                BLO CALC
0052 1E75 863D                LDA #61
0053 1E77 C660        CALC    LDB #96
0054 1E79 3D                  MUL
0055 1E7A 1F02                TFR D,Y
0056 1E7C F61E8F              LDB HORIJ
0057 1E7F 54                  LSRB
0058 1E80 C11E                CMPB #30
0059 1E82 2502                BLO DO
0060 1E84 C61E                LDB #30
0061 1E86 4F          DO      CLRA
0062 1E87 30AB                LEAX D,Y
0063 1E89 30890600            LEAX VIDBEG,X
0064 1E8D 39                  RTS
0065 1E8E             COUNT   RMB 1
0066 1E8F             HORIJ   RMB 1
0067 1E90             VERTJ   RMB 1
0068 1E91 00000AA0    TABLE   FCB 0,0,$A,$A0
0069 1E95 AAAAAAAA            FCB $AA,$AA,$AA,$AA
0070 1E99 08200000            FCB 8,$20,0,0
0071 1E9D                     END
```

```
CALC   1E77  COUNT  1E8E  DISPLA 1E60  DO     1E86
ERASE  1E3F  GO     1E37  HORIJ  1E8F  JOYST  A00A
KEYIN  A000  LOOP   1E13  MOVE   1E20  SCREEN 1E6E
START  1E00  TABLE  1E91  VERTJ  1E90  VIDBEG 0600
```

Figure 7-4. Flying Saucer

Using the Flying Saucer

Enter and run the flying saucer program. Move the right joystick up and down, and the saucer moves up and down. Move the joystick left to right, and the saucer moves left to right. You are in complete control of the saucer's movement. Practice takeoffs and landings. If you feel ambitious, add some objects on the ground. Add a corn field and spray your crops from the air. Add some cows and round up your cattle from the air. Try some of your own ideas.

New Instructions and Data Forms

Some new instructions were used in the program, and some data was expressed in decimal form.

- In line 14: CMPX #VIDBEG + $1800
  The assembler will add the address of VIDBEG ($600) to $1800 and compare the result to the value in register X. The arithmetic can be performed in an assembler instruction.
- In line 30: CLR ,X +
  This clears (or puts a zero into) the location stored in X. Register X is then incremented. This statement, along with CLR X which follows, clears one row of the saucer because X holds the screen address of the saucer.
- In line 32: LEAX 31,X
  We have usually used the hex form for operands or data. In that case, the value is preceded by a $ sign. The SDS80C assembler will interpret values as decimal if the $ sign is omitted. Thus 31 (decimal) is the same as $1F (hex). Either form can be used. The same form of this instruction is used in line 45. Decimal values are used again in the SCREEN subroutine. The assembler will convert the decimal values, as can be seen in the object program.
- In line 54: MUL
  This multiplies the value in register A by the value in register B. The result is placed in register D.

- In line 57: LSRB
  This shifts each bit in register B one place to the right. It is a quick way to divide a value by two.

  Example

  register B → | 00011010 | = 26 decimal

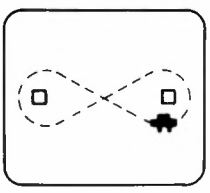
  After a LSRB instruction

  register B → | 00001101 | = 13 decimal

- In line 62: LEAX D,Y
  The value in D is added to the value in Y. The result is placed in X.

Saucer around the Pylons

Now that you've had some practice at maneuvering a flying saucer, why not set up a race course for the saucer? You could put a pylon near each side of the screen and fly around the pylons.



The flight pattern could be figure eights or whatever you decide. The length of the race could be any number of laps. You could even have the computer calculate the time it takes you to complete a given number of laps.

After we discuss how to use the computer's interrupts for timing, we'll modify Program 16 to include the pylons and timing.

Interrupts

Suppose the computer receives an interrupt signal while it is proceeding through a program. The central processor stops whatever it is doing, saves all of the registers on the stack, and jumps off to a special part of the program called an interrupt service routine. The interrupt routine is much like a subroutine, but it uses an RTI (ReTurn from Interrupt) instruction to return to the main program when the inter-

rupt service routine has been completed. In our case, we will have the interrupt service routine increment a clock counter. When the RTI instruction is executed, the routine restores all of the registers and control is returned to the main program at the point at which it was interrupted.

Example

```
INTRPT  INC  1087      ←increase the count in
                          this video location
        LDA  $FF02     ←clear interrupt status bit
        RTI            ←return to main program
```

What causes an interrupt? The interrupt system is a part of the computer's hardware. An electrical pulse is triggered from the computer's clock. Even though the interrupt system is in hardware, it can be controlled by software (your program). You can turn it on or off whenever you wish. The service routine is just a short block of program that you write. It can do whatever you want it to do. In the previous example, it merely displays the character whose ASCII code corresponds to the count in location 1087 at the upper right corner of the screen.

There are three interrupts in the 6809 CPU: the Interrupt ReQuest (IRQ), the Fast Interrupt ReQuest (FIRQ) and the Non-Maskable Interrupt (NMI). We will be using the IRQ in Program 17.

In order to enable the IRQ, bit 4 of the Condition Code Register must be cleared (set to zero). This can be done with a logical AND instruction with the Condition Code register:

```
ANDCC #%11101111
```

AND — Condition Code Register — with binary value that immediately follows — this zero assures that bit 4 will be cleared

To shut off (disable) the IRQ, bit 4 of the Condition Code Register must be set to one. To do this, you can use the logical OR instruction

```
ORCC #%00010000
```

OR — Condition Code Register — with immediate binary value — this one assures that bit 4 is set

As stated earlier, the processor pushes the values of all the registers onto the stack when the IRQ is triggered. The values are

saved because your interrupt service routine might alter one or more of the registers. The program counter is automatically loaded with whatever addresses are in locations $FFF8 and $FFF9. This is a ROM area, and the contents in these locations cannot be changed. The ROM holds the memory address $100 in these locations. Beginning at address $100 is a series of interrupt jump vectors (memory locations to select appropriate interrupts). The IRQ vector that we need is at location $10D. We must change the value at $10D to the address of the beginning of our interrupt routine. It is done in the following manner:

```
LDX  INTRPT      the label of our interrupt
                 service routine
STX  $10D        ←store the address of INTRPT
                 in $10D
```

The interrupt signal comes through the PIA (Peripheral Interface Adaptor) with which the computer communicates with the outside world. The next step in enabling a clock interrupt involves PIA #0, which is accessed through locations $FF00-$FF03, inclusive. A separate clock signal is located at each of two ports of PIA #0. A 63.5 microsecond clock is accessed at $FF00 and $FF01. A 60-Hz (a Hertz unit is one cycle per second) signal is accessed at $FF02 and $FF03. This is the signal that we will use. When you write to $FF03, the 60-Hz clock is enabled. When you read $FF02, the 60-Hz clock is disabled. $FF02 accesses the Data Register, and $FF03 accesses the Control Register. To enable the interrupt, you set bit #0 of the Control Register. Since the Control Register normally contains 00110101 = 35 hex, bit zero can be set by

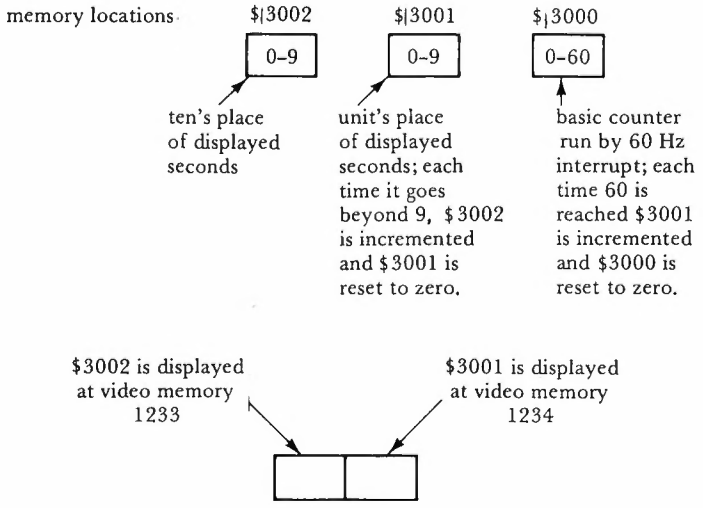
```
LDA #$35
STA $FF03
```

When an interrupt occurs, bit #7 of the Control Register goes high (is set to one). This bit must be cleared before another interrupt can occur. To clear bit #7 of the Control Register, you must read the Data Register of the port as shown in an earlier example in the interrupt service routine. The instruction that does this is

```
LDA $FF02      ←turns off bit #7
```

You now have all the bits and pieces necessary to write a program that will display a timer that will count 60 times each second.

The interrupt service routine that we will use is a little more detailed than our earlier example. The computer displays characters on the screen according to their ASCII codes (see Appendix B). Therefore, you must convert the counter to meaningful codes, one digit at a time. This is done in the following way. The two digits together can count in decimal values from zero through 99 seconds.



The interrupt service routine to be used is as follows:

```
INTRPT  LDA $3000     $3000 is originally loaded with 0
        INC $3000     count up one
        CMPA #$3C     see if it's 60 (3C hex)
        BLO SKIP      if not, skip to the exit of the inter-
                      rupt routine
        INC $3001     if 60, increment the units place of
                      the seconds counter
        LDA $3001     load unit's place in accumulator A
        ADDA #$30     add $30 for correct ASCII code
        CMPA #$3A     compare to ASCII code one
                      beyond a 9
        BNE ON        if 0-9, go on to ON to display the
                      unit's digit
        LDA #0        if greater than 9, load a zero into A
```

```
        STA $3001     put the zero in the unit's place
                      ($3001)
        INC $3002     increase count by one in ten's place
        LDA $3002     load ten's place
        ADDA #$30     convert to ASCII code
        STA 1233      display the ten's digit
        LDA #$30      load ASCII code for zero in ac-
                      cumulator A
ON      STA 1234      display the unit's digit
        LDA #0        load a zero
        STA $3000     store in clock counter (count from
                      zero to 60 again)
SKIP    LDA $FF02     turn off interrupt
        RTI           return to main program
```

The complete timer program is given in Figure 7-5. Note that our interrupt service routine makes up about one-half of the program.

```
0001 0600                      ORG $2000
0002 2000 8600       GO        LDA #0
0003 2002 B73000               STA $3000
0004 2005 B73001               STA $3001
0005 2008 B73002               STA $3002
0006 200B 868F                 LDA #$8F
0007 200D C68F                 LDB #$8F
0008 200F 8E0400               LDX #$400
0009 2012 ED81       LOOP      STD ,X++
0010 2014 8C0600               CMPX #$600
0011 2017 25F9                 BLO LOOP
0012 2019 8E2034               LDX #INTRPT
0013 201C BF010D               STX $10D
0014 201F 8635                 LDA #$35
0015 2021 B7FF03               STA $FF03
0016 2024 1CEF                 ANDCC #%11101111
0017 2026 AD9FA000   KEYIN     JSR [$A000]
0018 202A 27FA                 BEQ KEYIN
0019 202C 1A10                 ORCC #%00010000
0020 202E 8634                 LDA #$34
0021 2030 B7FF03               STA $FF03
0022 2033 3F                   SWI
0023 2034 B63000     INTRPT    LDA $3000
0024 2037 7C3000               INC $3000
0025 203A 813C                 CMPA #$3C
0026 203C 2526                 BLO SKIP
0027 203E 7C3001               INC $3001
0028 2041 B63001               LDA $3001
0029 2044 8B30                 ADDA #$30
0030 2046 813A                 CMPA #$3A
0031 2048 2612                 BNE ON
```

```
0032 204A 8600                LDA #0
0033 204C B73001              STA $3001
0034 204F 7C3002              INC $3002
0035 2052 B63002              LDA $3002
0036 2055 8B30                ADDA #$30
0037 2057 B704D1              STA 1233
0038 205A 8630                LDA #$30
0039 205C B704D2        ON    STA 1234
0040 205F 8600                LDA #0
0041 2061 B73000              STA $3000
0042 2064 B6FF02        SKIP  LDA $FF02
0043 2067 3B                  RTI
0044 2068                     END

GO      2000   INTRPT 2034   KEYIN  2026   LOOP   2012
ON      205C   SKIP   2064
```

Figure 7-5. Timer

How Program 17 Works

After the IRQ is enabled, the computer proceeds to its POLCAT routine to read the keyboard. However, the interrupt occurs 60 times per second while the computer is scanning the keyboard. Each time the interrupt occurs, the interrupt service routine (INTRPT) is executed. The count is shown on the screen as programmed in the interrupt service routine.

Thus, the computer seems to be doing two things at once: scanning the keyboard and counting on the video screen. Whenever you want to stop the timer, press any key and the program will return to the monitor (ABUG in our system).

Using the Timer Function

It is quite simple to combine the timer and the pylons, mentioned earlier, to the Flying Saucer program to race the saucer around the course and time the flight.

A start timer section is added after the saucer first appears.

```
TIME   LDX INTRPT
       STX $10D
       LDA #$35
       STA $FF03
       ANDCC #%11101111
```

When you have completed your flight, press the X key. This will turn off IRQ and restore the text screen so that you can see the time of flight.

```
MOVE   JSR [KEYIN]
       CMPA #'X
       BNE GO
       ORCC #%00010000 ← turn off IRQ
       LDA #$34
       STA $FF03
       LDA #0
       STA $FF22
       STA $FFC2
       STA $FFC4
       STA $FFC6
       SWI             ←——— return to monitor
```

The pylons are added after the screen is first cleared, and the interrupt service routine is added near the end of the program.

```
PYLONS LDA #$FF
       STA $11C4
       STA $11E4
       STA $1204
       STA $11DA
       STA $11FA
       STA $121A
INTRPT LDA $3000
       INC $3000
       CMPA #$3C
       BLO SKIP
       INC $3001
       LDA $3001
       ADDA #$30
       CMPA #$3A
       BNE ON
       LDA #0
       STA $3001
       INC $3002
       LDA $3002
       ADDA #$30
       STA $4FC
       LDA #$30
```

```
ON      STA $4FD
        LDA #0
        STA $3000
SKIP    LDA $FF02
        RTI
```

The complete Timed Flying Saucer program is shown in Figure 7-6.

```
0001 0600                JOYST  EQU $A00A
0002 0600                KEYIN  EQU $A000
0003 0600                VIDBEG EQU $600
0004 0600                       ORG VIDBEG+$1800
0005 1E00 86E0           START  LDA #$E0
0006 1E02 B7FF22                STA $FF22
0007 1E05 B7FFC3                STA $FFC3
0008 1E08 B7FFC5                STA $FFC5
0009 1E0B B7FFC7                STA $FFC7
0010 1E0E 4F                    CLRA
0011 1E0F 5F                    CLRB
0012 1E10 8E0600                LDX #VIDBEG
0013 1E13 ED81           LOOP   STD ,X++
0014 1E15 8C1E00                CMPX #VIDBEG+$1800
0015 1E18 25F9                  BLO LOOP
0016 1E1A 86FF           PYLONS LDA #$FF
0017 1E1C B711C4                STA $11C4
0018 1E1F B711E4                STA $11E4
0019 1E22 B71204                STA $1204
0020 1E25 B711DA                STA $11DA
0021 1E28 B711FA                STA $11FA
0022 1E2B B7121A                STA $121A
0023 1E2E 8600                  LDA #0
0024 1E30 B73000                STA $3000
0025 1E33 B73001                STA $3001
0026 1E36 B73002                STA $3002
0027 1E39 F71EF6                STB HORIJ
0028 1E3C F71EF7                STB VERTJ
0029 1E3F 8E1EC1         TIME   LDX #INTRPT
0030 1E42 BF010D                STX $10D
0031 1E45 8635                  LDA #$35
0032 1E47 B7FF03                STA $FF03
0033 1E4A 1CEF                  ANDCC #%11101111
0034 1E4C AD9FA000       MOVE   JSR [KEYIN]
0035 1E50 8158                  CMPA #'X
0036 1E52 2616                  BNE GO
0037 1E54 1A10                  ORCC #%00010000
0038 1E56 8634                  LDA #$34
0039 1E58 B7FF03                STA $FF03
```

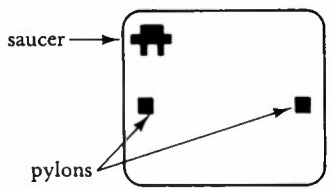
```
0040 1E5B 8600              LDA #0
0041 1E5D B7FF22            STA $FF22
0042 1E60 B7FFC2            STA $FFC2
0043 1E63 B7FFC4            STA $FFC4
0044 1E66 B7FFC6            STA $FFC6
0045 1E69 3F                SWI
0046 1E6A AD9FA00A    GO    JSR [JOYST]
0047 1E6E 8D31              BSR SCREEN
0048 1E70 8606              LDA #6
0049 1E72 6F80        ERASE CLR ,X+
0050 1E74 6F84              CLR ,X
0051 1E76 30881F            LEAX 31,X
0052 1E79 4A                DECA
0053 1E7A 26F6              BNE ERASE
0054 1E7C B6015A            LDA $15A
0055 1E7F B71EF6            STA HORIJ
0056 1E82 B6015B            LDA $15B
0057 1E85 B71EF7            STA VERTJ
0058 1E88 8D17              BSR SCREEN
0059 1E8A 338D006A          LEAU TABLE,PCR
0060 1E8E 8606              LDA #6
0061 1E90 B71EF5            STA COUNT
0062 1E93 ECC1        DISPLA LDD ,U++
0063 1E95 ED84              STD ,X
0064 1E97 308820            LEAX 32,X
0065 1E9A 7A1EF5            DEC COUNT
0066 1E9D 26F4              BNE DISPLA
0067 1E9F 20AB              BRA MOVE
0068 1EA1 B61EF7      SCREEN LDA VERTJ
0069 1EA4 813D              CMPA #61
0070 1EA6 2502              BLO CALC
0071 1EA8 863D              LDA #61
0072 1EAA C660        CALC  LDB #96
0073 1EAC 3D                MUL
0074 1EAD 1F02              TFR D,Y
0075 1EAF F61EF6            LDB HORIJ
0076 1EB2 54                LSRB
0077 1EB3 C11E              CMPB #30
0078 1EB5 2502              BLO DO
0079 1EB7 C61E              LDB #30
0080 1EB9 4F          DO    CLRA
0081 1EBA 30AB              LEAX D,Y
0082 1EBC 30890600          LEAX VIDBEG,X
0083 1EC0 39                RTS
0084 1EC1 B63000      INTRPT LDA $3000
0085 1EC4 7C3000            INC $3000
0086 1EC7 813C              CMPA #$3C
0087 1EC9 2526              BLO SKIP
0088 1ECB 7C3001            INC $3001
0089 1ECE B63001            LDA $3001
0090 1ED1 8B30              ADDA #$30
0091 1ED3 813A              CMPA #$3A
0092 1ED5 2612              BNE ON
0093 1ED7 8600              LDA #0
```

```
0094 1ED9 B73001              STA $3001
0095 1EDC 7C3002              INC $3002
0096 1EDF B63002              LDA $3002
0097 1EE2 8B30                ADDA #$30
0098 1EE4 B704FC              STA $4FC
0099 1EE7 8630                LDA #$30
0100 1EE9 B704FD       ON     STA $4FD
0101 1EEC 8600                LDA #0
0102 1EEE B73000              STA $3000
0103 1EF1 B6FF02       SKIP   LDA $FF02
0104 1EF4 3B                  RTI
0105 1EF5              COUNT  RMB 1
0106 1EF6              HORIJ  RMB 1
0107 1EF7              VERTJ  RMB 1
0108 1EF8 00000AA0     TABLE  FCB 0,0,$A,$A0
0109 1EFC AAAAAAAA            FCB $AA,$AA,$AA,$AA
0110 1F00 08200000            FCB 8,$20,0,0
0111 1F04                     END

CALC    1EAA  COUNT   1EF5  DISPLA 1E93  DO      1EB9
ERASE   1E72  GO      1E6A  HORIJ  1EF6  INTRPT  1EC1
JOYST   A00A  KEYIN  A000  LOOP   1E13  MOVE    1E4C
ON      1EE9  PYLONS 1E1A  SCREEN 1EA1  SKIP    1EF1
START   1E00  TABLE   1EF8  TIME   1E3F  VERTJ   1EF7
VIDBEG  0600
```

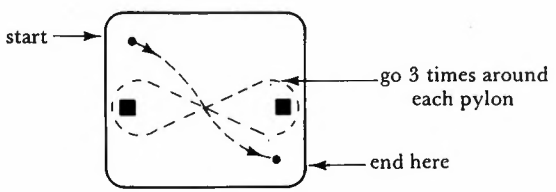
Figure 7-6. Timed Flying Saucer

Timing your Flights

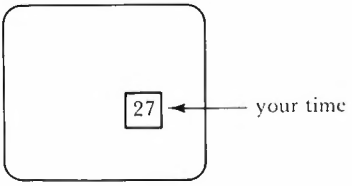
When the timer is started in Program 18, the screen will display the following:



First, fly three laps of the course in figure eight fashion.



After the third complete lap, proceed to the lower right corner of the screen. Then press the X key on the keyboard to display your time. The screen will display



Choice of shape and color for the saucer can be varied as you wish. Figure 7-7 shows a few possibilities that are more colorful.

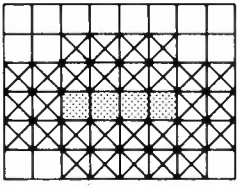
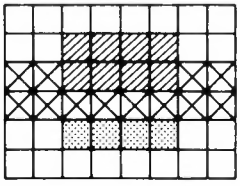
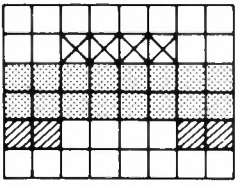
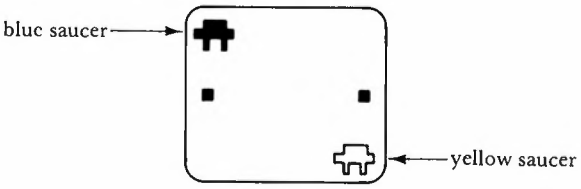
SHAPE	TABLE VALUES	
	Binary	Hex
	00000000 00000000 00001111 11110000 11111111 11111111 11110101 01011111 11111111 11111111 00001111 11110000	00 00 0F F0 FF FF F5 5F FF FF 0F F0
	00000000 00000000 00001010 10100000 11111010 10101111 11111111 11111111 00000101 01010000 00000000 00000000	00 00 0A A0 FA AF FF FF 05 50 00 00
	00000000 00000000 00001111 11110000 01010101 01010101 01010101 01010101 10100000 00001010 00000000 00000000	00 00 0F F0 55 55 55 55 A0 0A 00 00

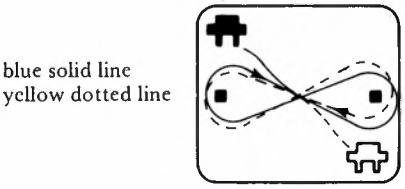
Figure 7-7. Other Saucer Shapes

Double Saucers

It would be interesting to put two saucers on the screen at one time. Let one of them be controlled by the right joystick and the other by the left joystick. Let one start at the upper left corner of the screen and the other start at the lower right corner of the screen. Let one be blue and the other yellow.



The two saucers could then fly figure eights in opposite directions.

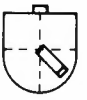


To add the second saucer, you could use the following information.

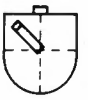
- The left joystick data is available at $15C (horizontal) and $15D (vertical). The data could be stored in memory as LHOR and LVER.

```
LDA $15C
STA LHOR
LDA $15D
STA LVERT
```

- To start the second saucer in the lower right corner of the screen, begin the program with the left joystick in the down, right position. The right joystick should start in the up, left position.



left joystick
right, down



right joystick
left, up

```
LDA #61
STA LHOR
STA LVER
```

- The ERASE and SCREEN sections of the previous program would have to be modified to erase and move both saucers.
- A second data table would be needed for the display of the second saucer.

We'll let you design the program using these suggestions. The results may surprise you. Write your program, then enter and run it. What happens if the saucers collide? Do you get a blend of colors, or does one saucer wipe out the other? Play around with your program for some relaxation before going on to the chapter summary and test.

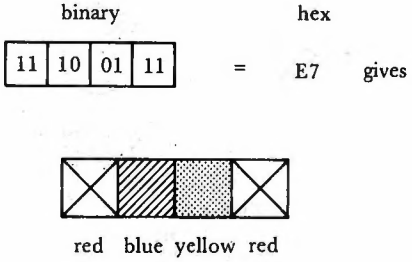
Summary

This chapter was devoted to the use of joysticks to move objects on the video screen. The key to quick drawing and movement of the objects is to keep the drawings small. Then the time consumed to draw and move them will be small, and the objects will appear to move smoothly.

- A table was used to define the objects in mode 6C. One pair of bits define one colored element.

Bit Pair	*Color Set 0*	*Color Set 1*
11	red	orange
10	blue	magenta
01	yellow	cyan
00	green	buff

- One byte of data provides four colored elements.

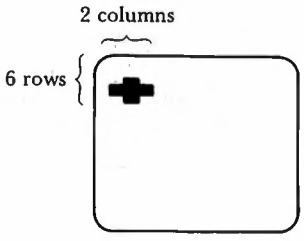


- Placing combinations of data bytes in appropriate video memory locations displays an object.

Hex Data

Byte	*Location*
0	600
0	601
A	620
A0	621
AA	640
AA	641
AA	660
AA	661
A	680
A0	681
0	6A0
0	6A1

Display



- The joystick positions were read by a ROM subroutine and stored in RAM.

```
JSR JOYST
LDA $15A
STA HORIJ
LDA $15B
STA VERTJ
```

- The joystick readings were modified to give row and column values for the video screen.

The vertical value was multiplied by 96.
The horizontal value was divided by 2.
The graphic screen offset was added to the sum of these two values.

- New Instructions

BSR SCREEN branches to a subroutine named SCREEN located within the program.

CMPX #VIDBEG + $1800 compares the value in X to the sum of beginning video memory and 1800.

CLR ,X + clears the memory location stored in X and increments X.

INC $3001 increments the value in memory location #3001.

JSR KEYIN jumps to a subroutine whose starting address is stored in the memory location called KEYIN.

LDD ,U + loads register D from U and increments U.

LEAX D,Y adds the values in D and Y and places the result in X.

LEAX 31,X adds decimal value 31 to the value in X register.

LSRB shifts each bit in register B one place to the right.

MUL multiplies the values in registers A and B. The result is placed in register D.

RTI returns from an interrupt.

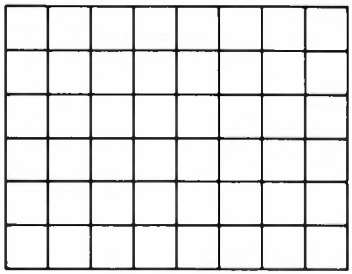
- The use of interrupts was introduced.

```
LDX INTRPT  }  modifies IRQ vector to point to your
STX $10D    }  interrupt service routine.
LDA #$35
STA $FF03       enables interrupt
ANDCC #%11101111
LDA $FF02       clears interrupt status bit
ORCC #%00010000 disables interrupt
```

Chapter Test

1. Fill in the object that would be created on the screen by this data table. (Used as in Table 7-1.) Use these color codes:


```
TABLE     FCB 1, $40, $28, $28
          FCB $D6, $97, $C, $30
          FCB $30, $C, $C0, 3
```



2. Fill in the hex values for a data table that would produce this figure. (Use the same color codes as in exercise 1.)

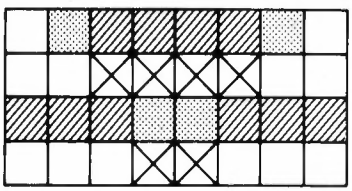


TABLE FCB _____, _____, _____, _____
FCB _____, _____, _____, _____

3. Give the purpose of each of the following instructions.

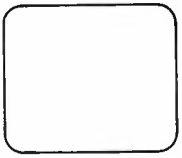
JSR JOYST _____
LDA $15A _____
STA HORIJ _____
LDA $15B _____
STA VERTJ _____

4. If you are using mode 6C with the beginning location of video memory offset to $600, what would the last video memory location (lower right corner) be?

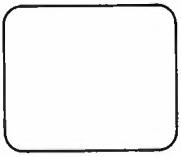
$ _____

5. Using the Flying Saucer joystick program of Figure 7-4, show the approximate position of the saucer on the screen for the following joystick positions.

a.



b.



6. In Figure 7-4, the instruction LEAX 31,X was used with the decimal operand 31,X. Give the equivalent instruction in hex form.

_____

7. Suppose register B holds the hex value $A5. What would be in register B after the following two instructions are executed? (Give binary, hex, and decimal forms of the result.)

LSRB
LSRB

Register B would now hold:

_____ binary

_____ hex

_____ decimal

8. Is the result (in exercise 7 of chapter test) equal to one-fourth of the original value in register B? If not, why not?

_____

_____

9. To enable the IRQ in the timer program (Figure 7-5), we used the AND instruction to clear bit #4 of the Condition Code register. Suppose the Condition Code register contained $D3 just before the AND instruction was executed. What would be in the Condition Code register after the execution of

ANDCC #%11101111

_____

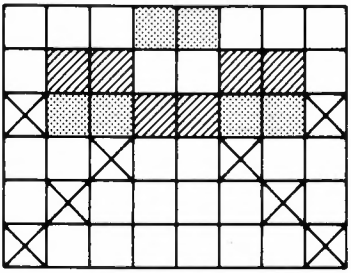
10. If the result of exercise 9 of the chapter test is in the Condition Code register, how would it be changed by this instruction?

    ORCC #%00010000

    _____

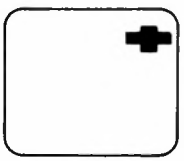
    _____

Answers to Odd-Numbered Exercises of Chapter Test

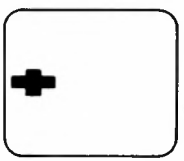
1. 

3.
```
JSR JOYST    reads the joystick
LDA $15A     gets horizontal joystick data
STA HORIJ    stores the horizontal data
LDA $15B     gets vertical joystick data
STA VERTJ    stores the vertical data
```

5. a. b.



upper, right



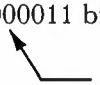
center, left

7. After two shift rights, B would contain

    00101001 binary
    29 hex
    41 decimal

9. Since $D3 = 11010011 binary, the value resulting from the AND instruction would be:

    11000011 binary or $C3 hex



this bit turned off

Chapter 8

Text

The TRS-80 Color Computer may be programmed for either the text mode or one of the many graphic modes. In the first part of this chapter, we'll discuss displaying text in the normal text mode. Then, in the second part, we'll show how to draw text characters in one of the graphic modes.

Using a Text Processor in the Text Mode

To demonstrate the creation of text, we will use a program written by Bill Sias in the magazine *Color Computer News* (Sept./Oct., 1981). Bill has granted us permission to use, discuss, and expand upon his program. Keep in mind that Bill did not intend this to be a finished product. He was merely showing capabilities of machine language programming in an educational setting.

The program uses the following three subroutines that reside in the Color Computer ROM area:

1. POLCAT—the keyboard scanning routine
2. PRINIT—the print to video screen routine
3. $A2BF—the output to printer routine

It is important to remember that the locations of subroutines in ROM may be changed in future Color Computer ROM versions. In that case, check your computer manuals for the locations that should be used.

Once again we used some equate instructions and saved some memory for some labeled data.

1. RMB—Reserve Memory Byte(s)—an operation that reserves the specified number of bytes in memory. The area is assigned the NAME specified.

   Examples

```
PRESS   RMB 1      ←saves 1 memory location (byte)
                    referred to as PRESS
TEXT    RMB $FF    ←saves 255 memory locations for the
                    data named TEXT
```

2. EQU—EQUate—an operation that assigns a specified name to the specified memory location. Names are easier to remember than numerical locations

   Examples

```
POLCAT  EQU $A1B1   ←the POLCAT subroutine is
                     located at $A1B1
PRINIT  EQU $A30A   ←the PRINIT subroutine is
                     located at $A30A
```

The program has been kept simple for ease of understanding. True editing is not allowed. Type carefully. This is not an actual word processor but works like an electric typewriter. You can correct characters on the screen but not in the text buffer.

Program 19—Word Processor

The program is broken up into functional parts so that you can closely examine each function.

Part 1—Set Up

```
0001  POLCAT  EQU $A1B1  KEYBOARD SCAN
0002  PRINIT  EQU $A30A  PRINT ROUTINE
0003  PRESS   RMB 1      # OF KEYS PRESSED
0004  TEXT    RMB $FF    TEXT BUFFER
```

The first two lines equate (EQU) the addresses of the keyboard scan routine ($A1B1) to the name POLCAT and the address of the screen print routine ($A30A) to the name PRINIT. From this point on, the addresses may be referred to by their names instead of their addresses.

Line 3 saves one memory location, which will be referred to by the name PRESS. The number of key presses that you make will be accumulated at this location.

Line 4 saves 255 bytes ($FF) for the text buffer, called TEXT. This is an area of memory that will be used to save your text so that it can be sent to the printer at a later point in the program.

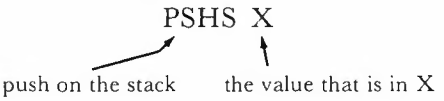
Part 2—Get Ready

```
0005  ZERO  CLR PRESS   START AT 0
0006        LDX #TEXT   GET BUFFER ADDR
0007        PSHS X      SAVE ON STACK
```

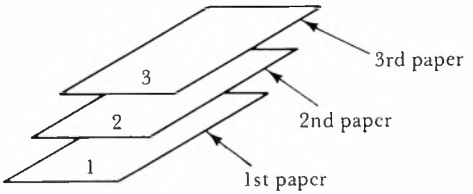
Line 5 places a zero in the area called PRESS. In other words, start with a count of zero. You haven't typed any text yet.

Line 6 puts the address of the beginning of the text buffer into register X. This is where the text that you type will be stored.

Line 7 saves the data that was just placed in register X onto the USER STACK. X is used in the ROM's keyboard scan routine. Therefore, you must save the text buffer location while the subroutine is being performed. A new instruction is used:



The 6809 has two stack areas. One is called the hardware stack. It is used by the computer in keeping track of its many chores. The second stack area is called the USER STACK. It is used by the programmer to store values for later retrieval. It is like a Last-In, First-Out file system. You might think of the stack as a pile of papers. You add to the pile by placing a paper on top. You remove papers from the file, one at a time, from the top.



In this sketch, the third paper must be removed before the second; the second must be removed before the first. In other words, papers are removed in the reverse order from the way they were put on the stack.

The instruction (PSHS X) at line 7 puts the value in the X register on the top of the stack. The data will be retrieved in Part 3.

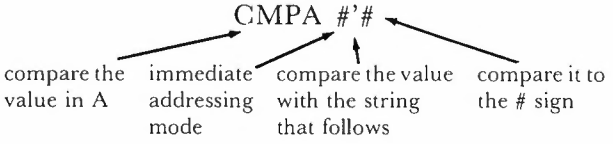
Part 3—Begin

This is the heart of the program. It takes the data typed, saves it in the text buffer, and puts it on the video screen. The number of keystrokes that you make is accumulated.

```
0008  BEGIN  JSR POLCAT   GET A KEY
0009         CMPA #'#     CONTROL CODE?
0010         BEQ PRINT    YES! DO IT.
0011         PULS X       GET BUFFER ADDR
0012         STA ,X+      SAVE AND UPDATE
0013         PSHS X       SAVE NEW POINTER
0014         INC PRESS    UPDATE # OF CHARS
0015         JSR PRINIT   PUT ON SCREEN
0016         BRA BEGIN    DO IT AGAIN
```

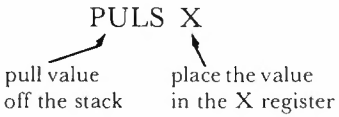
Line 8 jumps to a ROM subroutine to scan the keyboard to see if you have pressed a key. When a key is pressed, the computer goes on to the next line.

Line 9 compares the key pressed with the ASCII code for the # sign. When you are through entering text, type the # key (SHIFT 3). Line 9 will discover if there is a match by the CMPA instruction.



Line 10 causes a branch to the PRINT routine if the result of the comparison in line 9 is equal to zero, that is, when the # key is pressed. If the # key has not been pressed, the program continues at line 11.

Line 11 pulls the value of the buffer address off the stack. That value was previously placed there at line 7. The value is pulled off the stack and placed in the X register by the new instruction:



Line 12 stores the code of the last keystroke (which is in accumulator A) into the memory location stored in X. The X register is then incremented.

Line 13 pushes the new value held by the X register onto the user stack.

Line 14 increments the keystroke count held in the location called PRESS. Thus, this memory location keeps track of how many keystrokes have been made.

Line 15 jumps to the subroutine (PRINIT) that puts the text character on the screen. Thus, the text is displayed on the screen one character at a time.

Line 16 causes a branch back to BEGIN to scan the keyboard again.

Part 4—Print

This section sends the text that has been stored in the text buffer out to a printer.

```
0017  PRINT  LDX #TEXT   GET BUFFER
0018         LDB PRESS   THIS MANY
0019  LOOP   LDA ,X+     PUT CHAR IN A
0020         PSHS B,X    SAVE BOTH
0021         JSR #A2BF   PRINT#-2
0022         PULS B,X    RESTORE B & X
0023         DECB        CHARS LEFT
0024         BNE LOOP    IF B>0 DO AGAIN
0025         LDA #$D     GET CR
0026         JSR $A2BF   PRINT#-2
0027         SWI         GO BACK TO SDS80C
```

Line 17 loads the X register with the beginning location of the text buffer (where the text is stored).

Line 18 loads accumulator B with the value in PRESS (which contains the total number of keys pressed in creating the text file).

Line 19 is the beginning of the loop that sends the data in the text buffer to the printer. It loads a character into accumulator A from the memory location whose address is in the X register. It then increments X to point at the next character.

Line 20 saves both registers B and X on the stack in preparation for a subroutine that will use these registers.

Line 21 jumps to the subroutine that sends the character in accumulator A to the printer.

Line 22 pulls the values off the stack that were saved at line 20. They are placed back in the B and X registers.

Line 23 decrements accumulator B. Thus each time a character is printed, the count in B is decreased by one.

Line 24 causes a branch back to LOOP (line 19) to get another character if accumulator B did not contain a zero after being decre-

mented in line 23. When all the text has been printed, the value in B will be zero, and the computer will go on to line 25.

Line 25 loads accumulator A with $D, the ASCII code for a carriage return.

Line 26 goes to the subroutine that sends the carriage return to the printer.

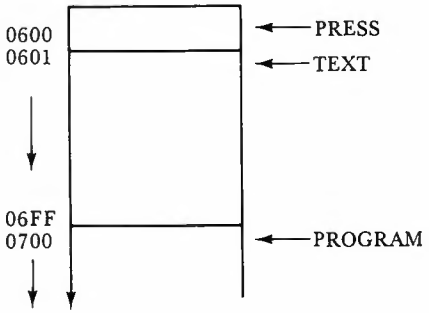
Line 27 is the software interrupt that sends the computer back to the monitor.

A listing of the program is shown in Figure 8-1.

```
0001 0600             POLCAT EQU $A1B1     KEYBOARD SCAN
0002 0600             PRINIT EQU $A30A     PRINT ROUTINE
0003 0600             PRESS  RMB 1         # OF KEY PRESSES
0004 0601             TEXT   RMB $FF       TEXT BUFFER
0005 0700 7F0600      ZERO   CLR PRESS     START AT 0
0006 0703 8E0601             LDX #TEXT     GET BUFFER ADR
0007 0706 3410               PSHS X        SAVE ON STACK
0008 0708 BDA1B1      BEGIN  JSR POLCAT    GET A KEY
0009 070B 8123               CMPA #'#      CONTROL CODE?
0010 070D 270E               BEQ PRINT     YES! DO IT
0011 070F 3510               PULS X        GET BUFFER ADDR
0012 0711 A780               STA ,X+       SAVE AND UPDATE
0013 0713 3410               PSHS X        SAVE NEW POINTER
0014 0715 7C0600             INC PRESS     UPDATE # CHARS
0015 0718 BDA30A             JSR PRINIT
0016 071B 20EB               BRA BEGIN
0017 071D 8E0601      PRINT  LDX #TEXT     GET BUFFER
0018 0720 F60600             LDB PRESS     THIS MANY
0019 0723 A680        LOOP   LDA ,X+       PUT CHAR IN A
0020 0725 3414               PSHS B,X      SAVE BOTH
0021 0727 BDA2BF             JSR $A2BF     PRINT "-2
0022 072A 3514               PULS B,X      RESTORE B & X
0023 072C 5A                 DECB
0024 072D 26F4               BNE LOOP
0025 072F 860D               LDA #$D       GET CR
0026 0731 BDA2BF             JSR $A2BF     PRINT #-2
0027 0734 3F                 SWI           GO BACK TO SDS80C
0028 0735                    END

BEGIN  0708  LOOP   0723  POLCAT A1B1  PRESS  0600
PRINIT A30A  PRINT  071D  TEXT   0601  ZERO   0700
```

Figure 8-1. Program 19—Word Processor #1

We will use the program to describe itself. The printer is turned on and set to the top of a page before beginning. Read the text before entering it in the word processor. Our results are shown in Figure 8-2. Be careful that you send the text to the printer before the buffer is full. The text buffer is reserved in the area that precedes the program.



If the text buffer exceeds 255 bytes, the first lines of the program will be written over and the program will cease to function. This feature will be corrected in the next version of the program.

```
USING THE WORD PROCESSOR

   Iff you want the printed
material to be the same width
as shown on the screen, be sure
to press RETURN before the text
wraps around from one line to
the next.  The computer dis-

PLAYS ONLY 32
CHARACTERS per line.  Your
printer probalby prints 80 or
132 characters per line.
   You should notice that only
255 bytes have been reserved
for the text buffer.  This
means that you will only ben
able to fill about
half of the video screen before
sending text to the printer.
   THEREfore, type only about 7
or 8 nlines.  Then send the results
to the printer.  The
program will return to pabugi
after the printer is finished.
The program has a carriage
return so that the printer
will be ready for a new line.
When control is returned to ABUG
the program may be RUN again to
asdd to the previous text.n
```

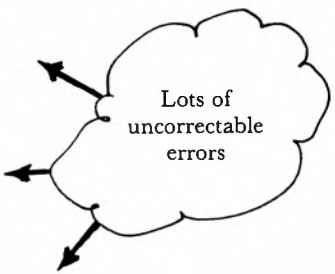


Figure 8–2. Sample Text from Word Processor

You could enlarge the area reserved for the text buffer (RMB $FF). However, the memory location PRESS (used to count the keystrokes) and accumulator B (used in the print routine to count the characters printed) each hold a maximum value of 255. You would have to use some other method to count a larger value. You could substitute index register Y because it is not used in the program. The Y register holds 16 bits and could handle a value up to, and including, 65535.

Use the program as it is for awhile. See if there are any modifications that you want to make. Then we'll show a few modifications that we have tried.

Adding Backspace to the Word Processor

For our first addition to the word processing program, we'll add the capability of modifying the word count to take care of minor typing corrections. As Bill Sias suggested in his article, this modification is a simple matter of adding some code to test for a backspace and to reduce the pointer to the text buffer by one when a backspace occurs. In addition, we have added a feature that will cause the computer to go to the print routine when the text buffer is full.

One addition is made to part 2 and several additions and changes are made to part 3.

Part 2—Modified Get Ready

```
        ZERO    CLR PRESS
                INC PRESS       ←added
                LDX #TEXT
                PSHS X
        BEGIN   JSR POLCAT  }
                CMPA #'#    }   as before
                BEQ PRINT   }
                PULS X
added           CMPA #8         a backspace?
                BNE ON          if not, go on
                LEAX -1,X       move back 1 in buffer
                DEC PRESS       decrease char. count
                PSHS X          save X
                JSR PRINIT      backspace on video
                BRA BEGIN       get new character
```

```
ON      STA ,X+   \
        PSHS X    |
        JSR PRINIT | changed
        INC PRESS |
        BNE BEGIN /
```

```
0001 0600                POLCAT EQU $A1B1
0002 0600                PRINIT EQU $A30A
0003 0600                PRESS  RMB 1
0004 0601                TEXT   RMB $FF
0005 0700 7F0600         ZERO   CLR PRESS
0006 0703 7C0600                INC PRESS
0007 0706 8E0601                LDX #TEXT
0008 0709 3410                  PSHS X
0009 070B BDA1B1         BEGIN  JSR POLCAT
0010 070E 8123                  CMPA #'#
0011 0710 271E                  BEQ PRINT
0012 0712 3510                  PULS X
0013 0714 8108                  CMPA #8
0014 0716 260C                  BNE ON
0015 0718 301F                  LEAX -1,X
0016 071A 7A0600                DEC PRESS
0017 071D 3410                  PSHS X
0018 071F BDA30A                JSR PRINIT
0019 0722 20E7                  BRA BEGIN
0020 0724 A780           ON     STA ,X+
0021 0726 3410                  PSHS X
0022 0728 BDA30A                JSR PRINIT
0023 072B 7C0600                INC PRESS
0024 072E 26DB                  BNE BEGIN
0025 0730 8E0601         PRINT  LDX #TEXT
0026 0733 F60600                LDB PRESS
0027 0736 A680           LOOP   LDA ,X+
0028 0738 3414                  PSHS B,X
0029 073A BDA2BF                JSR $A2BF
0030 073D 3514                  PULS B,X
0031 073F 5A                    DECB
0032 0740 26F4                  BNE LOOP
0033 0742 860D                  LDA #$D
0034 0744 BDA2BF                JSR $A2BF
0035 0747 3F                    SWI
0036 0748                       END

BEGIN  070B  LOOP   0736  ON     0724  POLCAT A1B1
PRESS  0600  PRINIT A30A  PRINT  0730  TEXT   0601
ZERO   0700
```

Figure 8-3. Program 20—Word Processor with Back Space

A sample of text produced with the previous modification follows:

USING THE WORD PROCESSOR

Now, when you make a mistake, you can back up and correct it. However, you will have to type in each character that you have backspaced over.

When the character buffer is full, the print routine is automatically ca
called—as just happened.

This is better than before, but still not perfect. The automatic break left a partially completed word.

When the modified program is used, you can now make corrections to the text buffer as well as the screen. The POLCAT subroutine gets the keystroke. If it is a backspace, it is as follows:

LEAX -1,X	subtracts 1 from the buffer address in register X
DEC PRESS	subtracts 1 from the keystroke count in the memory location labeled PRESS
BRA BEGIN	returns to POLCAT subroutine to get the correct keystroke, or another backspace, or the control code to send results to the printer

Suppose that you had typed in:

SOMETIMES ERROT

The computer has now stored

0600 PRESS	0F
0601 TEXT	53

0602	4F
0603	4D
0604	45
0605	54
0606	49
0607	4D
0608	45
0609	53
060A	20
060B	45
060C	52
060D	52
060E	4F
060F	54
X REGISTER	0610

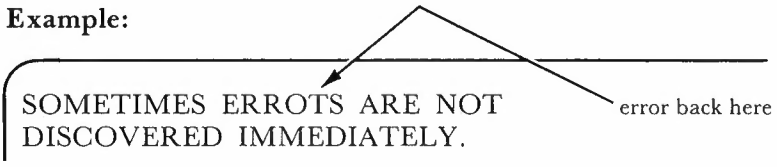
After the backstroke is pressed, the added code changes these

0600 PRESS	0E
X REGISTER	060F

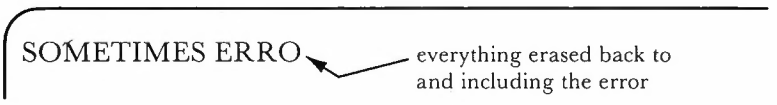
The ASCII code of the new keystroke will then replace the value at 060F, increase PRESS by one, and increase the value in the X register by one.

If the error is not immediately discovered, you may have to make several backspaces and retype everything after the first correction.

Example:



To correct, backspace until you see:



Now, retype the correction and all text that followed it.

You can see that our modification still does not allow complete editing. If desired, you could add features that would allow you to move a cursor backward and forward without erasing the text. Only the errors would be modified.

Enlarging the Text Buffer

To enlarge the text buffer of the word processor, you must change the method of counting the keystrokes and the method of recalling the text. We'll use the Y register in the next modification (which will be our last) to accomplish these changes.

The following changes will be made to the listing shown in Figure 8-3.

PRESS RMB 1	will be eliminated
TEXT RMB $1FF	changed to allow a full screen of text
LDY #0	will replace CLR PRESS
LEAY -1,Y	will replace DEC PRESS
LEAY 1,Y	will replace INC PRESS
LDB PRESS	will be eliminated
PSHS Y,X	will replace PSHS B,X
PULS Y,X	will replace PULS B,X
LEAY -1,Y	will replace DEC B

A listing of the modified program is shown in Figure 8-4. Further increases to the size of the text buffer may be made if you wish. To do so, change the following line of the program:

```
TEXT RMB $.....
```

any amount that will fit in your computer

Using Word Processor #3, we entered and printed the following:

```
0001 0600                POLCAT EQU $A1B1
0002 0600                PRINIT EQU $A30A
0003 0600                TEXT   RMB $1FF
0004 07FF 108E0000       ZERO   LDY #0
0005 0803 3121                  LEAY 1,Y
0006 0805 8E0600                LDX #TEXT
0007 0808 3410                  PSHS X
0008 080A BDA1B1         BEGIN  JSR POLCAT
0009 080D 8123                  CMPA #'#
0010 080F 271C                  BEQ PRINT
0011 0811 3510                  PULS X
0012 0813 8108                  CMPA #8
0013 0815 260B                  BNE ON
0014 0817 301F                  LEAX -1,X
0015 0819 313F                  LEAY -1,Y
0016 081B 3410                  PSHS X
0017 081D BDA30A                JSR PRINIT
0018 0820 20E8                  BRA BEGIN
0019 0822 A780           ON     STA ,X+
0020 0824 3410                  PSHS X
0021 0826 BDA30A                JSR PRINIT
0022 0829 3121                  LEAY 1,Y
0023 082B 20DD                  BRA BEGIN
0024 082D 8E0600         PRINT  LDX #TEXT
0025 0830 A680           LOOP   LDA ,X+
0026 0832 3430                  PSHS Y,X
0027 0834 BDA2BF                JSR $A2BF
0028 0837 3530                  PULS Y,X
0029 0839 313F                  LEAY -1,Y
0030 083B 26F3                  BNE LOOP
0031 083D 860D                  LDA #$D
0032 083F BDA2BF                JSR $A2BF
0033 0842 3F                    SWI
0034 0843                       END

BEGIN  080A  LOOP   0830  ON     0822  POLCAT A1B1
PRINIT A30A  PRINT  082D  TEXT   0600  ZERO   07FF
```

Figure 8-4. Program 21—Word Processor #3

USING THE WORD PROCESSOR

You now have a much larger buffer. Therefore, you do not have to worry about filling up the buffer. It will hold a screen full of characters.

You should still be careful about wrapping around a line of text. Keep your eyes on the screen as you are typing and everything should be all right.

This is the last demonstration before moving on to a new subject.

Maybe you can now use this as a letter writer.

Yours truly,

Don and Kurt Inman

Creating Text Characters

It is often desirable to display text characters when you are in a graphics mode. You might want to label certain figures that have been drawn, or you might want to ask for a response in a game that uses graphics. Other occasions will no doubt arise where you would like to mix text and graphics.

To display text characters along with graphics, you can draw the characters just as you would draw a graphics figure. In order to keep a standard size, we have arbitrarily chosen a 9 by 8 grid for each character.

We have used the highest resolution graphics made (6C) to demonstrate how an alphabet can be created. In this mode, one byte of data defines eight graphics elements. Thus a single byte can define one row of a text character. Nine bytes can define the 9 by 8 grid of the complete character.

For example, an A might be

X	X	X	X	X	X	X	X
X	X					X	X
X	X					X	X
X	X	X	X	X	X	X	X
X	X					X	X
X	X					X	X
X	X					X	X

1st row	00	or	00000000
2nd row	FF	or	11111111
3rd row	C3	or	11000011
4th row	C3	or	11000011
5th row	FF	or	11111111
6th row	C3	or	11000011
7th row	C3	or	11000011
8th row	C3	or	11000011
9th row	00	or	00000000

This would leave blank rows at the top and the bottom.

To put a letter on the screen, you would have to define the address where you wanted to put the upper left corner of the character and then add the correct increment to that address for each additional row.

In Chapter 4, you learned that the resolution of mode 6C is 128 by 192. Our characters are four elements wide; therefore we can put 128 ÷ 4 or 32 characters on each line. We could squeeze in 192 ÷ 9 or 21 lines of text on the screen.

The data for the characters could be put in a data table and selected by the computer as you type in the character from the keyboard. In Chapter 10, we will show how to put such a table on an EPROM (Erasable Programmed Read Only Memory) and insert the EPROM in a cartridge so that it would always be available for use in a program.

Figure 8-5 shows the characters that we will use, and Table 8-1 shows the data bytes used to draw the characters.

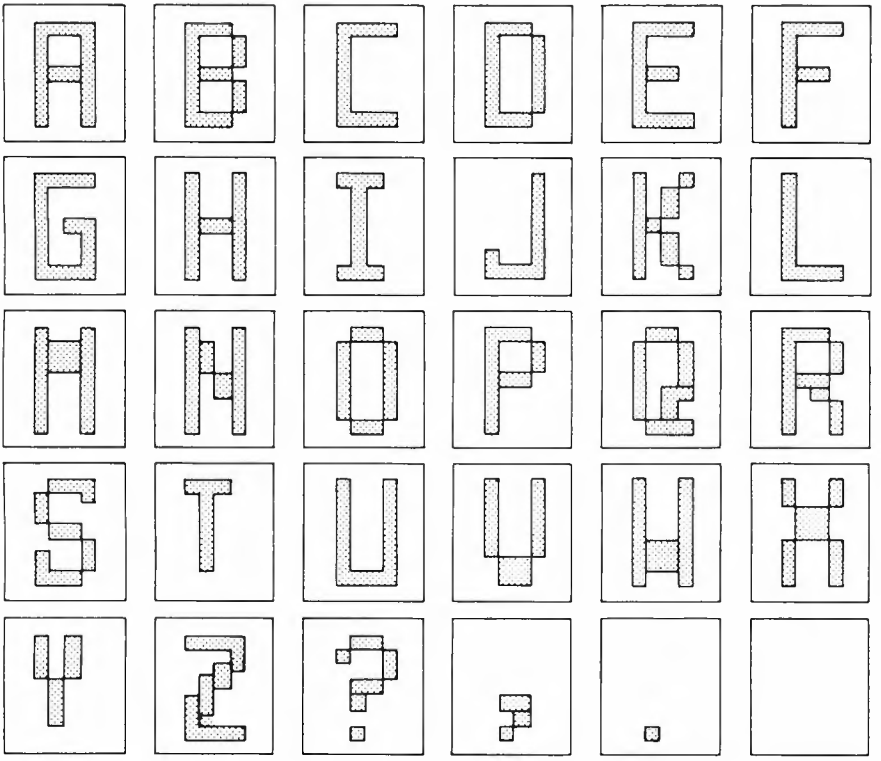


Figure 8-5. Text Characters

Table 8-1. Data Bytes for Text Characters

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
FF	FC	FF	FC	FF	FF	FF	C3	FC	03	C3	C0	C3	C3	3C
C3	C3	C0	C3	C0	C0	C0	C3	30	03	CC	C0	FF	F3	C3
C3	C3	C0	C3	C0	C0	C0	C3	30	03	CC	C0	FF	F3	C3
FF	FC	C0	C3	FC	FC	CF	FF	30	03	F0	C0	C3	CF	C3
C3	C3	C0	C3	C0	C0	C3	C3	30	03	CC	C0	C3	CF	C3
C3	C3	C0	C3	C0	C0	C3	C3	30	C3	CC	C0	C3	C3	C3
C3	FC	FF	FC	FF	C0	FF	C3	FC	FF	C3	FF	C3	C3	3C
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

P	Q	R	S	T	U	V	W	X	Y	Z	?	,	.	Sp.
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
FC	3C	FC	3F	FC	C3	C3	C3	C3	CC	FF	3C	00	00	00
C3	C3	C3	C3	30	C3	C3	C3	C3	CC	03	C3	00	00	00
C3	C3	C3	C0	30	C3	C3	C3	3C	CC	0C	03	00	00	00
FC	C3	FC	3C	30	C3	C3	C3	3C	30	3C	3C	00	00	00
C0	CF	CC	03	30	C3	C3	FF	C3	30	30	30	00	00	00
C0	CC	C3	C3	30	C3	3C	FF	C3	30	C0	00	3C	00	00
C0	3F	C3	FC	30	FF	3C	C3	C3	30	FF	30	0C	03	00
00	00	00	00	00	00	00	00	00	00	00	00	30	00	00

In Chapter 9, we'll go into more detail on how to use the text characters from a table.

Displaying Text in a Graphics Mode

The next program shows how to place a word on the screen in a specified location when you are in a graphics mode. The letters, TEXT (as shown in Figure 8-5), are drawn using the data bytes from Table 8-1.

Program 20—Displaying Text

Part 1—Set up Graphics

```
ORG  $1E00
LDA  #$E8
STA  $FF22
STA  $FFC3
STA  $FFC5
```

Select 6C, color set 1

Part 2—Clear Screen

```
        CLRA
        CLRB
        LDX #$400
LOOP1   STD ,X++
        CMPX #$1C00
        BLO LOOP1
```

Part 3—Display

```
        LDX #$1268
        LDB #4
        LDA #9
        STA $1F00
        LDY #TABLE
DRAW    BSR LET
        LEAX $80,X
        LDA #9
        STA $1F00
        DECB
        BNE DRAW
```

Part 4—Letter Subroutine

```
LET     LDA ,Y+         load 1 byte
        STA ,X          display it
        LEAX $20,X      jump to next row
        DEC $1F00
        BNE LET
        RTS <---------- go back if not done
```

Part 5—Byte Table

```
TABLE   FCB 0,$FC,$30,$30,$30   } T
        FCB $30,$30,$30,0       }
        FCB 0,$FF,$C0,$C0,$FC   } E      orange
        FCB $C0,$C0,$FF,0       }        letters
        FCB 0,$C3,$C3,$3C,$3C   } X
        FCB $C3,$C3,$C3,0       }
        FCB 0,$FC,$30,$30,$30   } T
        FCB $30,$30,$30,0       }
        END
```

A listing of the source and object programs is given in Figure 8-6. Enter and run the program. Then study the chapter summary and try

```
0001 0600                        ORG $1E00
0002 1E00 86E8                   LDA ##$E8
0003 1E02 B7FF22                 STA $FF22
0004 1E05 B7FFC3                 STA $FFC3
0005 1E08 B7FFC5                 STA $FFC5
0006 1E0B 4F                     CLRA
0007 1E0C 5F                     CLRB
0008 1E0D 8E0400                 LDX #$400
0009 1E10 ED81          LOOP1    STD ,X++
0010 1E12 8C1C00                 CMPX #$1C00
0011 1E15 25F9                   BLO LOOP1
0012 1E17 8E1268                 LDX #$1268
0013 1E1A C604                   LDB #4
0014 1E1C 8609                   LDA #9
0015 1E1E B71F00                 STA $1F00
0016 1E21 108E1E53               LDY #TABLE
0017 1E25 8D1F          DRAW     BSR LET
0018 1E27 30890080               LEAX $80,X
0019 1E2B 8609                   LDA #9
0020 1E2D B71F00                 STA $1F00
0021 1E30 5A                     DECB
0022 1E31 26F2                   BNE DRAW
0023 1E33 BDA1B1        KEY      JSR $A1B1
0024 1E36 8123                   CMPA #'#
0025 1E38 26F9                   BNE KEY
0026 1E3A 8600                   LDA #0
0027 1E3C B7FF22                 STA $FF22
0028 1E3F B7FFC2                 STA $FFC2
0029 1E42 B7FFC4                 STA $FFC4
0030 1E45 3F                     SWI
0031 1E46 A6A0          LET      LDA ,Y+
0032 1E48 A784                   STA ,X
0033 1E4A 308820                 LEAX $20,X
0034 1E4D 7A1F00                 DEC $1F00
0035 1E50 26F4                   BNE LET
0036 1E52 39                     RTS
0037 1E53 007C101010   TABLE     FCB 0,$FC,$30,$30,$30
0038 1E58 10101000               FCB $30,$30,$30,0
0039 1E5C 007E40407C             FCB 0,$FF,$C0,$C0,$FC
0040 1E61 40407E00               FCB $C0,$C0,$FF,0
0041 1E65 0042422418             FCB 0,$C3,$C3,$3C,$3C
0042 1E6A 24424200               FCB $C3,$C3,$C3,0
0043 1E6E 007C101010             FCB 0,$FC,$30,$30,$30
0044 1E73 10101000               FCB $30,$30,$30,0
0045 1E77                        END

DRAW   1E25  KEY    1E33  LET    1E46  LOOP1  1E10
TABLE  1E53
```

Figure 8-6. Text by Graphics

the chapter test before going on to Chapter 9 to mix the text with graphics.

Summary

In this chapter, you learned to display text on the video screen when using machine language in two ways:

1. From the text mode, you used several versions of a simple word processor.
2. From the graphics mode, you used graphic techniques to draw the characters.

You started with a very simple version of a word processor that used three of the Color Computer's ROM subroutines:

1. POLCAT—the keyboard scan routine
2. PRINIT—the print to video screen routine
3. $A2BF—the output to printer routine

Two pseudo-operations that were introduced earlier were also used in the word processor program:

PRESS RMB nn	(Reserve Memory Bytes), which reserves nn bytes or memory locations for data. A reference to this area is made by using the word PRESS.
NAME EQU $xxxx	(Equate), which assigns NAME to an area of memory beginning at location $xxxx. Any future reference to that location may be made using NAME. (Names are easier to remember than numbers.)

The memory location used for PRESS was set to zero by the instruction:

```
CLEAR PRESS
```

Two stack instructions were used:

PSHS X—push the value in register X onto the stack
PULS X—pull the top value off the stack and place it in register X

The stack is an area in memory where values can be temporarily stored. Each new piece of data, when placed on the stack, goes on the

top. Pieces of data are removed in the opposite order than when placed on the stack. In other words, data is removed from the top of the stack. The last piece on will be the first piece off.

A sample page of text was provided to allow you to test the simple word processor (Program 20). Each section (255 bytes or less) was sent to the printer as it was completed.

Word Processor #2 added a backspace feature so that you could back up and correct typing errors. The character count was decremented to keep a true count of characters actually used. This feature was implemented by the two instructions:

LEAX -1,X—subtracts one from the buffer address in X
DEC PRESS—subtracts one from the keystroke count in PRESS

Also added was a feature to dump the buffer to the screen when it filled with 255 bytes.

Word Processor #3 added features to allow extension of the text buffer and a keystroke count beyond 255 characters. Register Y, which holds 16 bits, was used to hold a keystroke count of up to 65,535. Allowance was also made to enlarge the text buffer to hold a full screen of text.

A method of drawing text characters was given so that you could display text characters when in a graphics mode. Data bytes were given that would create alphabetic characters on a 9 by 8 grid. Each data byte drew one of 9 rows for a given character. Drawings for a question mark, a comma, a period, and a blank space were added to the 26 alphabetic characters.

Chapter Test

1. Describe what each of these ROM subroutines do.
   a. PRINIT _____
   _____
   b. POLCAT _____
   _____
2. At what address is the ROM subroutine that sends the computer's output to a printer located? $_____
3. Explain why the pseudo-operation EQU is used. _____
   _____
   _____
4. What precaution must be observed when using programs that call a subroutine from ROM? _____

5. Give the assembler code for
   a. pushing the value in register Y onto the stack.

   b. pulling a value off the stack and placing it in the X register.

6. Describe the use of the user stack.

7. One byte of data was reserved (by PRESS RMB 1) in Word Processor #1 to count the keystrokes. This limited the amount of text that could be saved at one time to how many characters?

8. Due to the limitations in test exercise 7, about how many lines of text, shown on the screen, may be typed before the text buffer is full? about _____ lines.
9. Word Processor #2 included modifications to allow backspacing for a true character count. Changes were made by

   LEAX -1,X and DEC PRESS

   What is the function of
   a. LEAX -1,X?

   b. DEC PRESS?

10. What was the purpose, in Word Processor #3, of replacing register B by register Y?

Answers to Odd-Numbered Exercises in Chapter Test

1. a. a routine that outputs data to the video screen
   b. a routine that scans the keyboard to see if a key has been pressed
3. EQU is used to assign names to areas of memory. You may then refer to that name rather than trying to remember its numerical location.

5. a. PSHS Y
   b. PULS X

7. One-byte (one memory location) can count up to 255 keystrokes (characters).

9. a. LEAX -1,X subtracts one from the buffer address in register X so that a correction will be made in the correct place in the text buffer.

   b. DEC PRESS subtracts one from the keystroke count in PRESS so that the number of keystrokes will be changed to the correct value.

Chapter 9

Graphics with Text

You learned to put a word of text on the screen when using a graphic mode in Chapter 8. We will go into more detail in this chapter using mode 6C so that you can add colorful graphics and some text.

You will learn to select the data for a given text character from a table contained in your program. Your program will use this data to draw the character at a selected position on the video screen. You can select any character in the table and place it wherever you want on the screen.

The programming involved contains quite a bit of detail. Therefore, this chapter is limited to the description of two programs. The first program uses preselected screen positions for displaying a preselected message. You would have to modify the program if you wanted to change the message or the screen position where it is displayed. The second program allows you to select the number of characters in your message, the characters to be displayed, and the placement of the characters on the screen. All this is accomplished from the keyboard.

Because the second program is a modification of the first program, it is advisable to save the first source program on tape or disk (if you have a disk).

Quite a bit of planning is necessary for this technique, and several changes would have to be made if you wanted to change graphics modes. In designing the program, you must consider the following:

- the size and shape for your characters
- the relationship of the characters to the screen resolution being used

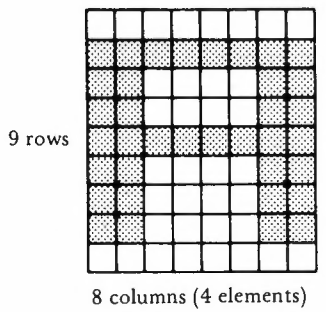
- the method to select the desired character
- the method of placing the character on the screen
- how to make the computer do the detailed calculations
- how to make the program easy to use, yet as versatile as possible

Planning the Display

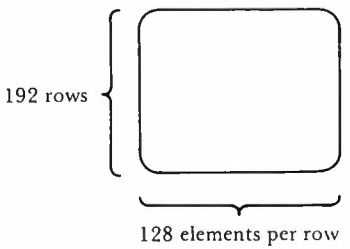
We will use a four-color mode so that the characters will be orange. The background will be buff. Mode 6C, with color set 1, will be used because it is the highest resolution four-color mode. If the mode or color of the characters is changed, the data in the character table should be changed accordingly.

The table of data values for our characters (shown in Table 8-1) will display a graphic block of eight columns and nine rows on the screen.

Example
The character A



The resolution of the screen is 256 by 192. Because each element in mode 6C is two columns wide, 128 elements will fit on each row.



Therefore, you could fit approximately 21 rows of 32 characters per row on the screen. This is derived from the size of the graphics elements in mode 6C

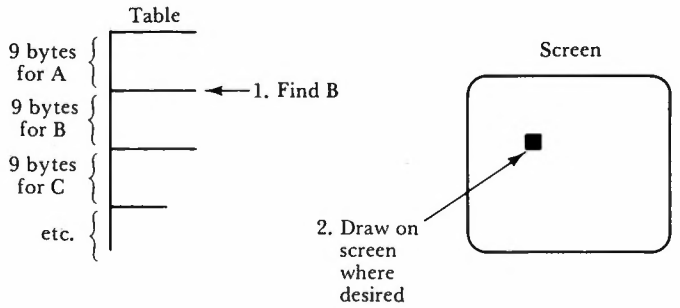
9 elements (rows) and 4 elements for each character.

```
   32 characters/row        21 rows
4)128                    9)192
 12                       18
  8                        12
  8                         9
  0                         3
```

For our purposes, we will provide a blank space between characters. Therefore, we will restrict the number of characters to 16 per row. The data for the characters can be placed in a table, as before.

Your main concern in writing a program consists of two parts. First, the computer must select the desired character from the data table. Then, it must draw it in the desired location on the screen.

Example



Selecting a Character

When you use a program to draw a character, it would be desirable to find some systematic way for the computer to find the desired character. Study the ASCII codes (in hex form) for the characters used in our data table (Table 9-1).

Table 9-1. ASCII Codes for Characters

Letter	*Code*	*Letter*	*Code*	*Letter*	*Code*
A	41	K	4B	U	55
B	42	L	4C	V	56
C	43	M	4D	W	57
D	44	N	4E	X	58
E	45	O	4F	Y	59
F	46	P	50	Z	5A
G	47	Q	51	?	3F
H	48	R	52	,	2C
I	49	S	53	.	2E
J	4A	T	54	space	20

You can see that the codes for the letters A–Z follow each other in regular order. The punctuation and space codes are the only ones that are out of order.

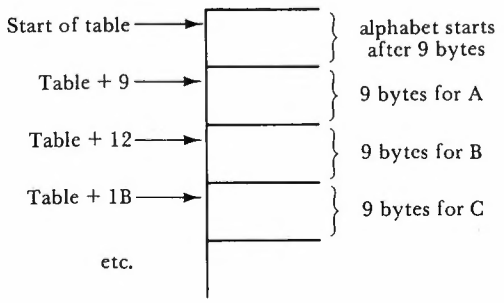
Because each character will occupy nine bytes in the data table, we will use a bit of mathematics to select the correct data. The arithmetic involved uses the ASCII code of the character and will be performed by the computer in the following way:

OFFSET IN TABLE = (ASCII CODE − 40) * 9

Examples (in hex) using this relationship

A = (41-40)*9 = 1*9 = 9
B = (42-40)*9 = 2*9 = 12
C = (43-40)*9 = 3*9 = 1B
Z = (5A-40)*9 = 1A*9 = EA

In the table



To select the correct character, you could use the instructions

```
LDD #OFFSET     to find character in table
ADDD #TABLE     add start of table
TFR D,U         let U point to character
LDA, U+         get code for 1 line
STA SCREEN      draw 1 line of character
```

A loop would put nine lines on the screen in successive rows to form the character. We'll discuss this in more detail later in the chapter.

Placing the Character on the Screen

The screen, in mode 6C, contains 128 elements in each row starting at memory location $600 in the upper left corner. Each byte of data fills a block four elements wide. Because we are providing a space between characters, 16 characters are displayed on each line. Each character (with space) fills two screen locations, and each screen line is 32 memory locations wide. The characters occupy 9 lines. Therefore, the first row of characters occupies memory locations $600-71F as shown in Figure 9-1.

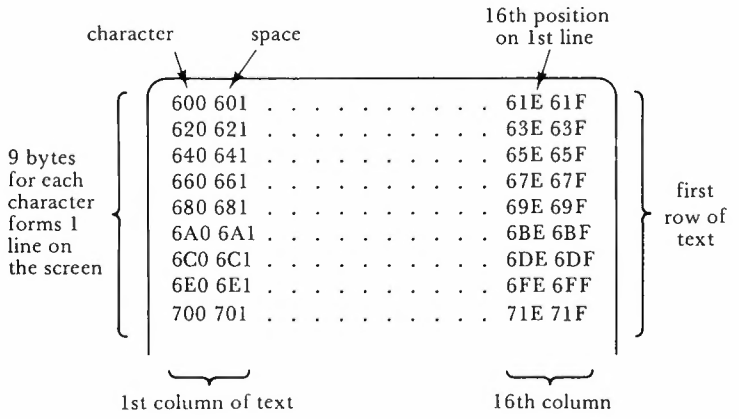


Figure 9-1. Screen Memory for Characters in Row 1

The memory that will be used for the complete screen is shown in Table 9-2. We will refer to the placement of characters by row and column in hex notation. Now let's try to put all this together in a program.

Table 9-2. Screen Memory for 21 Character Rows

	Column 01 – 10	
Row		
01	600	71F
02	720	83F
03	840	95F
04	960	A7F
05	A80	B9F
06	BA0	CBF
07	CC0	DDF
08	DE0	EFF
09	F00	101F
0A	1020	113F
0B	1140	125F
0C	1260	137F
0D	1380	149F
0E	14A0	15BF
0F	15C0	16DF
10	16E0	17FF
11	1800	191F
12	1920	1A3F
13	1A40	1B5F
14	1B60	1C7F
15	1C80	1D9F

Program 23—Orange Text by Graphics

We chose mode 6C with color set 1 for a simple program that will put two lines of text on the screen. Two data tables are used. TABLE1 contains the data used to draw the characters. TABLE2 contains values that supply the offsets necessary to find the characters to be displayed from TABLE1.

The program displays the following:

As you might guess, the background of the screen is buff and the color of the letters is orange. If we had used color set zero, the message would be the same but the letters would have been red on a green background. You might want to try changing the data to match the colors for color set zero.

We have used several pointers: X points to the screen location where the character will be displayed; Y points to the TABLE2 offset values; and U points to the data in TABLE1, which are used to draw the characters.

We'll revise Program 23 later in this chapter to make it more useful in drawing any desired character in TABLE1. Therefore, you should save the source program so that it can be modified at that time.

```
0001 0600                SCREE1 EQU $1024
0002 0600                SCREE2 EQU $1268
0003 0600                VIDBEG EQU $600
0004 0600                POLCAT EQU $A1B1
0005 0600                       ORG VIDBEG+$1800
0006 1E00 86E8           START  LDA #$E8
0007 1E02 B7FF22                STA $FF22
0008 1E05 B7FFC3                STA $FFC3
0009 1E08 B7FFC5                STA $FFC5
0010 1E0B B7FFC7                STA $FFC7
0011 1E0E 4F                    CLRA
0012 1E0F 5F                    CLRB
0013 1E10 8E0600                LDX #VIDBEG
0014 1E13 ED81           LOOP1  STD ,X++
0015 1E15 8C1E00                CMPX #VIDBEG+$1800
0016 1E18 25F9                  BLO LOOP1
0017 1E1A 8E1024                LDX #SCREE1
0018 1E1D 108E1F98              LDY #TABLE2
0019 1E21 860B                  LDA #$B
0020 1E23 B71E88                STA COUNT
0021 1E26 4F             LOOP2  CLRA
0022 1E27 E6A0                  LDB ,Y+
0023 1E29 C31E8A                ADDD #TABLE1
0024 1E2C 1F03                  TFR D,U
0025 1E2E 8609                  LDA #9
0026 1E30 B71E89                STA LINE
0027 1E33 A6C0           LOOP3  LDA ,U+
0028 1E35 A784                  STA ,X
0029 1E37 308820                LEAX $20,X
0030 1E3A 7A1E89                DEC LINE
0031 1E3D 26F4                  BNE LOOP3
0032 1E3F 3089FEE2              LEAX -$11E,X
0033 1E43 7A1E88                DEC COUNT
0034 1E46 26DE                  BNE LOOP2
0035 1E48 8E1268                LDX #SCREE2
0036 1E4B 8607                  LDA #7
0037 1E4D B71E88                STA COUNT
```

```
0038 1E50 4F             LOOP4  CLRA
0039 1E51 E6A0                  LDB ,Y+
0040 1E53 C31E8A                ADDD #TABLE1
0041 1E56 1F03                  TFR D,U
0042 1E58 8609                  LDA #9
0043 1E5A B71E89                STA LINE
0044 1E5D A6C0           LOOP5  LDA ,U+
0045 1E5F A784                  STA ,X
0046 1E61 308820                LEAX $20,X
0047 1E64 7A1E89                DEC LINE
0048 1E67 26F4                  BNE LOOP5
0049 1E69 3089FEE2              LEAX -$11E,X
0050 1E6D 7A1E88                DEC COUNT
0051 1E70 26DE                  BNE LOOP4
0052 1E72 BDA1B1         LOOP6  JSR POLCAT
0053 1E75 8123                  CMPA #"#
0054 1E77 26F9                  BNE LOOP6
0055 1E79 8600                  LDA #0
0056 1E7B B7FF22                STA $FF22
0057 1E7E B7FFC2                STA $FFC2
0058 1E81 B7FFC4                STA $FFC4
0059 1E84 B7FFC6                STA $FFC6
0060 1E87 3F                    SWI
0061 1E88                COUNT  RMB 1
0062 1E89                LINE   RMB 1
0063 1E8A 0000000000     TABLE1 FCB 0,0,0,0,0
0064 1E8F 00000000               FCB 0,0,0,0
0065 1E93 00FFC3C3FF             FCB 0,$FF,$C3,$C3,$FF
0066 1E98 C3C3C300               FCB $C3,$C3,$C3,0
0067 1E9C 00FCC3C3FC             FCB 0,$FC,$C3,$C3,$FC
0068 1EA1 C3C3FC00               FCB $C3,$C3,$FC,0
0069 1EA5 00FFC0C0C0             FCB 0,$FF,$C0,$C0,$C0
0070 1EAA C0C0FF00               FCB $C0,$C0,$FF,0
0071 1EAE 00FCC3C3C3             FCB 0,$FC,$C3,$C3,$C3
0072 1EB3 C3C3FC00               FCB $C3,$C3,$FC,0
0073 1EB7 00FFC0C0FC             FCB 0,$FF,$C0,$C0,$FC
0074 1EBC C0C0FF00               FCB $C0,$C0,$FF,0
0075 1EC0 00FFC0C0FC             FCB 0,$FF,$C0,$C0,$FC
0076 1EC5 C0C0C000               FCB $C0,$C0,$C0,0
0077 1EC9 00FFC0C0CF             FCB 0,$FF,$C0,$C0,$CF
0078 1ECE C3C3FF00               FCB $C3,$C3,$FF,0
0079 1ED2 00C3C3C3FF             FCB 0,$C3,$C3,$C3,$FF
0080 1ED7 C3C3C300               FCB $C3,$C3,$C3,0
0081 1EDB 00FC303030             FCB 0,$FC,$30,$30,$30
0082 1EE0 3030FC00               FCB $30,$30,$FC,0
0083 1EE4 0003030303             FCB 0,3,3,3,3
0084 1EE9 03C3FF00               FCB 3,$C3,$FF,0
```

```
0085 1EED 00C3CCCCF0          FCB 0,$C3,$CC,$CC,$F0
0086 1EF2 CCCCC300            FCB $CC,$CC,$C3,0
0087 1EF6 00C0C0C0C0          FCB 0,$C0,$C0,$C0,$C0
0088 1EFB C0C0FF00            FCB $C0,$C0,$FF,0
0089 1EFF 00C3FFFFC3          FCB 0,$C3,$FF,$FF,$C3
0090 1F04 C3C3C300            FCB $C3,$C3,$C3,0
0091 1F08 00C3F3F3CF          FCB 0,$C3,$F3,$F3,$CF
0092 1F0D CFC3C300            FCB $CF,$C3,$C3,0
0093 1F11 003CC3C3C3          FCB 0,$3C,$C3,$C3,$C3
0094 1F16 C3C33C00            FCB $C3,$C3,$3C,0
0095 1F1A 00FCC3C3FC          FCB 0,$FC,$C3,$C3,$FC
0096 1F1F C0C0C000            FCB $C0,$C0,$C0,0
0097 1F23 003CC3C3C3          FCB 0,$3C,$C3,$C3,$C3
0098 1F28 CFCC3F00            FCB $CF,$CC,$3F,0
0099 1F2C 00FCC3C3FC          FCB 0,$FC,$C3,$C3,$FC
0100 1F31 CCC3C300            FCB $CC,$C3,$C3,0
0101 1F35 003FC3C03C          FCB 0,$3F,$C3,$C0,$3C
0102 1F3A 033CFC00            FCB 3,$3C,$FC,0
0103 1F3E 00FC303030          FCB 0,$FC,$30,$30,$30
0104 1F43 30303000            FCB $30,$30,$30,0
0105 1F47 00C3C3C3C3          FCB 0,$C3,$C3,$C3,$C3
0106 1F4C C3C3FF00            FCB $C3,$C3,$FF,0
0107 1F50 00C3C3C3C3          FCB 0,$C3,$C3,$C3,$C3
0108 1F55 C33C3C00            FCB $C3,$3C,$3C,0
0109 1F59 00C3C3C3C3          FCB 0,$C3,$C3,$C3,$C3
0110 1F5E FFFFC300            FCB $FF,$FF,$C3,0
0111 1F62 00C3C33C3C          FCB 0,$C3,$C3,$3C,$3C
0112 1F67 C3C3C300            FCB $C3,$C3,$C3,0
0113 1F6B 00CCCCCC30          FCB 0,$CC,$CC,$CC,$30
0114 1F70 30303000            FCB $30,$30,$30,0
0115 1F74 00FF030C3C          FCB 0,$FF,3,$C,$3C
0116 1F79 30C0FF00            FCB $30,$C0,$FF,0
0117 1F7D 003CC3033C          FCB 0,$3C,$C3,3,$3C
0118 1F82 30003000            FCB $30,0,$30,0
0119 1F86 0000000000          FCB 0,0,0,0,0
0120 1F8B 003C0C30            FCB 0,$3C,$C,$30
0121 1F8F 0000000000          FCB 0,0,0,0,0
0122 1F94 00000300            FCB 0,0,3,0
0123 1F98 87A2097E3F          FCB $87,$A2,9,$7E,$3F
0124 1F9D 2D00B42DD8          FCB $2D,0,$B4,$2D,$D8
0125 1FA2 B4877E00            FCB $B4,$87,$7E,0
0126 1FA6 12BD3636            FCB $12,$BD,$36,$36
0127 1FAA                     END

COUNT   1E88  LINE    1E89  LOOP1   1E13  LOOP2   1E26
LOOP3   1E33  LOOP4   1E50  LOOP5   1E5D  LOOP6   1E72
POLCAT  A1B1  SCREE1  1024  SCREE2  1268  START   1E00
TABLE1  1E8A  TABLE2  1F98  VIDBEG  0600
```

Figure 9-2. Program 23—Text by Graphics

How Program 23 Works

Four equate instructions are used to name

1. the beginning of video graphics (VIDBEG)
2. the beginning of the first line of text (SCREE1)
3. the beginning of the second line of text (SCREE2)
4. the ROM keyboard scan routine (POLCAT)

Graphics mode 6C is then selected, and the screen is cleared. The X register is set to point to screen memory, and the Y register is set to point to TABLE2 for the offsets. Each character is nine bytes high. One line of the character is drawn at a time (controlled by LINE). Eleven characters are drawn on the first line of text (controlled by COUNT):

ORANGE TEXT

Seven characters are drawn on the second line of text (also controlled by COUNT):

ON BUFF

After drawing the message, the computer loops at POLCAT so that you can read it. Press [SHIFT] [#] to exit from the program. The computer then returns to the text mode and the monitor.

The next step is to modify Program 23 so that it is more useful.

Selecting Characters from the Keyboard

The characters displayed in Program 23 were selected from TABLE1 by the offsets listed in TABLE2. In other words, the program selected the characters that were to be displayed. Because the offset for the alphabetic characters are readily calculated, the computer could do this task after you typed the desired letter on the keyboard.

We used the Y register to point to the correct offset in the previous program. This time, we will use the formula mentioned earlier:

$$\text{OFFSET} = (\text{ASCII}-40) * 9$$

This can be done by the following sequence of instructions:

```
SUBA #$40          subtract $40
STA HOLD           save the result
LSLA               shift left 3 times
LSLA               —this multiplies
LSLA               the result by 8
ADDA #HOLD         add original—gives original times 9
TFR A,B
CLRA               add offset to TABLE1
ADDD #TABLE1
```

The computer must also test for the special characters: the question mark, comma, period, and space because the ASCII codes for these characters do not fit the equation used for the alphabetic characters. If one of these characters is found, the computer branches to the appropriate place to make the necessary conversion.

```
KEY  JSR POLCAT
     BEQ KEY
     CMPA #$20     is it a space?
     BEQ ZERO      if so, go to ZERO
     CMPA #$2E     is it a period?
     BEQ DOT       if so, go to DOT
     CMPA #$2C     is it a comma?
     BEQ COMMA     if so, go to COMMA
     CMPA #$3F     is it a question mark?
     BEQ QUEST     if so, go to QUEST
     CMPA #$41     is code lower than A?
     BLO KEY       if so, bad entry—go back
     CMPA #$5A     is code higher than Z?
     BGT KEY       if so, bad entry—go back
```

These tests are performed before the use of the equation that converts the codes for the letters A-Z.

After the ASCII codes have been converted, the offset value is added to the address of the beginning of TABLE1 to find the correct data for drawing the character. The sum is transferred to U, which points to the data values as the character is drawn.

Placement of Characters on the Screen

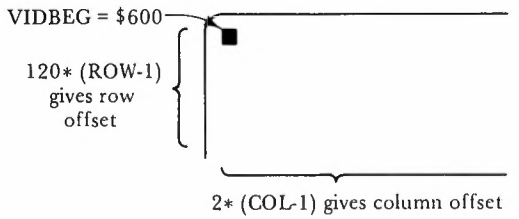
The X register is used as before to point to the screen memory where the characters are to be placed. In this program, you will make the

selection of the starting location for your message by stating a row and column value. The computer will calculate the memory location from your inputs. A typical calculation would be

```
CONVT   JSR POLCAT   get ten's digit
        BEQ CONVT    of the character
        AND #1       convert ASCII to 1 or 0
        LDB #$10
        MUL          multiply by 16 (10 hex)
        STB ROW      save the ten's digit
CONVU   JSR POLCAT   get one's digit
        BEQ CONVU
        CMPA #$41    if in range of 0-9
        BLO ON       go to ON
        CMPA #$46    if greater than F, bad input
        BGT CONVU    go back for another input
        SUBA #7      if A-F, subtract 7
ON      SUBA #$30    change ASCII to hex 0-F
        ADDA ROW     add ten's value to units value
```

When the computer is ready to put the character on the screen, the value in ROW will be placed in X to calculate the correct memory location from the equation

$$\text{LOCATION} = (\text{ROW}-1) * 120 + (\text{COL}-1) * 2 + \text{VIDBEG}$$



Message to Select Inputs

A series of messages, or prompts, will be put on the screen so that you can select a starting row, a starting column, and the number of characters that you want to display. The characters for these messages are accessed from TABLE2. The messages are displayed near the top of the screen at a location selected by

```
LDX #VIDBEG + $124
     ↑              ↖
   $600          1 row from top (120), 2 columns
                 from the left (2*2)
```

The listing of Program 24, Keyboard Graphic Text, is shown in Figure 9-3.

```
0001 0600                VIDBEG EQU $600
0002 0600                POLCAT EQU $A1B1
0003 0600                       ORG VIDBEG+$1800
0004 1E00 86E8           START  LDA #$E8
0005 1E02 B7FF22                STA $FF22
0006 1E05 B7FFC3                STA $FFC3
0007 1E08 B7FFC5                STA $FFC5
0008 1E0B B7FFC7                STA $FFC7
0009 1E0E 4F                    CLRA
0010 1E0F 5F                    CLRB
0011 1E10 8E0600                LDX #VIDBEG
0012 1E13 ED81           LOOP1  STD ,X++
0013 1E15 8C1E00                CMPX #VIDBEG+$1800
0014 1E18 25F9                  BLO LOOP1
0015 1E1A 8E0724                LDX #VIDBEG+$124
0016 1E1D 108E2077              LDY #TABLE2
0017 1E21 8604                  LDA #4
0018 1E23 B71F67                STA COUNT
0019 1E26 1700F8                LBSR DISPLA
0020 1E29 170118                LBSR CONVT
0021 1E2C B71F61                STA ROW
0022 1E2F 17011F                LBSR CONVU
0023 1E32 8030                  SUBA #$30
0024 1E34 BB1F61                ADDA ROW
0025 1E37 8001                  SUBA #1
0026 1E39 B71F61                STA ROW
0027 1E3C 8E0724                LDX #VIDBEG+$124
0028 1E3F 8604                  LDA #4
0029 1E41 B71F67                STA COUNT
0030 1E44 1700DA                LBSR DISPLA
0031 1E47 1700FA                LBSR CONVT
0032 1E4A B71F62                STA COL
0033 1E4D 170101                LBSR CONVU
0034 1E50 8030                  SUBA #$30
0035 1E52 BB1F62                ADDA COL
0036 1E55 8001                  SUBA #1
0037 1E57 B71F62                STA COL
0038 1E5A 8E0724                LDX #VIDBEG+$124
0039 1E5D 8605                  LDA #5
0040 1E5F B71F67                STA COUNT
0041 1E62 1700BC                LBSR DISPLA
0042 1E65 1700DC                LBSR CONVT
0043 1E68 B71F63                STA CHRS
0044 1E6B 1700E3                LBSR CONVU
```

```
0045 1E6E 8030                SUBA #$30
0046 1E70 BB1F63              ADDA CHRS
0047 1E73 B71F63              STA CHRS
0048 1E76 8E0724              LDX #VIDBEG+$124
0049 1E79 8605                LDA #5
0050 1E7B B71F67              STA COUNT
0051 1E7E 108E1F69            LDY #TABLE1
0052 1E82 17009C      ERASE   LBSR DISPLA
0053 1E85 4F                  CLRA
0054 1E86 F61F61              LDB ROW
0055 1E89 1F01                TFR D,X
0056 1E8B 4F                  CLRA
0057 1E8C 5F                  CLRB
0058 1E8D C30120      CALC    ADDD #$120
0059 1E90 301F                LEAX -1,X
0060 1E92 26F9                BNE CALC
0061 1E94 FD1F64              STD OFFSET
0062 1E97 4F                  CLRA
0063 1E98 F61F62              LDB COL
0064 1E9B 58                  LSLB
0065 1E9C F31F64              ADDD OFFSET
0066 1E9F C30600              ADDD #VIDBEG
0067 1EA2 1F01                TFR D,X
0068 1EA4 108E2077            LDY #TABLE2
0069 1EA8 B61F63              LDA CHRS
0070 1EAB B71F67              STA COUNT
0071 1EAE BDA1B1      KEY     JSR POLCAT
0072 1EB1 27FB                BEQ KEY
0073 1EB3 8120                CMPA #$20
0074 1EB5 2766                BEQ ZERO
0075 1EB7 812E                CMPA #$2E
0076 1EB9 275D                BEQ DOT
0077 1EBB 812C                CMPA #$2C
0078 1EBD 2754                BEQ COMMA
0079 1EBF 813F                CMPA #$3F
0080 1EC1 274B                BEQ QUEST
0081 1EC3 8141                CMPA #$41
0082 1EC5 25E7                BLO KEY
0083 1EC7 815A                CMPA #$5A
0084 1EC9 2EE3                BGT KEY
0085 1ECB 8040                SUBA #$40
0086 1ECD B71F66              STA HOLD
0087 1ED0 48                  LSLA
0088 1ED1 48                  LSLA
0089 1ED2 48                  LSLA
0090 1ED3 BB1F66              ADDA HOLD
0091 1ED6 1F89                TFR A,B
0092 1ED8 4F                  CLRA
0093 1ED9 C31F69      GO      ADDD #TABLE1
0094 1EDC 1F03                TFR D,U
0095 1EDE 8609                LDA #9
0096 1EE0 B71F68              STA LINE
0097 1EE3 A6C0        LOOP3   LDA ,U+
```

```
0098 1EE5 A784                STA ,X
0099 1EE7 308820              LEAX $20,X
0100 1EEA 7A1F68              DEC LINE
0101 1EED 26F4                BNE LOOP3
0102 1EEF 3089FEE2            LEAX -$11E,X
0103 1EF3 7A1F67              DEC COUNT
0104 1EF6 26B6                BNE KEY
0105 1EF8 BDA1B1      LOOP6   JSR POLCAT
0106 1EFB 8123                CMPA #'#
0107 1EFD 26F9                BNE LOOP6
0108 1EFF 8600                LDA #0
0109 1F01 B7FF22              STA $FF22
0110 1F04 B7FFC2              STA $FFC2
0111 1F07 B7FFC4              STA $FFC4
0112 1F0A B7FFC6              STA $FFC6
0113 1F0D 3F                  SWI
0114 1F0E C6F3        QUEST   LDB #$F3
0115 1F10 4F                  CLRA
0116 1F11 20C6                BRA GO
0117 1F13 C6FC        COMMA   LDB #$FC
0118 1F15 4F                  CLRA
0119 1F16 20C1                BRA GO
0120 1F18 CC0105      DOT     LDD #$105
0121 1F1B 20BC                BRA GO
0122 1F1D 4F          ZERO    CLRA
0123 1F1E 5F                  CLRB
0124 1F1F 20B8                BRA GO
0125 1F21 4F          DISPLA  CLRA
0126 1F22 E6A0                LDB ,Y+
0127 1F24 C31F69              ADDD #TABLE1
0128 1F27 1F03                TFR D,U
0129 1F29 8609                LDA #9
0130 1F2B B71F68              STA LINE
0131 1F2E A6C0        LOOP    LDA ,U+
0132 1F30 A784                STA ,X
0133 1F32 308820              LEAX $20,X
0134 1F35 7A1F68              DEC LINE
0135 1F38 26F4                BNE LOOP
0136 1F3A 3089FEE2            LEAX -$11E,X
0137 1F3E 7A1F67              DEC COUNT
0138 1F41 26DE                BNE DISPLA
0139 1F43 39                  RTS
0140 1F44 BDA1B1      CONVT   JSR POLCAT
0141 1F47 27FB                BEQ CONVT
0142 1F49 8401                ANDA #1
0143 1F4B C610                LDB #$10
0144 1F4D 3D                  MUL
0145 1F4E 1F98                TFR B,A
0146 1F50 39                  RTS
0147 1F51 BDA1B1      CONVU   JSR POLCAT
0148 1F54 27FB                BEQ CONVU
0149 1F56 8141                CMPA #$41
0150 1F58 2506                BLO ON
```

```
0151 1F5A 8146                CMPA #$46
0152 1F5C 2EF3                BGT CONVU
0153 1F5E 8007                SUBA #7
0154 1F60 39           ON     RTS
0155 1F61              ROW    RMB 1
0156 1F62              COL    RMB 1
0157 1F63              CHRS   RMB 1
0158 1F64              OFFSET RMB 2
0159 1F66              HOLD   RMB 1
0160 1F67              COUNT  RMB 1
0161 1F68              LINE   RMB 1
0162 1F69 0000000000   TABLE1 FCB 0,0,0,0,0
0163 1F6E 00000000             FCB 0,0,0,0
0164 1F72 00FFC3C3FF           FCB 0,$FF,$C3,$C3,$FF
0165 1F77 C3C3C300             FCB $C3,$C3,$C3,0
0166 1F7B 00FCC3C3FC           FCB 0,$FC,$C3,$C3,$FC
0167 1F80 C3C3CF00             FCB $C3,$C3,$CF,0
0168 1F84 00FFC0C0C0           FCB 0,$FF,$C0,$C0,$C0
0169 1F89 C0C0FF00             FCB $C0,$C0,$FF,0
0170 1F8D 00FCC3C3C3           FCB 0,$FC,$C3,$C3,$C3
0171 1F92 C3C3FC00             FCB $C3,$C3,$FC,0
0172 1F96 00FFC0C0FC           FCB 0,$FF,$C0,$C0,$FC
0173 1F9B C0C0FF00             FCB $C0,$C0,$FF,0
0174 1F9F 00FFC0C0FC           FCB 0,$FF,$C0,$C0,$FC
0175 1FA4 C0C0C000             FCB $C0,$C0,$C0,0
0176 1FA8 00FFC0C0CF           FCB 0,$FF,$C0,$C0,$CF
0177 1FAD C3C3FF00             FCB $C3,$C3,$FF,0
0178 1FB1 00C3C3C3FF           FCB 0,$C3,$C3,$C3,$FF
0179 1FB6 C3C3C300             FCB $C3,$C3,$C3,0
0180 1FBA 00FC303030           FCB 0,$FC,$30,$30,$30
0181 1FBF 3030FC00             FCB $30,$30,$FC,0
0182 1FC3 0003030303           FCB 0,3,3,3,3
0183 1FC8 03C3FF00             FCB 3,$C3,$FF,0
0184 1FCC 00C3CCCCF0           FCB 0,$C3,$CC,$CC,$F0
0185 1FD1 CCCCC300             FCB $CC,$CC,$C3,0
0186 1FD5 00C0C0C0C0           FCB 0,$C0,$C0,$C0,$C0
0187 1FDA C0C0FF00             FCB $C0,$C0,$FF,0
0188 1FDE 00C3FFFFC3           FCB 0,$C3,$FF,$FF,$C3
0189 1FE3 C3C3C300             FCB $C3,$C3,$C3,0
0190 1FE7 00C3F3F3CF           FCB 0,$C3,$F3,$F3,$CF
0191 1FEC CFC3C300             FCB $CF,$C3,$C3,0

0192 1FF0 003CC3C3C3           FCB 0,$3C,$C3,$C3,$C3
0193 1FF5 C3C33C00             FCB $C3,$C3,$3C,0
0194 1FF9 00FCC3C3FC           FCB 0,$FC,$C3,$C3,$FC
0195 1FFE C0C0C000             FCB $C0,$C0,$C0,0
0196 2002 003CC3C3C3           FCB 0,$3C,$C3,$C3,$C3
0197 2007 CFCC3F00             FCB $CF,$CC,$3F,0
0198 200B 00FCC3C3FC           FCB 0,$FC,$C3,$C3,$FC
0199 2010 CCC3C300             FCB $CC,$C3,$C3,0
0200 2014 003FC3C03C           FCB 0,$3F,$C3,$C0,$3C
0201 2019 033CFC00             FCB 3,$3C,$FC,0
0202 201D 00FC303030           FCB 0,$FC,$30,$30,$30
0203 2022 30303000             FCB $30,$30,$30,0
0204 2026 00C3C3C3C3           FCB 0,$C3,$C3,$C3,$C3
```

```
0205 202B C3C3FF00              FCB $C3,$C3,$FF,0
0206 202F 00C3C3C3C3            FCB 0,$C3,$C3,$C3,$C3
0207 2034 C33C3C00              FCB $C3,$3C,$3C,0
0208 2038 00C3C3C3C3            FCB 0,$C3,$C3,$C3,$C3
0209 203D FFFFC300              FCB $FF,$FF,$C3,0
0210 2041 00C3C33C3C            FCB 0,$C3,$C3,$3C,$3C
0211 2046 C3C3C300              FCB $C3,$C3,$C3,0
0212 204A 00CCCCCC30            FCB 0,$CC,$CC,$CC,$30
0213 204F 30303000              FCB $30,$30,$30,0
0214 2053 00FF030C3C            FCB 0,$FF,3,$C,$3C
0215 2058 30C0FF00              FCB $30,$C0,$FF,0
0216 205C 003CC3033C            FCB 0,$3C,$C3,3,$3C
0217 2061 30003000              FCB $30,0,$30,0
0218 2065 0000000000            FCB 0,0,0,0,0
0219 206A 003C0C30              FCB 0,$3C,$C,$30
0220 206E 0000000000            FCB 0,0,0,0,0
0221 2073 00000300              FCB 0,0,3,0
0222 2077 A287CFF3      TABLE2  FCB $A2,$87,$CF,$F3
0223 207B 1B876CF3              FCB $1B,$87,$6C,$F3
0224 207F 1B48A2ABF3            FCB $1B,$48,$A2,$AB,$F3
0225 2084                       END

CALC    1E8D  CHRS    1F63  COL     1F62  COMMA   1F13
CONVT   1F44  CONVU   1F51  COUNT   1F67  DISPLA  1F21
DOT     1F18  ERASE   1E82  GO      1ED9  HOLD    1F66
KEY     1EAE  LINE    1F68  LOOP    1F2E  LOOP1   1E13
LOOP3   1EE3  LOOP6   1EF8  OFFSET  1F64  ON      1F60
POLCAT  A1B1  QUEST   1F0E  ROW     1F61  START   1E00
TABLE1  1F69  TABLE2  2077  VIDBEG  0600  ZERO    1F1D
```

Figure 9-3. Program 24—Keyboard Graphic Text

Using Keyboard Graphics

The program is entered at memory location $1E00. After the program is assembled, it is initiated by jumping to that location from your monitor. Using the SDS80C assembler, the entry is made by the command: J1E00.

```
    .
    .
    .
ABUG: J1E00
```

The screen then displays its first prompt.

```
ROW?
```

The input must be a two-digit number from 01 through 15 in hex notation. The first digit that you input must be a 0 or a 1. The second digit must be 0 through F. This combination will form a two-digit hex number from this set in Table 9-3.

Table 9-3. Row Inputs

Input	*Row*	*Input*	*Row*
01	1	0B	B
02	2	0C	C
03	3	0D	D
04	4	0E	E
05	5	0F	F
06	6	10	10
07	7	11	11
08	8	12	12
09	9	13	13
0A	A	14	14
		15	15

After you have entered the row value, the screen will display:

```
COL?
```

The column input must be a two-digit number from 01 through 10 in hex notation. The first digit must be a 1 or a 0. If the first digit is zero, the second must be 1 through F. Otherwise, it must be a zero. These two inputs will form a two-digit hex number from the set in Table 9-4.

After the column is input, the screen will display

```
CHRS?
```

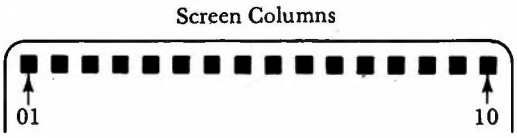
The computer is now requesting the number of inputs that you desire. Once again, a two-digit hex number must be input. A full line starting in the first column could contain 16 (10 hex) characters.

CAUTION

> The program was designed to display one line of text. To keep the program simple, few traps have been set for incorrect inputs. All inputs require two hex values.

Table 9-4.
Column Inputs

Input	*Column*
01	1
02	2
03	3
04	4
05	5
06	6
07	7
08	8
09	9
0A	A
0B	B
0C	C
0D	D
0E	E
0F	F
10	10



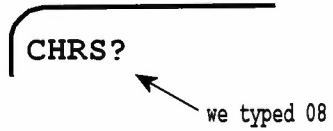
After entering the number of characters, the screen goes blank while it waits for your characters to be typed from the keyboard. The characters are displayed as you type. When your message is complete, type the # key to go back to the monitor. A typical run follows.

1. ABUG: 1E00

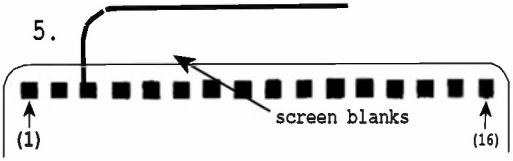
2. ROW?
we typed 0A

3. COL?
we typed 05

4.

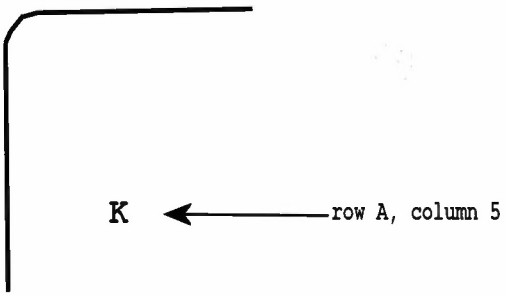


5.

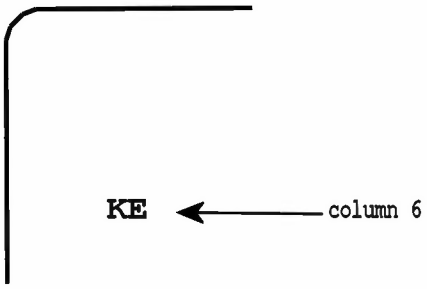


6. We typed the following message

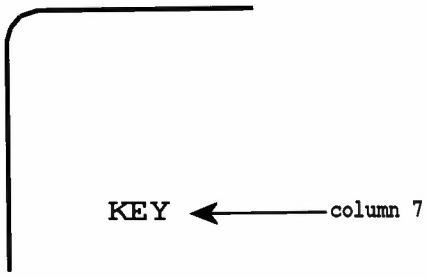
a. K



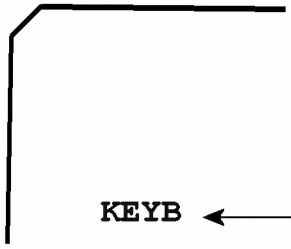
b. E



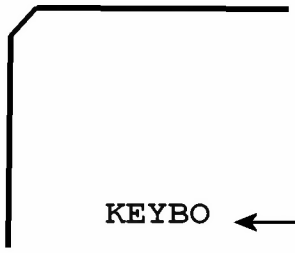
c. Y



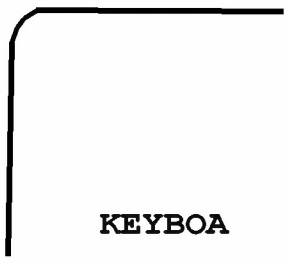
d. B


KEYB
column 8

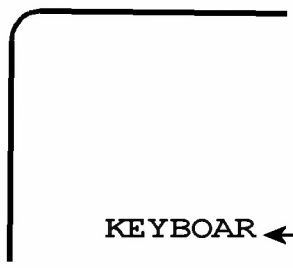
e. 0


KEYBO
column 9

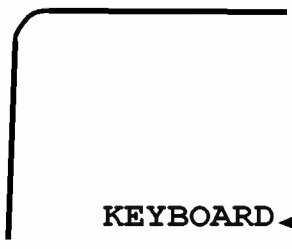
f. A


KEYBOA
column A

g. R


KEYBOAR
column B

h. D


KEYBOARD
column C

7. Your display will stay on the screen until you type the # key. At that time it will return to the monitor.

In Chapter 10, we'll show how to use a modified version of this program as a subroutine to a BASIC program accessed by the BASIC USR function.

Summary

Two methods were used in this chapter to display text characters created by graphics.

The first method demonstrated how to select predetermined characters from a table and display them on the screen at a predetermined position. This method would be satisfactory for labelling areas of a graphic display when the desired position of the labels is known before the program is run.

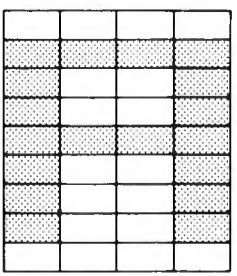
The second method was more flexible because you selected the characters and their position on the screen directly from the program. This method could be used as a subroutine called from any program using graphics. You could create graphic displays and could experiment by adding the text dynamically during your program.

Although both methods created one-color characters in mode 6C, the method could be readily modified to use a different color, several colors, or even a different graphic mode. You could also design additional characters such as lower case letters, numbers, and punctuation marks. You might want to change the size or the shape of the characters used. A simple modification of the data tables would produce the desired changes.

Each character was created from nine data bytes forming characters that are nine elements high by four elements wide.

Example

the letter A



Because the ASCII codes for alphabetic characters follow a regular pattern (A = 41, B = 42, ..., Z = 5A), the data was placed in

the table in corresponding order. Each character requires nine data bytes. Therefore, the beginning of the data for each character is readily found in the table by subtracting 40 from the character's ASCII code and multiplying the result by nine.

$$\text{OFFSET} = (\text{ASCII CODE} - 40) * 9$$

The memory location where the text is to begin on the screen can also be calculated for a selected row and column by

$$\text{LOCATION} = (\text{ROW}-1) * 120 + (\text{COLUMN}-1) * 2 + \text{VIDBEG}$$

- (ROW-1) * 120: allows for 9 rows per character 32 locations per row
- (COLUMN-1) * 2: allows for space between letters
- VIDBEG: beginning of graphic memory

New Instructions in This Chapter

Instruction	*Program Where Used and Description*	
ADDD #$120	24	add $120 to the value in D
BGT KEY	24	used after a compare—if value is greater than the one it is compared to, a branch is made
LBSR DISPLAY	24	used when subroutine is more than 255 bytes away (Long Branch SubRoutine)
LDX #VIDBEG + $124	24	X is loaded from the location (VIDBEG + 124)
LSLB	24	Logical Shift Left register B—shifts all bits in B one place left
SUBA #$30	24	subtracts $30 from value in A
TFR D,U	23	transfers data from U to D—data is also left in U

Chapter Test

1. How many video screen lines do the text characters used in Programs 23 and 24 occupy?

   _____

2. The video memory used in the programs in this chapter starts at $600 and ends at $1DFF.

a. How many memory locations are used for each row of the display? _____

b. According to your answer in part a, how many rows fill the screen? _____

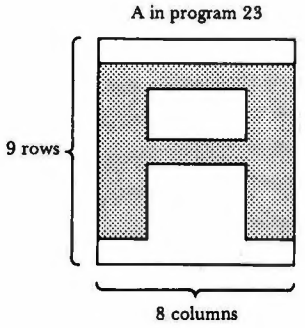
3. Using graphic mode 6C with color set 1, one data byte gives color information for four graphic elements (each two blocks wide ▯▯). Thus one element is coded for color by two of the eight bits of the data byte. The letter A is encoded for orange by the nine bytes shown. Give the nine bytes necessary to change the A to magenta.

A in orange	A in magenta
00	
FF	
C3	
C3	
FF	
C3	
C3	
C3	
00	

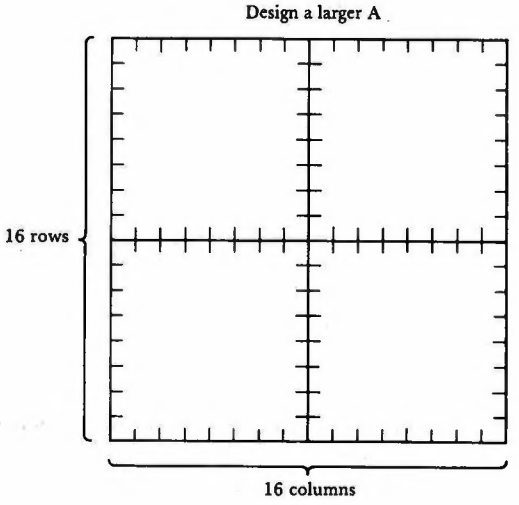
4. In Program 23, the letters O, R, A, N, G, E were accessed by the hex offsets 87, A2, 9, 7E, 3F, 2D in TABLE2. Give the offsets necessary to access the words: MAGENTA.

Letter	Offset
M	
A	
G	
E	
N	
T	
A	

5. An 8 by 9 grid was used to design the characters used in the programs of this chapter.
   a. Design a letter A for a 16 by 16 grid for mode 6C.



Remember, each element
is two columns wide
and 1 row deep



   b. How many data bytes would be needed to encode the letter (in part a) for color in mode 6C? _____

6. Give the data bytes necessary to encode the letter in test exercise 5 for mode 6C in an orange color.

	columns	
row	1-8	9-16
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		

7. Describe how Program 23 might be modified to show the message alternately in the colors
   a. orange on buff b. red on green

_____

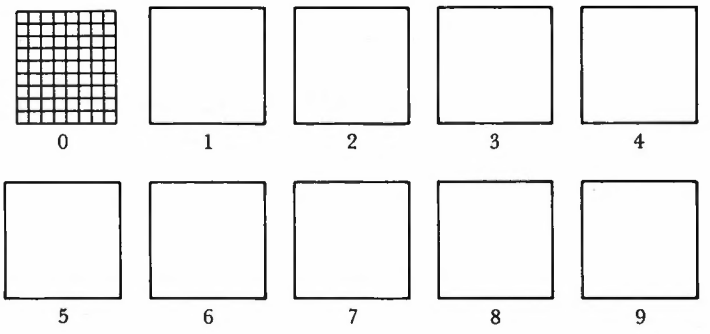
_____

_____

_____

_____

8. Make the modification described in test exercise 7. Don't change the message, just change the colors.
9. Programs 23 and 24 created only alphabetic characters and some punctuation marks. Design numerical digits 0–9 that could be added to the programs.



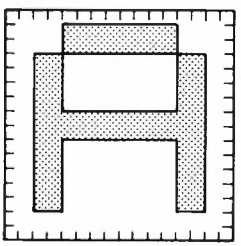
10. Give data that could be added to Programs 23 and 24 (as TABLE3) for drawing your digits of test exercise 9.

Answers to Odd-Numbered Exercises in Chapter Test

1. Nine lines (counting the blank lines at top and bottom)
3. A in Magenta

```
00
AA
82
82
AA
82
82
82
00
```

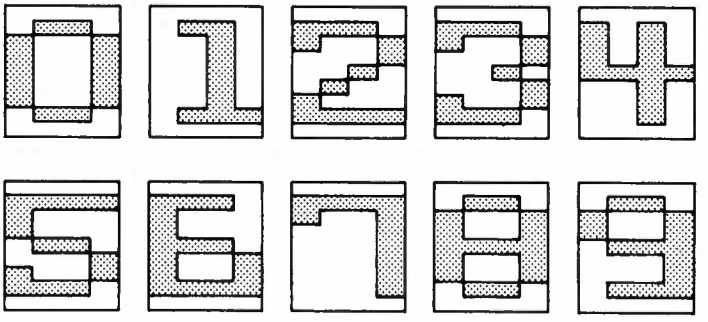
5. a. Here is one way. Yours is probably better.



b. 32 bytes (4 for each row), or
   24 bytes (ignoring blank rows at top and bottom)

7. One way—instead of ending at the POLCAT routine, write a loop that would alternately load E0 and E8 into memory location $FF22. A time delay could be used to keep the screen alternating at a reasonable rate.

9. Here are our designs.



Chapter 10

Vistas Beyond

This chapter contains several useful routines that you may want to use many times. The routines may be used as tools to help you build your own useful programs, or they may serve as models to help you create your own tools. The routines may be saved on tape and appended to other programs as needed.

Another possibility is to "burn" the routines into an EPROM (Erasable Programmable Read Only Memory). The EPROM may be inserted in a printed circuit board and mounted in a ROM cartridge. When you wish to use one of the routines contained in the EPROM, the cartridge can be slipped into the cartridge slot of the computer. The use of an EPROM programmer will be discussed, along with some sample programs.

EPROM Programmer

If you plan to use certain machine language programs often, you may want to consider investing in an EPROM programmer. In this way, you can make permanent copies of often used routines.

We will use a programmer that is designed for the Color Computer to put the text character program, used in Chapters 8 and 9, onto an EPROM so that it can be called from BASIC by a USR function. The programmer that we have used is copyrighted and sold by Spectral Associates.* It will program single-voltage 2716 or 2732 EPROM.

*EPROM Programmer, Spectral Associates, 141 Harvard Ave., Tacoma, WA 98466

The software used to program the EPROMs comes on a cassette tape and is quite easy to use. The programmer's printed circuit board has a Zero Force Insertion (ZIF) socket to hold the EPROM being programmed. You can "burn" the software of the programmer in an EPROM and insert the finished EPROM in a blank socket provided on the board. The EPROM can then be used to eliminate the need for loading the software from tape each time it is to be used.

Linking BASIC to Machine Language

The machine language program that we will put on an EPROM is similar to Program 24, Keyboard Text Graphics. That program was modified due to the assumption that it will be a subroutine called from a BASIC program. The BASIC program is assumed to have selected a high resolution graphics mode and drawn a design of some kind. The subroutine will then allow you to add text to your graphics.

The machine language program also assumes that you have selected a desired row and column where the text is to be started. This is done in the BASIC program and passed as a parameter to the machine language subroutine by the statement

```
X = USRO(20*256 + 20)
         column    row
```

The machine language subroutine finds the values for the column and row by branching to the ROM subroutine INTCNV (INTeger CoNVert). The INTCNV routine will place the values in register D. The most significant byte (hex value for column) will be in register A, and the least significant byte (hex value of the row) will be in register B. These are then stored in memory locations called COL and ROW, as in Program 24.

```
JSR  $B3ED    call INTCNV routine
STA  COL      store A in COL
STB  ROW      store B in ROW
```

The Machine Language Subroutine

The balance of the machine language subroutine, although similar to Program 24, contains additional data for lowercase letters and the digits 0–9. It also allows you to type any number of characters. When you press the ENTER key, a return is made to the BASIC program.

The subroutine may be called from BASIC as many times as you wish. Therefore, you can place several short messages on a drawing. The program is shown in Figure 10-1.

```
0001 0600              VIDBEG EQU $600
0002 0600              FOLCAT EQU $A1B1
0003 0600                     ORG $3C00
0004 3C00 BDB3ED              JSR $B3ED
0005 3C03 B73CC1              STA COL
0006 3C06 F73CC0              STB ROW
0007 3C09 4F                  CLRA
0008 3C0A 1F01                TFR D,X
0009 3C0C 5F                  CLRB
0010 3C0D C30120       CALC   ADDD #$120
0011 3C10 301F                LEAX -1,X
0012 3C12 26F9                BNE CALC
0013 3C14 FD3CC2              STD OFFSET
0014 3C17 4F                  CLRA
0015 3C18 F63CC1              LDB COL
0016 3C1B F33CC2              ADDD OFFSET
0017 3C1E C30600              ADDD #VIDBEG
0018 3C21 1F01                TFR D,X
0019 3C23 BDA1B1       KEY    JSR FOLCAT
0020 3C26 27FB                BEQ KEY
0021 3C28 8120                CMPA #$20
0022 3C2A 1027008C            LBEQ ZERO
0023 3C2E 812E                CMPA #$2E
0024 3C30 10270081            LBEQ DOT
0025 3C34 812C                CMPA #$2C
0026 3C36 2778                BEQ COMMA
0027 3C38 813F                CMPA #$3F
0028 3C3A 276F                BEQ QUEST
0029 3C3C 8140                CMPA #$40
0030 3C3E 2531                BLO NUM
0031 3C40 27E1                BEQ KEY
0032 3C42 815A                CMPA #$5A
0033 3C44 224A                BHI LOWC
0034 3C46 8040                SUBA #$40
0035 3C48 B73CBE              STA HOLD
0036 3C4B 48                  LSLA
0037 3C4C 48                  LSLA
0038 3C4D 48                  LSLA
0039 3C4E BB3CBE              ADDA HOLD
0040 3C51 1F89                TFR A,B
0041 3C53 4F                  CLRA
0042 3C54 C33CC4       UP     ADDD #TABLE1
0043 3C57 1F03         GO     TFR D,U
0044 3C59 8609                LDA #9
0045 3C5B B73CBF              STA LINE
0046 3C5E A6C0         LOOP   LDA ,U+
0047 3C60 A784                STA ,X
0048 3C62 308820              LEAX $20,X
```

```
0049 3C65 7A3CBF            DEC LINE
0050 3C68 26F4              BNE LOOP
0051 3C6A 3089FEE1          LEAX -$11F,X
0052 3C6E 20B3              BRA KEY
0053 3C70 39          BACK  RTS
0054 3C71 810D        NUM   CMPA #$0D
0055 3C73 27FB              BEQ BACK
0056 3C75 8130              CMPA #$30
0057 3C77 25AA              BLO KEY
0058 3C79 8139              CMPA #$39
0059 3C7B 22A6              BHI KEY
0060 3C7D 8030              SUBA #$30
0061 3C7F B73CBE            STA HOLD
0062 3C82 48                LSLA
0063 3C83 48                LSLA
0064 3C84 48                LSLA
0065 3C85 BB3CBE            ADDA HOLD
0066 3C88 1F89              TFR A,B
0067 3C8A 4F                CLRA
0068 3C8B C33EBC            ADDD #TABLE3
0069 3C8E 20C7              BRA GO
0070 3C90 8161        LOWC  CMPA #$61
0071 3C92 258F              BLO KEY
0072 3C94 817A              CMPA #$7A
0073 3C96 228B              BHI KEY
0074 3C98 8061              SUBA #$61
0075 3C9A B73CBE            STA HOLD
0076 3C9D 48                LSLA
0077 3C9E 48                LSLA
0078 3C9F 48                LSLA
0079 3CA0 BB3CBE            ADDA HOLD
0080 3CA3 1F89              TFR A,B
0081 3CA5 4F                CLRA
0082 3CA6 C33DD2            ADDD #TABLE2
0083 3CA9 20AC              BRA GO
0084 3CAB C6F3        QUEST LDB #$F3
0085 3CAD 4F                CLRA
0086 3CAE 20A4              BRA UP
0087 3CB0 C6FC        COMMA LDB #$FC
0088 3CB2 4F                CLRA
0089 3CB3 209F              BRA UP
0090 3CB5 CC0105      DOT   LDD #$105
0091 3CB8 209A              BRA UP
0092 3CBA 4F          ZERO  CLRA
0093 3CBB 5F                CLRB
0094 3CBC 2096              BRA UP
0095 3CBE             HOLD  RMB 1
```

```
0096 3CBF                LINE    RMB 1
0097 3CC0                ROW     RMB 1
0098 3CC1                COL     RMB 1
0099 3CC2                OFFSET  RMB 2
0100 3CC4 0000000000     TABLE1  FCB 0,0,0,0,0
0101 3CC9 00000000               FCB 0,0,0,0
0102 3CCD 003E22223E             FCB 0,$3E,$22,$22,$3E
0103 3CD2 22222200               FCB $22,$22,$22,0
0104 3CD6 003C22223C             FCB 0,$3C,$22,$22,$3C
0105 3CDB 22223C00               FCB $22,$22,$3C,0
0106 3CDF 001C222020             FCB 0,$1C,$22,$20,$20
0107 3CE4 20221C00               FCB $20,$22,$1C,0
0108 3CE8 003C222222             FCB 0,$3C,$22,$22,$22
0109 3CED 22223C00               FCB $22,$22,$3C,0
0110 3CF1 003E20203C             FCB 0,$3E,$20,$20,$3C
0111 3CF6 20203E00               FCB $20,$20,$3E,0
0112 3CFA 003E20203C             FCB 0,$3E,$20,$20,$3C
0113 3CFF 20202000               FCB $20,$20,$20,0
0114 3D03 003E20202E             FCB 0,$3E,$20,$20,$2E
0115 3D08 22223E00               FCB $22,$22,$3E,0
0116 3D0C 002222223E             FCB 0,$22,$22,$22,$3E
0117 3D11 22222200               FCB $22,$22,$22,0
0118 3D15 003E080808             FCB 0,$3E,8,8,8
0119 3D1A 08083E00               FCB 8,8,$3E,0
0120 3D1E 0002020202             FCB 0,2,2,2,2
0121 3D23 02223E00               FCB 2,$22,$3E,0
0122 3D27 0022242830             FCB 0,$22,$24,$28,$30
0123 3D2C 28242200               FCB $28,$24,$22,0
0124 3D30 0020202020             FCB 0,$20,$20,$20,$20
0125 3D35 20203E00               FCB $20,$20,$3E,0
0126 3D39 0022362A22             FCB 0,$22,$36,$2A,$22
0127 3D3E 22222200               FCB $22,$22,$22,0
0128 3D42 0022322A26             FCB 0,$22,$32,$2A,$26
0129 3D47 22222200               FCB $22,$22,$22,0
0130 3D4B 001C222222             FCB 0,$1C,$22,$22,$22
0131 3D50 22221C00               FCB $22,$22,$1C,0
0132 3D54 003C22223C             FCB 0,$3C,$22,$22,$3C
0133 3D59 20202000               FCB $20,$20,$20,0
0134 3D5D 001C222222             FCB 0,$1C,$22,$22,$22
0135 3D62 24261E00               FCB $24,$26,$1E,0
0136 3D66 003C22223C             FCB 0,$3C,$22,$22,$3C
0137 3D6B 28242200               FCB $28,$24,$22,0
0138 3D6F 001C22201C             FCB 0,$1C,$22,$20,$1C
0139 3D74 02221C00               FCB 2,$22,$1C,0
0140 3D78 003E080808             FCB 0,$3E,8,8,8
0141 3D7D 08080800               FCB 8,8,8,0
0142 3D81 0022222222             FCB 0,$22,$22,$22,$22
```

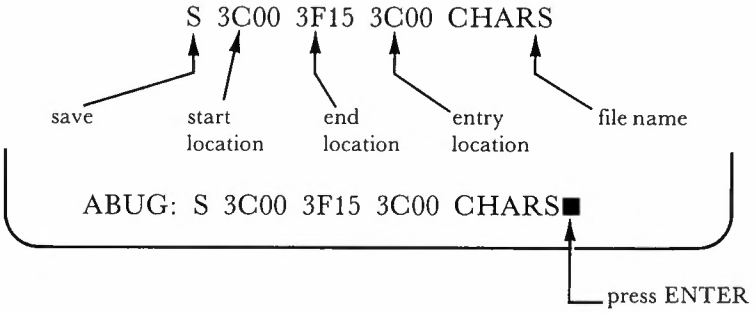
```
0143 3D86 22223E00            FCB $22,$22,$3E,0
0144 3D8A 0022222222          FCB 0,$22,$22,$22,$22
0145 3D8F 22140800            FCB $22,$14,8,0
0146 3D93 0022222222          FCB 0,$22,$22,$22,$22
0147 3D98 2A362200            FCB $2A,$36,$22,0
0148 3D9C 0022221408          FCB 0,$22,$22,$14,8
0149 3DA1 14222200            FCB $14,$22,$22,0
0150 3DA5 0022222214          FCB 0,$22,$22,$22,$14
0151 3DAA 08080800            FCB 8,8,8,0
0152 3DAE 003E020408          FCB 0,$3E,2,4,8
0153 3DB3 10203E00            FCB $10,$20,$3E,0
0154 3DB7 003E02020E          FCB 0,$3E,2,2,$E
0155 3DBC 08000800            FCB 8,0,8,0
0156 3DC0 0000000000          FCB 0,0,0,0,0
0157 3DC5 0C040408            FCB $C,4,4,8
0158 3DC9 0000000000          FCB 0,0,0,0,0
0159 3DCE 0C0C0000            FCB $C,$C,0,0
0160 3DD2 0000001E02  TABLE2  FCB 0,0,0,$1E,2
0161 3DD7 0E120E00            FCB $E,$12,$E,0
0162 3DDB 000020203C          FCB 0,0,$20,$20,$3C
0163 3DE0 22223C00            FCB $22,$22,$3C,0
0164 3DE4 0000001E20          FCB 0,0,0,$1E,$20
0165 3DE9 20201E00            FCB $20,$20,$1E,0
0166 3DED 000002021E          FCB 0,0,2,2,$1E
0167 3DF2 22221E00            FCB $22,$22,$1E,0
0168 3DF6 0000001C22          FCB 0,0,0,$1C,$22
0169 3DFB 3C201E00            FCB $3C,$20,$1E,0
0170 3DFF 00000E081C          FCB 0,0,$E,8,$1C
0171 3E04 08080800            FCB 8,8,8,0
0172 3E08 0000001C22          FCB 0,0,0,$1C,$22
0173 3E0D 221E020C            FCB $22,$1E,2,$C
0174 3E11 000020203C          FCB 0,0,$20,$20,$3C
0175 3E16 22222200            FCB $22,$22,$22,0
0176 3E1A 0000200020          FCB 0,0,$20,0,$20
0177 3E1F 20221C00            FCB $20,$22,$1C,0
0178 3E23 0000020002          FCB 0,0,2,0,2
0179 3E28 0202221C            FCB 2,2,$22,$1C
0180 3E2C 0000202224          FCB 0,0,$20,$22,$24
0181 3E31 38242200            FCB $38,$24,$22,0
0182 3E35 0000202020          FCB 0,0,$20,$20,$20
0183 3E3A 20202000            FCB $20,$20,$20,0
0184 3E3E 000000342A          FCB 0,0,0,$34,$2A
0185 3E43 2A2A2A00            FCB $2A,$2A,$2A,0
0186 3E47 0000003C22          FCB 0,0,0,$3C,$22
0187 3E4C 22222200            FCB $22,$22,$22,0
0188 3E50 0000001C22          FCB 0,0,0,$1C,$22
0189 3E55 22221C00            FCB $22,$22,$1C,0
0190 3E59 0000003C22          FCB 0,0,0,$3C,$22
0191 3E5E 223C2020            FCB $22,$3C,$20,$20
```

```
0192 3E62 0000001C22          FCB 0,0,0,$1C,$22
0193 3E67 221C0406            FCB $22,$1C,4,6
0194 3E6B 0000002E30          FCB 0,0,0,$2E,$30
0195 3E70 20202000            FCB $20,$20,$20,0
0196 3E74 0000001E20          FCB 0,0,0,$1E,$20
0197 3E79 1C023C00            FCB $1C,2,$3C,0
0198 3E7D 000008081C          FCB 0,0,8,8,$1C
0199 3E82 08080600            FCB 8,8,6,0
0200 3E86 0000002222          FCB 0,0,0,$22,$22
0201 3E8B 22221E00            FCB $22,$22,$1E,0
0202 3E8F 0000002222          FCB 0,0,0,$22,$22
0203 3E94 22140800            FCB $22,$14,8,0
0204 3E98 000000222A          FCB 0,0,0,$22,$2A
0205 3E9D 2A2A3600            FCB $2A,$2A,$36,0
0206 3EA1 0000002214          FCB 0,0,0,$22,$14
0207 3EA6 08142200            FCB 8,$14,$22,0
0208 3EAA 0000002222          FCB 0,0,0,$22,$22
0209 3EAF 221E021C            FCB $22,$1E,2,$1C
0210 3EB3 0000003E04          FCB 0,0,0,$3E,4
0211 3EB8 08103E00            FCB 8,$10,$3E,0
0212 3EBC 001C222222  TABLE3  FCB 0,$1C,$22,$22,$22
0213 3EC1 22221C00            FCB $22,$22,$1C,0
0214 3EC5 0038080808          FCB 0,$38,8,8,8
0215 3ECA 08083E00            FCB 8,8,$3E,0
0216 3ECE 001C220204          FCB 0,$1C,$22,2,4
0217 3ED3 08103E00            FCB 8,$10,$3E,0
0218 3ED7 001C22020C          FCB 0,$1C,$22,2,$C
0219 3EDC 02221C00            FCB 2,$22,$1C,0
0220 3EE0 0020283E3E          FCB 0,$20,$28,$28,$3E
0221 3EE5 08080800            FCB 8,8,8,0
0222 3EE9 003E20203C          FCB 0,$3E,$20,$20,$3C
0223 3EEE 02221C00            FCB 2,$22,$1C,0
0224 3EF2 001C22203C          FCB 0,$1C,$22,$20,$3C
0225 3EF7 22221C00            FCB $22,$22,$1C,0
0226 3EFB 001E220202          FCB 0,$1E,$22,2,2
0227 3F00 02020200            FCB 2,2,2,0
0228 3F04 001C22221C          FCB 0,$1C,$22,$22,$1C
0229 3F09 22221C00            FCB $22,$22,$1C,0
0230 3F0D 001C22221E          FCB 0,$1C,$22,$22,$1E
0231 3F12 02020200            FCB 2,2,2,0
0232 3F16                     END

BACK    3C70  CALC    3C0D  COL     3CC1  COMMA   3CB0
DOT     3CB5  GO      3C57  HOLD    3CBE  KEY     3C23
LINE    3CBF  LOOP    3C5E  LOWC    3C90  NUM     3C71
OFFSET  3CC2  POLCAT  A1B1  QUEST   3CAB  ROW     3CC0
TABLE1  3CC4  TABLE2  3DD2  TABLE3  3EBC  UP      3C54
VIDBEG  0600  ZERO    3CBA
```

Figure 10-1. Program 25—Text Subroutine

Preparing the Machine Language Tape

We entered the assembly language program using the SDS80C editor and assembled it. The source and object programs are shown in Figure 10-1. The object code was stored in memory beginning at memory location $3C00. After assembly, the ABUG monitor of the SDS80C system has control of the computer. A fresh cassette tape is placed in the recorder, and the recorder is set to record the program. The object program (the machine language subroutine) is recorded by the ABUG command



It is advisable to record at least two copies of the object program. Then turn off the computer and remove the SDS80C cartridge (or whatever was used to assemble the program).

Before putting the machine language program on an EPROM, it should be tested to make sure it works. We used the BASIC program shown in Figure 10-2 for the test.

```
  A CHAIR-RAISING PROGRAM

 90 DEFUSR0=&H3C00
100 'SET THE SCREEN AND DIMENSION
110 PMODE 4,1
120 PCLS
130 SCREEN 1,0
140 DIM A(8,8),B(20,30),C(12,2)

200 'DRAW FIGURES
210 GOSUB 1000
220 PAINT(112,182),1,1
230 FOR W=1 TO 200: NEXT W
240 LINE(100,20)-(120,25),PSET,BF
250 CIRCLE(110,30),4,,1,.1,.75
260 FOR W=1 TO 1000: NEXT W
```

```
300 'TO SUBROUTINE FOR LABELS
310 X=USR0(20*256+20)   'COL AND ROW
320 X=USR0(20*256+3)
330 X=USR0(8*256+1)

400 'GET HOOK AND LOWER
410 GET(106,26)-(114,34),A,G
420 FOR Y=28 TO 162 STEP2
430   PUT(106,Y)-(114,Y+8),A,PSET
440   PUT(106,Y-2)-(118,Y),C
450   LINE(110,26)-(110,Y+1),PSET
460   FOR W=1 TO 10:NEXT W
470 NEXT Y

500 'RAISE HOOK AND CHAIR
510 GET(106,160)-(126,190),B,G
520 FOR Y=160 TO 30 STEP-2
530   PUT(106,Y-4)-(126,Y+26),B,PSET
540   PUT(106,Y+28)-(126,Y+30),C
550   FOR W=1 TO 25:NEXT W
560 NEXT Y

600 'LOOP
610 GOTO 610

1000 DRAW"BM110,170D20U7E5R7D7U7G5D7U7L7"
1010 DRAW"E5U13D2G5D4E5"
1020 RETURN
```

Figure 10-2. A Chair-Raising Program

The Color Computer is turned on, and the machine language subroutine loaded with the command

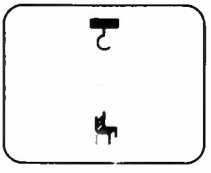
CLOADM "CHARS"

When the tape has been loaded, protect the machine language tape by typing

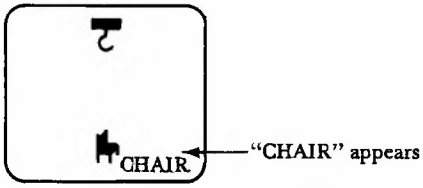
CLEAR &H315,&H3BFF

Then enter the BASIC program.

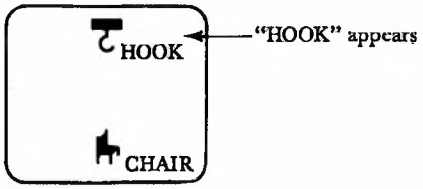
The BASIC program will draw



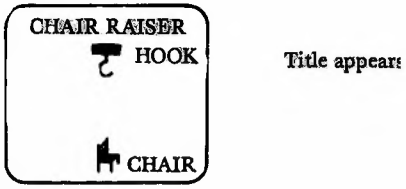
Now type: CHAIR



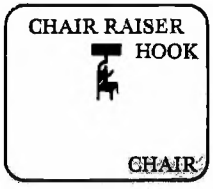
Now type: HOOK



Now type: CHAIR RAISER



Press the ENTER key and the hook will lower and then raise the chair to the top.



Using the Machine Language Subroutine

The machine language subroutine can be used from any BASIC program that calls it. The subroutine assumes that the column and row positions will be passed to it from the BASIC program with the column in accumulator A and the row in accumulator B. The subroutine will then print any of the characters in TABLE1, TABLE2, or TABLE3 that are typed from the keyboard. Pressing the ENTER key causes a return to the BASIC program.

You can use the lowercase capabilities by accessing the Color Computer's lowercase characters. Type SHIFT 0 to access lowercase. If you use lowercase, be sure to use the shift key when typing RUN.

Our machine language subroutine was stored in memory location $3C00 and tested. Our EPROM version (when programmed) will be placed in a cartridge and used from the cartridge slot (beginning at memory location $C000). Therefore, some addresses in the tested program must be changed before a new tape is made that will be used by the EPROM programmer. The necessary changes can be made from a machine language monitor. The values shown in the object code (Figure 10-1) must be changed as shown.

1. Change the values in the following locations from 3C to C0:

   3C04, 3C07, 3C15, 3C19, 3C1C, 3C49, 3C4F, 3C55, 3C5C, 3C66, 3C80, 3C86, 3C9B, 3CA1

2. Change memory location 3CA7 from 3D to C1.
3. Change memory location 3C8C from 3E to C2.

Preparing the EPROM Tape Version

Load the tested program from tape using the ABUG monitor and make the necessary changes. After all changes are made, save this revised tape for use with the EPROM programmer. Save this tape with a new name. We called ours EPCHR. This tape will still be loaded from address $3C00 as before, but it will execute from $C000 when put on the EPROM.

Using the EPROM Programmer

The EPROM programmer from Spectral Associates consists of a printed circuit board that plugs into the ROM cartridge slot and the programming software on cassette tape. The programmer board is

first inserted in the ROM cartridge slot of the computer with the power off. The computer is then turned on, and the software is loaded from the cassette by typing: CLOADM and pressing the ENTER key.

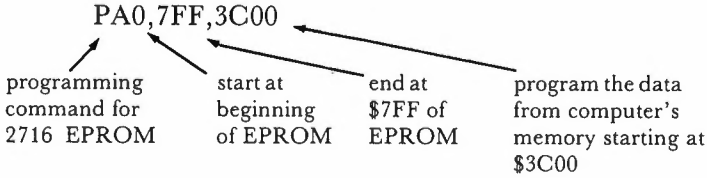
When the program has loaded, you will see

```
F PROMPROG
OK
■
```

Place the EPROM to be programmed in the ZIF socket as described in the programmer manual. Now, run the program by typing: EXEC and pressing the ENTER key. You will see

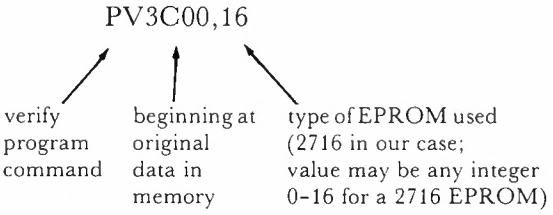
```
OK
EXEC

(C) 1981, SPECTRAL ASSOCIATES

>■
```

The programmer's software begins in RAM at location $800. To copy it to the EPROM, use the command



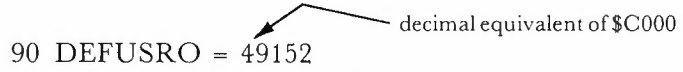
Display

When the programming is finished, be sure to verify that the EPROM now contains the same data as is in the computer's memory. Use the programmer's command



When the EPROM has been verified, turn off the computer. Remove the printed circuit board from the ROM cartridge slot. Remove the EPROM from the ZIF socket and insert it in the empty IC socket on the EPROM programmer board.

When the EPROM programmer is inserted in the cartridge slot in the future, the character producing subroutine can be accessed from memory location $C000. The DEFUSRO statement in line 90 of the BASIC program (Figure 10-2) would be changed to reflect the permanent location of the subroutine to



Other possibilities for the EPROM include commercially available blank ROMPACKs* that have empty sockets for EPROMS. These ROMPACKs can be slipped into the ROM cartridge slot.

Designing Graphic Figures

Creating graphics for use within a program involves much detailed planning. Any aids that you can find to ease and to speed up this process will be quite useful. We have written a program that should be helpful in creating figures to be used in games and graphic displays.

The program displays a 12 × 8 grid at the center of the screen. The grid has a yellow and black border. An inverted asterisk appears in the upper left corner of the grid.

As shown in Figure 10-3, the program title appears at the top of the screen. The color set and color being used to create the figure are shown on the right of the grid. You may change these with commands. The command prompt appears at the bottom left of the screen.

*One is available from the MICRO WORKS, PO Box 1110, Del Mar, CA 92014.

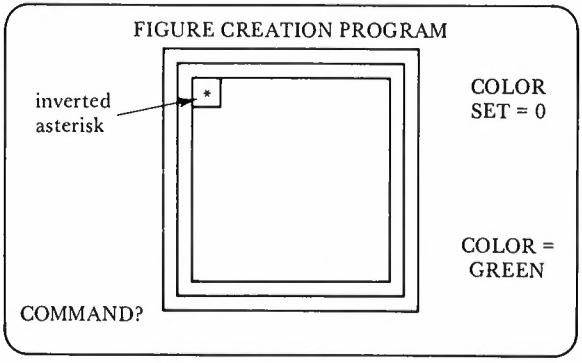


Figure 10–3. Screen for Figure Creator

You use this grid to design a figure that will be created automatically for graphic mode 6C (high resolution, four-color mode). The design is created by moving the inverted asterisk (called the cursor) within the grid by means of the arrow keys. Points are erased or set by using the appropriate color. You select the drawing color, which will remain the same until you give another color command. You can also change color sets, which changes the color of the whole figure. You cannot mix colors from the two color sets.

The colors used are as follows:

Color Set 0	*Color Set 1*
1 = green	5 = buff
2 = yellow	6 = cyan
3 = blue	7 = magenta
4 = red	8 = orange

Program Commands

The following commands are used to create the figures:

↑	move cursor up
↓	move cursor down
←	move cursor left
→	move cursor right
Cx	change draw color (x is color number)

Sy	change color set (y is either 0 or 1)
< space >	set a block where cursor is located
–	erase the block where cursor is located
D	display data table (can be used in future)
T	test (draw the figure, as is, in mode 6C)
R	restart with a clear grid

As you are creating the figure, the grid is displayed in the text mode using block graphics for the figure. While you are in the process of creating the figure, you can switch to high resolution mode 6C to see what your figure presently looks like by using the command "T." You can return to the creation mode by pressing any key. You may then add to the figure.

When your figure is finished to your satisfaction, you may display a table of data values that show the data necessary to create the graphics and the memory locations where each data byte would be stored to display the figure. The values in the table are displayed in hexadecimal form (as shown in Figure 10-4) for use in a machine language program.

```
          SCREEN DATA
ADDRESS     DATA BYTES (HEX)
XXXX + 00   00 00 00
XXXX + 20   00 00 00
XXXX + 40   00 00 00
XXXX + 60   00 00 00
XXXX + 80   00 00 00
XXXX + A0   00 00 00
XXXX + C0   00 00 00
XXXX + E0   00 00 00

HIT ENTER TO RETURN TO GRID
```

Figure 10-4. Data Table for Figure

When the figure is used in another program, any value within the screen area may be chosen for XXXX in (Figure 10-4) as the display address. Make sure that you stay in at least three locations from the right edge of the screen because each line of the figure takes three bytes. The data bytes for your drawing will replace the zeros shown in Figure 10-4 with the values produced by your drawing.

We have produced two versions of the program: an Extended Color BASIC version and an assembly language version. The operation of the BASIC program is shorter and easier to follow. However, it is very slow in creating the figure due to the calculations that must be made. The assembler version is much longer, but the action is instantaneous and therefore more satisfactory when used to draw a figure. The assembler is relocatable to any area of memory.

The assembler version is shown in Figure 10-5, and the BASIC version is shown in Figure 10-6.

Figure 10-5. Figure Creator (Assembler)

```
        ORG $3000
        NAM FIGURE
SCREEN  EQU $400            ;Start address of text screen memory
KEY     EQU $A1B1           ;ROM Keyboard routine
START   CLRA
        STA CSET            ;Color set = 0
        INCA
        STA CCOL            ;Current color = 1 (background)
        STA OLDC            ;Old color = 1
        STA XCUR            ;Draw Cursor X=1
        STA YCUR            ;Draw Cursor Y=1
        LDX #DISP           ;Set whole 96-byte display array
LOOP1   STA ,X+             ;to background color (green)
        CMPX #DISP+96
        BNE LOOP1
REDRAW  LBSR CLS            ;Clear screen
        LDX #SCREEN+4       ;Point X to screen address for title
        LDY #MSG1           ;Point Y to start of title message
        LBSR TEXT           ;Display title message on screen
        LDX #SCREEN+105     ;Display top row of graphics box
        LDY #TOPGR          ;using same method
        LBSR TEXT
        LDX #SCREEN+393     ;Display bottom row of box
        LDY #BOTGR
        LBSR TEXT
        LDX #SCREEN+137     ;Draw sides of box
        LDY #SCREEN+150
        LDA #154
        LDB #149
BACK1   STA ,X
        STB ,Y
        CMPX #SCREEN+361
        BEQ SHOW
        LEAX 32,X
        LEAY 32,Y
        BRA BACK1
SHOW    LBSR DRAW           ;Fill in inside of box
GETCOM  LDX #SCREEN+449     ;Display Command message
        LDY #MSG2
```

```
        LBSR TEXT
        JSR KEY          ;Wait for a key to be pressed
        CMPA #'↑         ;Was it the up-arrow?
        BNE NEXT2
        LDA YCUR         ;Yes, see if user tried to move
        DECA             ;cursor outside of box
        BEQ GETCOM       ;Yes, ignore command
        LBSR SETA        ;Set color of cursor position in
        DEC YCUR         ;array to OLDC, move cursor up
        LBSR READA       ;Set OLDC = color at new cursor posn.
        BRA SHOW         ;Re-display the inside of box
NEXT2   CMPA #10         ;Key = down-arrow?
        BNE NEXT3
        LDA YCUR         ;Yes, see if user tried to move
        INCA             ;cursor outside of box
        CMPA #9
        BEQ GETCOM       ;Yes, ignore command
        LBSR SETA        ;Update array
        INC YCUR         ;Move cursor down
        LBSR READA       ;Update old color (OLDC)
        BRA SHOW         ;Re-display
NEXT3   CMPA #8          ;Key = left-arrow?
        BNE NEXT4
        LDA XCUR         ;See if user tried to move
        DECA             ;cursor outside of box
        BEQ GETCOM       ;Yes, ignore command
        LBSR SETA        ;No, update array
        DEC XCUR         ;Move cursor left
        LBSR READA       ;Update OLDC
        BRA SHOW         ;Re-display
NEXT4   CMPA #9          ;Key = right-arrow?
        BNE NEXT5
        LDA XCUR         ;See if user tried to move
        INCA             ;cursor outside of box
        CMPA #13
        BEQ GETCOM       ;Yes, ignore command
        LBSR SETA        ;No, update array
        INC XCUR         ;Move cursor right
        LBSR READA       ;Update OLDC
        BRA SHOW         ;Re-display
NEXT5   STA SCREEN+458   ;Command is a letter, display it
        CMPA #$20        ;Key = space bar?
        BEQ SET          ;Yes, set point at cursor position
        CMPA #'-         ;Key = dash (-)?
        LBEQ RESET       ;Yes, erase point at cursor
        CMPA #'R         ;Key = R ?
        LBEQ START       ;Yes, restart with clear array
        CMPA #'S         ;Key = S ?
        BEQ CHSET        ;Yes, change color set
        CMPA #'C         ;Key = C ?
        BEQ CHCOL        ;Yes, change drawing color
        CMPA #'T         ;Key = T ?
        BEQ TEST         ;Yes, display figure in Hi-res
        CMPA #'D         ;Key = D ?
        LBEQ TABLE       ;Yes, display data byte table
        LBRA GETCOM      ;Otherwise, ignore command
CHSET   LDA OLDC         ;*** Change Color Set ***
        LBSR SWITCH      ;Change old color to new set
```

```
        STA OLDC
        LDA CCOL            ;Change current color
        LBSR SWITCH
        STA CCOL
        LDX #DISP           ;Point X to start of color array
NEXT6   LDA ,X              ;Get present color
        LBSR SWITCH         ;Change from old color set to new
        STA ,X              ;Put it back in same array position
        LEAX 1,X            ;Point to next array position
        CMPX #DISP+96       ;Done with whole array?
        BNE NEXT6
        LDA #1              ;Yes, change CSET variable
        SUBA CSET           ;CSET = 1 - CSET
        STA CSET            ;Save new CSET
        LBRA SHOW           ;Display new array
CHCOL   LDX #SCREEN+449     ;*** Change Current Color ***
        LDY #MSG3           ;Display message asking for new
        LBSR TEXT           ;color code
        JSR KEY             ;Get key
        STA SCREEN+470      ;Display key on screen
        CMPA #'1            ;If key is less than '1' or key
        BLO CHCOL           ;is greater than '8' then it is
        CMPA #'8            ;not a valid color. Ask for new
        BHI CHCOL           ;color code
        ANDA #%00001111     ;AND off ASCII bits leaving 1-8
        CLRB
        CMPB CSET           ;Is Color Set = 0?
        BNE NXTST
        CMPA #4             ;If color set = 0 and user's color
        BHI CHCOL           ;input > 4 then it is an invalid input
OK      STA CCOL            ;If color is ok, save it as new current
        LBRA SHOW           ;color, redisplay box
NXTST   CMPA #5             ;Color set = 1. Is input < 5?
        BLO CHCOL           ;Yes, it is an invalid input
        BRA OK              ;If input is 5-8 then it is ok
SET     LDA CCOL            ;*** Set Point at cursor position ***
        STA OLDC            ;Change old color to current color
        LBRA GETCOM         ;Don't display until cursor is moved
RESET   LDA #4              ;*** Erase Point at cursor ***
        LDB CSET            ;Change OLDC to background color of
        MUL                 ;current color set by doing
        INCB                ;OLDC = 4 * CSET + 1
        STB OLDC
        LBRA GETCOM         ;Get new command
TEST    LDA CSET            ;*** Display the figure in mode 6C ***
        LDB #8              ;Set up $FF22 byte according to CSET
        MUL
        ADDB #$E0
        STB $FFC5           ;Set up VDG for mode 6C
        STB $FFC3
        STB $FFC7           ;Move display offset to $600 so we
        STB $FF22           ;won't erase text screen, enter 6C
        LDX #$600           ;Point X to start of graphics memory
        CLRB                ;B=0. A=0 from result of MUL above
NEXT7   STD ,X++            ;Clear screen by storing zeroes
        CMPX #$1E00         ;Done with whole screen?
```

```
        BNE NEXT7           ;No, continue
        LBSR MAKEB          ;Yes, create output data list
        LDX #DATA           ;Point X to start of data list above
        LDY #$0C0D          ;Point Y to screen address to start at
BACK2   LDA ,X+             ;Put three bytes from table onto screen
        STA ,Y+             ;horizontally
        LDA ,X+
        STA ,Y+
        LDA ,X+
        STA ,Y+
        CMPY #$0CF0         ;Done with whole figure?
        BEQ NEXT8           ;Yes, go onward
        LEAY 29,Y           ;No, point Y to next screen row
        BRA BACK2           ;Draw next three data bytes
NEXT8   JSR KEY             ;Wait for a key to be pressed
        CLRA                ;Reset display offset to start of
        STA $FFC6             ;text memory ($400)
        STA $FFC0             ;Change VDG mode back to text mode
        STA $FFC2
        STA $FFC4
        STA $FF22             ;Change $FF22 back to text mode
        LBRA SHOW             ;Re-display screen
TABLE   LBSR MAKEB          ;*** Display table of data bytes ***
        BSR CLS             ;Create output data list, clear screen
        LDX #SCREEN+6       ;Display second title message
        LDY #MSG4
        BSR TEXT
        LDX #SCREEN+32      ;Display header message
        LDY #MSG5
        BSR TEXT
        CLRB                ;B = Line counter (0-7)
        LDX #SCREEN+96      ;Point X to display position
        LDY #MSG6           ;Point Y to message data
        LDU #DATA           ;Point U to start of output data
BACK3   BSR TEXT            ;Display message 'XXXX+'
        PSHS B              ;Save B (counter) on stack
        LDA #32             ;Compute offset byte to be
        MUL                 ;displayed = Counter * 32
        TFR B,A
        BSR ASCII           ;Display result, and a space
        LDA #$60            ;Display three more spaces
        STA ,X+
        STA ,X+
        STA ,X+
        LDA ,U+             ;Display byte from output data table
        BSR ASCII           ;pointed to by U
        LDA ,U+             ;Display next table byte
        BSR ASCII
        LDA ,U+             ;Display next table byte
        BSR ASCII
        PULS B              ;Restore counter
        INCB                ;Increment counter
        CMPB #8             ;Are we done with all eight rows?
        BEQ DONE4           ;Yes, go onward
        LEAX 12,X           ;No, point X to next display row
        LDY #MSG6           ;Point Y to start of 'XXXX+' again
        BRA BACK3           ;Do next row
```

```
DONE4   LDX #SCREEN+416     ;Display 'Hit Enter' message
        LDY #MSG7
        BSR TEXT
BACK4   JSR KEY             ;Get key
        CMPA #$0D           ;Is it the ENTER key?
        BNE BACK4           ;No, wait for ENTER key
        LBRA REDRAW         ;Yes, redraw original box display
CLS     LDX #SCREEN         ;This routine loads space ($60)
        LDA #$60            ;codes into the text memory,
WIPE    STA ,X+             ;clearing the screen
        CMPX #SCREEN+$200
        BNE WIPE
        RTS
TEXT    LDA ,Y+             ;This routine displays text on the
        CMPA #0             ;screen. On entry, X points to the
        BEQ DONE1           ;screen address to display data at,
        CMPA #$41           ;Y points to the start address of the
        BLO FIX             ;message data, and the data ends with
        CMPA #$7F           ;a zero byte. The routine also adds
        BHI AOK             ;$40 to any codes that need it so that
        CMPA #$5A           ;they are not displayed as inverted on
        BHI FIX             ;the screen
AOK     STA ,X+
        BRA TEXT
FIX     ADDA #$40
        BRA AOK
DONE1   RTS
SETA    BSR FINDG           ;Point X to Disp array address of
        LDA OLDC            ;cursor position
        STA ,X              ;Set array value = OLDC
        RTS
READA   BSR FINDG           ;Point X to Disp array address of
        LDA ,X              ;cursor position, get array value,
        STA OLDC            ;and store it in OLDC
        RTS
SWITCH  CMPA #5             ;This routine switches the color code
        BLO THENA           ;in A-reg. from one color set to the
        SUBA #4             ;other. If color > 4 then color =
        RTS                 ;color - 4. If color < 5 then
THENA   ADDA #4             ;color = color + 4
        RTS
ASCII   TFR A,B             ;Save hex number to be displayed in B
        RORA                ;Rotate left 4 bits so that they are
        RORA                ;now the right 4 bits
        RORA
        RORA
        ANDA #%00001111     ;Keep binary form of first digit
        ANDB #%00001111     ;Keep binary form of second digit
        BSR CREATE          ;Convert first digit to ASCII
        STA ,X+             ;Display it on screen at X
        TFR B,A             ;Convert second digit to ASCII
        BSR CREATE
        STA ,X+             ;Display it on screen
        LDA #$60            ;Display a space
        STA ,X+
        RTS                 ;Return with X point to next loc.
CREATE  CMPA #$A            ;Is data 0-9 ?
        BLO NUMBER          ;Yes, go convert it
```

```
        ADDA #$37           ;No, add $37 to display A-F
        RTS
NUMBER  ADDA #$70           ;Add $70 to display 0-9
        RTS
FINDG   LDA YCUR            ;This routine points X to the
        DECA                ;memory address where the color
        LDB #12             ;data corresponding to the cursor
        MUL                 ;position (XCUR,YCUR) can be found
        ADDD #DISP
        TFR D,X
        LDB XCUR
        DECB
        ABX
        RTS
MAKEB   CLR COUNT           ;This routine takes bytes from the
        LDX #DISP           ;color data array (Disp), four at a
        LDY #DATA           ;time, converts the color data from
MORE    BSR BITEM           ;1-8 to 0-3, temporarily ignoring the
        LDB #6              ;color set, crams these 4 bytes into
AGAIN   ASLA                ;one byte which represents the data
        DECB                ;to be stored to the screen memory
        BNE AGAIN           ;in graphics mode 6C, and puts this
        TFR A,B             ;byte into the 'output data table'
        BSR BITEM           ;called DATA.
        ASLA
        ASLA
        ASLA
        ASLA
        PSHS A
        ADDB ,S+
        BSR BITEM
        ASLA
        ASLA
        PSHS A
        ADDB ,S+
        BSR BITEM
        PSHS A
        ADDB ,S+
        STB ,Y+             ;Store byte in DATA array
        INC COUNT
        LDA #24
        CMPA COUNT          ;Have we done all 24 bytes?
        BNE MORE            ;No, go back and do next set
        RTS
BITEM   LDA ,X+             ;Get color data, increment X
        DECA                ;Color data = 0-3 or 4-7
        ANDA #%00000011     ;Strip off all but last two bits
        RTS                 ;Now A = 0-3 for either color set
DRAW    CLR COUNT           ;*** Display inside of box ***
        LDX #SCREEN+138     ;Clear counter, point X to inside box
        LDY #DISP           ;Point Y to Disp array of color data
ZOOM    LDA ,Y+             ;Get color byte, create code to store
        DECA                ;in screen memory to display a block
        LDB #16             ;of that color. CODE = (COLOR - 1) *
        MUL                 ;16 + 143.
        ADDB #143
        STB ,X+             ;Display block on screen
        INC COUNT           ;Increment horizontal count
```

```
        LDA #12           ;12 blocks across the screen
        CMPA COUNT        ;Are we done with this row?
        BNE ZOOM          ;No, continue
        CMPY #DISP+96     ;Yes, are we done with the whole thing?
        BEQ WOW           ;Yes, go onward
        CLR COUNT         ;No, clear horizontal count
        LEAX 20,X         ;Point X to start of next row
        BRA ZOOM          ;Go do next row
WOW     LDX #SCREEN+120   ;Display current color set, current
        LDY #MSG8         ;draw color messages and values
        LBSR TEXT
        LDX #SCREEN+152
        LDY #MSG9
        LBSR TEXT
        LDA CSET
        ADDA #$70
        STA ,X
        LDX #SCREEN+248
        LDY #MSG8
        LBSR TEXT
        LDA #$7D
        STA ,X
        LDX #SCREEN+280
        LDY #MSG10
        LBSR TEXT
        LDA CCOL
        DECA
        LDB #8
        MUL
        ADDD #COLORS
        TFR D,Y
        LDX #SCREEN+280
        LBSR TEXT
        LDA YCUR
        DECA
        LDB #32
        MUL
        TFR D,X
        LDB XCUR
        DECB
        ABX
        LEAX SCREEN+138,X
        LDA #'*           ;Display the cursor as an inverted *
        STA ,X            ;at the appropriate position
        RTS
CSET    RMB 1                   ;Current color set
CCOL    RMB 1                   ;Current draw color
OLDC    RMB 1                   ;Color at last position
XCUR    RMB 1                   ;X-coordinate of draw cursor
YCUR    RMB 1                   ;Y-coordinate of draw cursor
COUNT   RMB 1                   ;Counter
DISP    RMB 96                  ;Screen Color Data array
DATA    RMB 24                  ;Output data table created from DISP
MSG1    FCC "FIGURE CREATION PRO"  ;Text messages

        FCC "GRAM"
        FCB 0
TOPGR   FCB 158,156,156,156,156
        FCB 156,156,156,156,156
        FCB 156,156,156,157,0
```

```
BOTGR   FCB 155,147,147,147,147
        FCB 147,147,147,147,147
        FCB 147,147,147,151,0

MSG2    FCC "COMMAND?           "
        FCC "    "
        FCB 0
MSG3    FCC "TYPE NEW COLOR CODE"
        FCC ":   "
        FCB 0
MSG4    FCC "*SCREEN DATA*"
        FCB 0
MSG5    FCC "ADDRESS   DATA"
        FCC " BYTES (HEX)"
        FCB 0
MSG6    FCC "XXXX+"
        FCB 0
MSG7    FCC "HIT ENTER TO RETURN"
        FCC " TO GRID."
        FCB 0
MSG8    FCC "COLOR"
        FCB 0
MSG9    FCC "SET="
        FCB 0
MSG10   FCC "       "
        FCB 0
COLORS  FCC "GREEN "
        FCB 0
        FCC "YELLOW"
        FCB 0
        FCC "BLUE  "
        FCB 0
        FCC "RED   "
        FCB 0
        FCC "BUFF  "
        FCB 0
        FCC "CYAN  "
        FCB 0
        FCC "MAGENTA"
        FCB 0
        FCC "ORANGE"
        FCB 0
        END START
```

11 spaces
4 spaces
3 spaces
4 spaces
7 spaces
2 spaces
1 space
3 spaces
4 spaces
3 spaces
1 space

Figure 10-5. Figure Creator (Assembler)

```
10 'PROGRAM TO CREATE GAME FIGURES ON A 12X8 GRID THEN
15 'PRODUCE DATA LIST AND DISPLAY IN MODE 6C
20 DIM G(12,8),M(2,7),C$(8)
22 RESTORE:FORX=0TO8:READC$(X):NEXTX
24 DATA BLACK,GREEN,YELLOW,BLUE,RED,BUFF,CYAN,MAGENTA,ORANGE
25 CS=0:CC=1:OC=1:CX=1:CY=1
26 FORX=1TO12:FORY=1TO8:G(X,Y)=OC:NEXTY,X
30 CLS:PRINT@4,"FIGURE CREATION PROGRAM";
40 FORX=0TO12:PRINT@106+X,CHR$(156);:PRINT@394+X,CHR$(147);
45 NEXTX
50 FORY=0TO7:PRINT@128+9+32*Y,CHR$(154);
55 PRINT@128+22+32*Y,CHR$(149);:NEXTY
60 PRINT@105,CHR$(158);:PRINT@118,CHR$(157);
65 PRINT@393,CHR$(155);:PRINT@406,CHR$(151);
70 'LINES 40-60 DRAW BORDER
85 GOSUB2000 'DISPLAY ARRAY
90 PRINT@449,"COMMAND?";STRING$(15," ");
95 A$=INKEY$:IFA$=""THEN95
100 IFA$<>"^"THEN200
110 IFCY-1<1THEN90
120 G(CX,CY)=OC:CY=CY-1:OC=G(CX,CY):GOTO85
200 IFA$<>CHR$(10)THEN300 'DOWN-ARROW
210 IFCY+1>8THEN90
220 G(CX,CY)=OC:CY=CY+1:OC=G(CX,CY):GOTO85
300 IFA$<>CHR$(8)THEN400 'LEFT-ARROW
310 IFCX-1<1THEN90
320 G(CX,CY)=OC:CX=CX-1:OC=G(CX,CY):GOTO85
400 IFA$<>CHR$(9)THEN500 'RIGHT-ARROW
410 IFCX+1>12THEN90
420 G(CX,CY)=OC:CX=CX+1:OC=G(CX,CY):GOTO85
500 PRINT@458,A$;:IFA$=" "THENOC=CC:GOTO90 'SET POINT (SPC)
505 IFA$="-"THENOC=4*CS+1:GOTO90 'RESET TO BACKGROUND (-)
510 IFA$="R"THEN22
520 IFA$="S"THEN600
530 IFA$="C"THEN700
540 IFA$="T"THEN800
550 IFA$="D"THEN900
560 GOTO90
600 'CHANGE COLOR SET: CHANGE OC AND CS AND G(ARRAY)
610 OC=-CS*8+OC+4:CC=-CS*8+CC+4:FORX=1TO12
620 FORY=1TO8:G(X,Y)=-CS*8+G(X,Y)+4:NEXTY,X:CS=1-CS:GOTO85
700 'CHANGE COLOR OF SET: CHANGE CC/CHECK IF VALID
710 PRINT@449,"TYPE NEW COLOR CODE:   ";
720 C$=INKEY$:IFC$=""GOTO720
730 PRINT@470,C$;:IFC$<"1" OR C$>"8"GOTO710
740 C=VAL(C$):IFCS=0ANDC>4THEN710
750 IFCS=1ANDC<5THEN710
760 CC=C:GOTO85
800 'TEST FIGURE ON REAL DISPLAY
810 PMODE 3,1:SCREEN 1,CS:PCLS
820 GOSUB 1000 'GET DISPLAY ARRAY
830 FORX=0TO2:FORY=0TO7:POKE 3072+14+X+32*Y,M(X,Y):NEXTY,X
840 C$=INKEY$:IFC$=""THEN840
```

```
850 PCLS:SCREEN 0,0:GOTO85
900 'DISPLAY BYTES TO DRAW FIGURE
910 GOSUB 1000 'GET DISPLAY ARRAY
920 CLS:PRINT"POKE ADDRESS    DATA BYTES (HEX)":PRINT
930 FORY=0TO7
940 PRINT"XXXX+";Y*32;TAB(16);
950 FORX=0TO2:H$=HEX$(M(X,Y)):IFLEN(H$)=1THENH$="0"+H$
960 PRINTH$;" ";:NEXTX:PRINT:NEXTY:PRINT
970 INPUT"HIT ENTER TO RETURN TO GRID";H$:GOTO30
1000 'CREATE DISPLAY ARRAY
1010 FORY1=1TO8:FORX1=0TO2:X2=1:IFCS=1THENX2=5
1020 A1=G(X1*4+1,Y1)-X2:A2=G(X1*4+2,Y1)-X2
1025 A3=G(X1*4+3,Y1)-X2:A4=G(X1*4+4,Y1)-X2
1030 M(X1,Y1-1)=A1*64+A2*16+A3*4+A4:NEXTX1,Y1:RETURN
2000 'RE-DISPLAY INSIDE OF GRID & OTHER INFO
2010 FORY1=1TO8:FORX1=1TO12
2015 PRINT@138+32*(Y1-1)+(X1-1),CHR$((G(X1,Y1)-1)*16+143);
2020 NEXTX1,Y1:PRINT@120,"COLOR";:PRINT@152,"SET=";CS;
2030 PRINT@248,"COLOR=";:PRINT@280,"        ";:PRINT@280,C$(CC);
2040 PRINT@138+32*(CY-1)+(CX-1),"*": 'CURSOR POSITION
2050 RETURN
2060 END
```

Figure 10-6. Figure Creator (BASIC)

Keyboard Sounds

We discussed ways of using *sound* in programs earlier in the book. The program that follows lets you input values for tone and duration from the keyboard. The tone value is input as a two-digit hex number. The duration is input as a four-digit hex number. The program will not accept inputs unless they are in the hex ranges (0–9 and A–F). The inputs are prompted by

1. TONE? ← input tone (2-digit hex)
2. DURATION? ← input duration (4-digit hex)

After the note is sounded, a prompt appears for another note. The process repeats over and over. Here is the program in functional blocks.

```
          NAM KEYBOARD SOUNDS
POLCAT    EQU $A1B1
PRINIT    EQU $A30A

START     LDA #$3F  ; SET-UP AND MAIN PROGRAM
          STA $FF23
          LDY #TONE
          BSR NEXT  ; PRINT TONE PROMPT
          BSR GET2  ; GET TONE VALUE
          STA SPOT
          LDY #DUR
          BSR NEXT  : PRINT DURATION PROMPT
          BSR GET2  : GET DURATION VALUE
          CLRB
          TFR D,X
          BSR GET2  ; GET 2ND DURATION DIGIT PAIR
          TFR A,B
          ABX
BACK      LDB SPOT  ; PLAY IT
BACK2     STB $FF20
          NOP
          NOP
          NOP
          INCB
          BNE BACK2
          DEX
          BNE BACK
          BRA START  ; GET NEXT TONE

NEXT      LDA ,Y+   ; PRINT PROMPT
          CMPA #0   ;   POINTED TO BY Y
          BEQ DONE
          JSR PRINIT
          BRA NEXT

DONE      RTS  ; A 0 ENTRY PUTS YOU HERE

GET2      JSR POLCAT  : THIS ROUTINE GETS
          BSR QUALIF  :    2 HEX DIGITS FROM KBD
          CMPA #0  ;       AND PACKS THEM INTO A
          BMI GET2
          JSR PRINIT
          BSR HEXIT  ; GO CONVERT ASCII TO HEX
          ASLA
          ASLA
          ASLA
          ASLA
          STA SPOT2
TRY2      JSR POLCAT
          BSR QUALIF
          CMPA #0
          BMI TRY2
          JSR PRINIT
          BSR HEXIT
          ADDA SPOT2
          RTS
```

```
QUALIF    CMPA #$30   ; THIS ROUTINE TAKES THE ASCII
          BLO INVERT  ;  CODE IN A AND CHECKS TO SEE
          CMPA #$39   ;   IF IT IS CODE 0-9 OR A-F.
          BLS OKAY    ;   IF IT DOES, A IS UNCHANGED.
          CMPA #$41   ;   IF IT DOESN'T, A IS MADE
          BLO INVERT  ;  NEGATIVE.
          CMPA #$46
          BLS OKAY
INVERT    ORA #$80
OKAY      RTS

HEXIT     CMPA #$41   ;  THIS ROUTINE CONVERTS ASCII
          BLO NUMBER  ;   CODE FOR 0-F TO HEX
          SUBA #$40   ;   00-0F.
          ADDA #9
          RTS

 NUMBER   SUBA #$30
          RTS

 TONE     FCB $0D      ; CARRIAGE RETURN/LINE FEED
          FCC "TONE? "
          FCB 0
 DUR      FCB $0D
          FCC D"DURATION? "
          FCB 0

 SPOT     RMB 1
 SPOT2    RMB 1

          END START
```

Use your assembler to enter KEYBOARD SOUNDS. Then compose your own music. You might want to save the tones and durations that you like so that you can put the data in one of the earlier programs. You could then build up a tape album of your own music.

SURPRISE! No chapter test this time.

The 6809 instruction set is very powerful. We have only shown an exploration of the instructions as used on the Color Computer. Don't stop here. The more that you use assembly language, the easier it will become.

We have included a brief summary of the 6809 instruction set in Appendix H. However, we suggest you get the book *6809 Assembly Language Programming* by Lance Leventhal (Osborne/McGraw-Hill, 630 Bancroft Way, Berkeley, CA 94710). It describes the instructions and how they work in more detail than we have room for in this book.

Appendix A

Saving and Loading Programs Using Tape

The following procedures are used with the SDS80C Development System for tape input and output. Tape commands will differ if you are using some other assembler or machine language monitor. Consult your user's manual for methods to save and load cassette tapes.

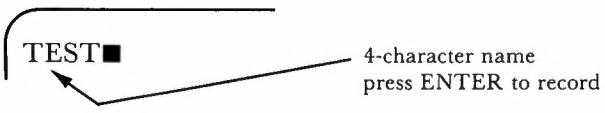
SDS80C System

Assembler Commands

To Write a Text File (Source Program) to Cassette

Be sure the recorder is on and ready to record. Type: W, then enter a string that names the file on tape. Up to eight characters may be used in the string. Longer names are truncated. A null string, obtained by pressing ENTER immediately after the W, will work, but is not recommended.

Example

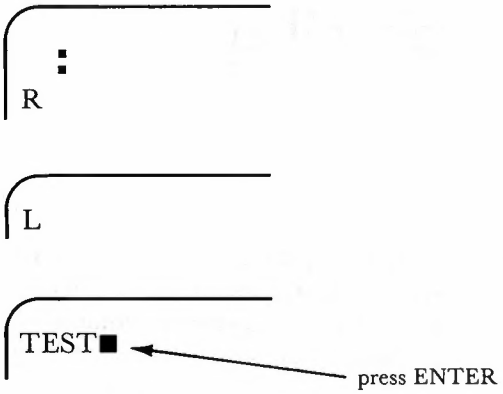


To Read a Source Program into the Text Buffer

Be sure the cassette is positioned correctly and the recorder ready to play.

Type: R, then L, then a string composed of the name under which the tape was recorded.

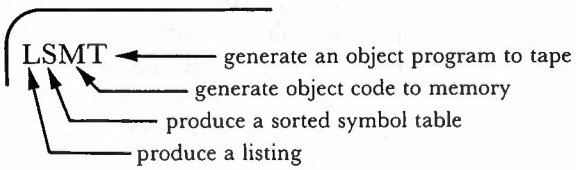
Example



To Generate an Object Program to Cassette

The assembler T option is used.

Example



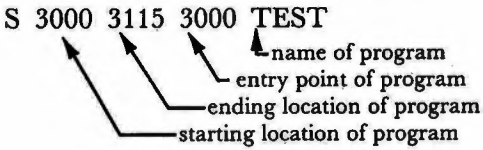
The name for the object tape is taken from the NAM statement, which should appear at the top of the program. If there is no name, the file name will consist of all spaces.

ABUG Commands

Object programs may also be saved and loaded from the ABUG monitor.

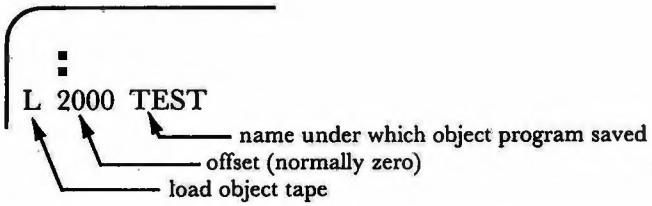
The S command is used from ABUG to save an object program.

Example



The L command is used to load an object program that has been saved by the ABUG S command or the assembler's T option.

Example



The L command is the same as the BASIC command:

CLOADM "TEST",2000

Appendix B

ASCII and Screen Codes

The codes used for the keyboard and video screen for the Color Computer differ. The values shown in this appendix are used.

ASCII CODE (Keyboard)	*SCREEN CODE*	*CHAR.*	*ASCII CODE (Keyboard)*	*SCREEN CODE*	*CHAR.*
20	60	space	35	75	5
21	61	!	36	76	6
22	62	"	37	77	7
23	63	#	38	78	8
24	64	$	39	79	9
25	65	%	3A	7A	:
26	66	&	3B	7B	;
27	67	'	3C	7C	<
28	68	(	3D	7D	=
29	69	)	3E	7E	>
2A	6A	*	3F	7F	?
2B	6B	+	40	40	@
2C	6C	,	41	41	A
2D	6D	—	42	42	B
2E	6E	.	43	43	C
2F	6F	/	44	44	D
30	70	0	45	45	E
31	71	1	46	46	F
32	72	2	47	47	G
33	73	3	48	48	H
34	74	4	49	49	I

ASCII CODE (Keyboard)	*SCREEN CODE*	*CHAR.*	*ASCII CODE (Keyboard)*	*SCREEN CODE*	*CHAR.*
4A	4A	J	65	05	e
4B	4B	K	66	06	f
4C	4C	L	67	07	g
4D	4D	M	68	08	h
4E	4E	N	69	09	i
4F	4F	O	6A	0A	j
50	50	P	6B	0B	k
51	51	Q	6C	0C	l
52	52	R	6D	0D	m
53	53	S	6E	0E	n
54	54	T	6F	0F	o
55	55	U	70	10	p
56	56	V	71	11	q
57	57	W	72	12	r
58	58	X	73	13	s
59	59	Y	74	14	t
5A	5A	Z	75	15	u
5B	5B	[	76	16	v
5C	5C	\	77	17	w
5D	5D	]	78	18	x
5E	5E	Λ	79	19	y
5F	5F	←	7A	1A	z
60	—	'	7B	—	{
61	01	a	7C	—	:
62	02	b	7D	—	}
63	03	c	7E	—	~
64	04	d	7F	—	rub

NOTE: On the screen of the Color Computer, the lowercase characters appear as uppercase but inverted (green on black). Also, the Λ character appears as an up-arrow, and the ← character appears as a back-arrow.

Appendix C

SAM and VDG Settings

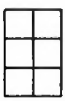
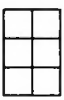
Graphics Mode	*Color Set*	*Register Instructions*	
		To Set	*To Reset to Text*
6R	0	LDA #$F0 STA $FF22 STA $FFC3 STA $FFC5	LDA #0 STA $FF22 STA $FFC2 STA $FFC4
	1	LDA #$F8 STA $FF22 STA $FFC3 STA $FFC5	LDA #0 STA $FF22 STA $FFC2 STA $FFC4
6C	0	LDA #$E0 STA $FF22 STA $FFC3 STA $FFC5	LDA #0 STA $FF22 STA $FFC2 STA $FFC4
	1	LDA #$E8 STA $FF22 STA $FFC3 STA $FFC5	LDA #0 STA $FF22 STA $FFC2 STA $FFC4
3R	0	LDA #$D0 STA $FF22 STA $FFC1 STA $FFC5	LDA #0 STA $FF22 STA $FFC0 STA $FFC4
	1	LDA #$D8 STA $FF22 STA $FFC1 STA $FFC5	LDA #0 STA $FF22 STA $FFC0 STA $FFC4

Graphics Mode	*Color Set*	*Register Instructions*	
		To Set	*To Reset to Text*
3C	0	LDA #$C0 STA $FF22 STA $FFC5	LDA #0 STA $FF22 STA $FFC4
	1	LDA #$C8 STA $FF22 STA $FFC5	LDA #0 STA $FF22 STA $FFC4
2R	0	LDA #$B0 STA $FF22 STA $FFC1 STA $FFC3	LDA #0 STA $FF22 STA $FFC0 STA $FFC2
	1	LDA #$B8 STA $FF22 STA $FFC1 STA $FFC3	LDA #0 STA $FF22 STA $FFC0 STA $FFC2
2C	0	LDA #$A0 STA $FF22 STA $FFC3	LDA #0 STA $FF22 STA $FFC2
	1	LDA #$A8 STA $FF22 STA $FFC3	LDA #0 STA $FF22 STA $FFC2
1R	0	LDA #$90 STA $FF22 STA $FFC1	LDA #0 STA $FF22 STA $FFC0
	1	LDA #$98 STA $FF22 STA $FFC1	LDA #0 STA $FF22 STA $FFC0
1C	0	LDA #$80 STA $FF22 STA $FFC1	LDA #0 STA $FF22 STA $FFC0
	1	LDA #$88 STA $FF22 STA $FFC1	LDA #0 STA $FF22 STA $FFC0

Color Set 0: green/black for modes 6R, 3R, 2R, and 1R
green/yellow/blue/red for modes 6C, 3C, 2C, and 1C
Color Set 1: buff/black for modes 6R, 3R, 2R, and 1R
buff/cyan/magenta/orange for modes 6C, 3C, 2C, and 1C

Appendix D

Graphic Mode Description

Graphic Mode	*Resolution*	*Element Size*	*Code*	*Color Set Set 0*	*Set 1*	*Memory Used (with no offset)*
6R	256 × 192		0	black	black	0400-1BFF
			1	green	buff	
6C	128 × 192		00	green	buff	0400-1BFF
			01	yellow	cyan	
			10	blue	magenta	
			11	red	orange	
3R	128 × 192		0	black	black	0400-0FFF
			1	green	buff	
3C	128 × 96		00	green	buff	0400-0FFF
			01	yellow	cyan	
			10	blue	magenta	
			11	red	orange	
2R	128 × 96		0	black	black	0400-09FF
			1	green	buff	
2C	128 × 64		00	green	buff	0400-0BFF
			01	yellow	cyan	
			10	blue	magenta	
			11	red	orange	
1R	128 × 64		0	black	black	0400-07FF
			1	green	buff	
1C	64 × 64		00	green	buff	0400-07FF
			01	yellow	cyan	
			10	blue	magenta	
			11	red	orange	

Appendix E

Screen Offsets

Desired Starting Address	*To Set Write to Address(es)*	*To Reset to Zero Write to Address(es)*
0000	none	
0200	$FFC7	$FFC6
0400	$FFC9	$FFC8
0600	$FFC7,FFC9	$FFC6,FFC8
0800	$FFCB	$FFCA
0A00	$FFCB,FFC7	$FFCA,FFC6
0C00	$FFCB,FFC9	$FFCA,FFC8
0E00	$FFCB,FFC9,FFC7	$FFCA,FFC8,FFC6
1000	$FFCD	$FFCC
1200	$FFCD,FFC7	$FFCC,FFC6
1400	$FFCD,FFC9	$FFCC,FFC8
1600	$FFCD,FFC9,FFC7	$FFCC,FFC8,FFC6
1800	$FFCD,FFCB	$FFCC,FFCA
1A00	$FFCD,FFCB,FFC7	$FFCC,FFCA,FFC6
1C00	$FFCD,FFCB,FFC9	$FFCC,FFCA,FFC8
1E00	$FFCD,FFCB,FFC9,FFC7	$FFCC,FFCA,FFC8,FFC6
2000	$FFCF	$FFCE

The starting Address is changed by writing to the following registers.

Offset Added	*Write to Register*	*To Clear Offset Write to Register*
$200	$FFC7	$FFC6
$400	$FFC9	$FFC8
$800	$FFCB	$FFCA
$1000	$FFCD	$FFCC
$2000	$FFCF	$FFCE
$4000	$FFD1	$FFD0
$8000	$FFD3	$FFD2

The computer is set to the text mode when it is turned on. The offset is automatically set to offset $400 at that time.

Appendix F

Table to Determine Forward Branches

Steps Forward (Decimal)	*Branch Operand (Hex)*	*Steps Forward (Decimal)*	*Branch Operand (Hex)*	*Steps Forward (Decimal)*	*Branch Operand (Hex)*
1	01	49	31	97	61
2	02	50	32	98	62
3	03	51	33	99	63
4	04	52	34	100	64
5	05	53	35	101	65
6	06	54	36	102	66
7	07	55	37	103	67
8	08	56	38	104	68
9	09	57	39	105	69
10	0A	58	3A	106	6A
11	0B	59	3B	107	6B
12	0C	60	3C	108	6C
13	0D	61	3D	109	6D
14	0E	62	3E	110	6E
15	0F	63	3F	111	6F
16	10	64	40	112	70
17	11	65	41	113	71
18	12	66	42	114	72
19	13	67	43	115	73
20	14	68	44	116	74
21	15	69	45	117	75
22	16	70	46	118	76

Steps Forward (Decimal)	*Branch Operand (Hex)*	*Steps Forward (Decimal)*	*Branch Operand (Hex)*	*Steps Forward (Decimal)*	*Branch Operand (Hex)*
23	17	71	47	119	77
24	18	72	48	120	78
25	19	73	49	121	79
26	1A	74	4A	122	7A
27	1B	75	4B	123	7B
28	1C	76	4C	124	7C
29	1D	77	4D	125	7D
30	1E	78	4E	126	7E
31	1F	79	4F	127	7F
32	20	80	50		
33	21	81	51		
34	22	82	52		
35	23	83	53		
36	24	84	54		
37	25	85	55		
38	26	86	56		
39	27	87	57		
40	28	88	58		
41	29	89	59		
42	2A	90	5A		
43	2B	91	5B		
44	2C	92	5C		
45	2D	93	5D		
46	2E	94	5E		
47	2F	95	5F		
48	30	96	60		

Appendix G

Table to Determine Backward Branches

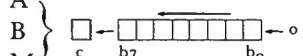
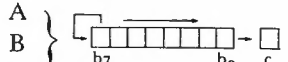
Steps Backward (Decimal)	*Branch Operand (Hex)*	*Steps Backward (Decimal)*	*Branch Operand (Hex)*	*Steps Backward (Decimal)*	*Branch Operand (Hex)*
− 1	FF	− 24	E8	− 47	D1
− 2	FE	− 25	E7	− 48	D0
− 3	FD	− 26	E6	− 49	CF
− 4	FC	− 27	E5	− 50	CE
− 5	FB	− 28	E4	− 51	CD
− 6	FA	− 29	E3	− 52	CC
− 7	F9	− 30	E2	− 53	CB
− 8	F8	− 31	E1	− 54	CA
− 9	F7	− 32	E0	− 55	C9
− 10	F6	− 33	DF	− 56	C8
− 11	F5	− 34	DE	− 57	C7
− 12	F4	− 35	DD	− 58	C6
− 13	F3	− 36	DC	− 59	C5
− 14	F2	− 37	DB	− 60	C4
− 15	F1	− 38	DA	− 61	C3
− 16	F0	− 39	D9	− 62	C2
− 17	EF	− 40	D8	− 63	C1
− 18	EE	− 41	D7	− 64	C0
− 19	ED	− 42	D6	− 65	BF
− 20	EC	− 43	D5	− 66	BE
− 21	EB	− 44	D4	− 67	BD
− 22	EA	− 45	D3	− 68	BC
− 23	E9	− 46	D2	− 69	BB

Steps Backward (Decimal)	*Branch Operand (Hex)*	*Steps Backward (Decimal)*	*Branch Operand (Hex)*	*Steps Backward (Decimal)*	*Branch Operand (Hex)*
−70	BA	−90	A6	−110	92
−71	B9	−91	A5	−111	91
−72	B8	−92	A4	−112	90
−73	B7	−93	A3	−113	8F
−74	B6	−94	A2	−114	8E
−75	B5	−95	A1	−115	8D
−76	B4	−96	A0	−116	8C
−77	B3	− 97	9F	−117	8B
−78	B2	− 98	9E	−118	8A
−79	B1	− 99	9D	−119	89
−80	B0	−100	9C	−120	88
−81	AF	−101	9B	−121	87
−82	AE	−102	9A	−122	86
−83	AD	−103	99	−123	85
−84	AC	−104	98	−124	84
−85	AB	−105	97	−125	83
−86	AA	−106	96	−126	82
−87	A9	−107	95	−127	81
−88	A8	−108	94	−128	80
−89	A7	−109	93		

Appendix H

6809 Instruction Set

6809 Instruction Set (cont'd)

Instr./Forms		*Addressing Modes*	*Description*	*Flags Eff.*
ABX		Inherent	B + X → X	none
ADC	ADCA	Direct, Extended, Immediate, Indexed	A + M + C → A	HNZVC
	ADCB	Direct, Extended, Immediate, Indexed	B + M + C → B	HNZVC
ADD	ADDA	Direct, Extended, Immediate, Indexed	A + M → A	HNZVC
	ADDB	Direct, Extended, Immediate, Indexed	B + M → B	HNZVC
	ADDD	Direct, Extended, Immediate, Indexed	D + M, M + 1 → D	NZVC
AND	ANDA	Direct, Extended, Immediate, Indexed	A∧M → A	NZV
	ANDB	Direct, Extended, Immediate, Indexed	B∧M → B	NZV
	ANDCC	Immediate	CC∧IMM → CC	HNZVC
ASL	ASLA	Inherent	A, B, M 	HNZVC
	ASLB	Inherent		HNZVC
	ASL	Direct, Extended, Indexed		HNZVC
ASR	ASRA	Inherent	A, B, M 	HNZC
	ASRB	Inherent		HNZC
	ASR	Direct, Extended, Indexed		HNZC
BCC	BCC	Relative	Branch C = 0	none
	LBCC	Relative	Long branch C = 0	none
BCS	BCS	Relative	Branch C = 1	none
	LBCS	Relative	Long branch C = 1	none

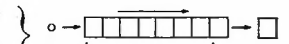
BEQ	BEQ	Relative	Branch Z = 0	none
	LBEQ	Relative	Long branch Z = 0	none
BGE	BGE	Relative	Branch ≥ 0	none
	LBGE	Relative	Long branch ≥ 0	none
BGT	BGT	Relative	Branch > 0	none
	LBGT	Relative	Long branch > 0	none
BHI	BHI	Relative	Branch higher	none
	LBHI	Relative	Long branch higher	none
BHS	BHS	Relative	Branch higher or =	none
	LBHS	Relative	Long branch higher or =	none
BIT	BITA	Direct,Extended,Immediate,Indexed	Bit test A	NZV
	BITB	Direct,Extended,Immediate,Indexed	BIT test B	NZV
BLE	BLE	Relative	Branch ≤ 0	none
	LBLE	Relative	Long branch ≤ 0	none
BLO	BLO	Relative	Branch lower	none
	LBLO	Relative	Long branch lower	none
BLS	BLS	Relative	Branch lower or same	none
	LBLS	Relative	Long branch lower/same	none
BLT	BLT	Relative	Branch < 0	none
	LBLT	Relative	Long branch < 0	none
BMI	BMI	Relative	Branch minus	none
	LBMI	Relative	Long branch minus	none

6809 Instruction Set (cont'd)

Instr./Forms		*Addressing Modes*	*Description*	*Flags Eff.*
BNE	BNE	Relative	Branch not =	none
	LBNE	Relative	Long branch not =	none
BPL	BPL	Relative	Branch plus	none
	LBPL	Relative	Long branch plus	none
BRA	BRA	Relative	Branch always	none
	LBRA	Relative	Long branch always	none
BRN	BRN	Relative	Branch never	none
	LBRN	Relative	Long branch never	none
BSR	BSR	Relative	Branch sub.	none
	LBSR	Relative	Long branch subroutine	none
BVC	BVC	Relative	Branch V = 0	none
	LBVC	Relative	Long branch V = 0	none
BVS	BVS	Relative	Branch V = 1	none
	LBVS	Relative	Long branch V = 1	none
CLR	CLRA	Inherent	0→A	NZVC
	CLRB	Inherent	0→B	NZVC
	CLR	Direct, Extended, Indexed	0→M	NZVC
CMP	CMPA	Direct,Extended,Immediate,Indexed	A-M	HNZVC
	CMPB	Direct,Extended,Immediate,Indexed	B-M	HNZVC
	CMPD	Direct,Extended,Immediate,Indexed	D-M,M + 1	NZVC
	CMPS	Direct,Extended,Immediate,Indexed	S-M,M + 1	NZVC
	CMPU	Direct,Extended,Immediate,Indexed	U-M,M + 1	NZVC
	CMPX	Direct,Extended,Immediate,Indexed	X-M,M + 1	NZVC
	CMPY	Direct,Extended,Immediate,Indexed	Y-M,M + 1	NZVC

COM	COMA	Inherent	A→A	NZVC
	COMB	Inherent	B→B	NZVC
	COM	Direct,Extended,Indexed	M→M	NZVC
CWAI		Inherent,Immediate	CC∧IMM→CC	HNZVC
DAA		Inherent	Dec.Adj.A	NZVC
DEC	DECA	Inherent	A-1→A	NZV
	DECB	Inherent	B-1→B	NZV
	DEC	Direct,Extended,Indexed	M-1→M	NZV
EOR	EORA	Direct,Extended,Immediate,Indexed	A⊻M→A	NZV
	EORB	Direct,Extended,Immediate,Indexed	B⊻M→B	NZV
EXB	R1,R2	Inherent	R1←R2	none
INC	INCA	Inherent	A + 1→A	NZV
	INCB	Inherent	B + 1→B	NZV
	INC	Direct,Extended,Indexed	M + 1→M	NZV
JMP		Direct,Extended,Indexed	Eff.Add→PC	none
JSR		Direct,Extended,Indexed	Jump subr.	none
LD	LDA	Direct,Extended,Immediate,Indexed	M→A	NZV
	LDB	Direct,Extended,Immediate,Indexed	M→B	NZV
	LDD	Direct,Extended,Immediate,Indexed	M,M + 1→D	NZV
	LDS	Direct,Extended,Immediate,Indexed	M,M + 1→S	NZV
	LDU	Direct,Extended,Immediate,Indexed	M,M + 1→U	NZV
	LDX	Direct,Extended,Immediate,Indexed	M,M + 1→X	NZV
	LDY	Direct,Extended,Immediate,Indexed	M,M + 1→Y	NZV

6809 Instruction Set (cont'd)

Instr./Forms		*Addressing Modes*	*Description*	*Flags Eff.*
LEA	LEAS	Indexed	EA→S	none
	LEAU	Indexed	EA→U	none
	LEAX	Indexed	EA→X	Z
	LEAY	Indexed	EA→Y	Z
LSL	LSLA	Inherent	A	NZVC
	LSLB	Inherent	B 	NZVC
	LSL	Direct,Extended,Indexed	M	NZVC
LSR	LSRA	Inherent	A	NZC
	LSRB	Inherent	B 	NZC
	LSR	Direct,Extended,Indexed	M	NZC
MUL		Inherent	A x B→D	ZC
NEG	NEGA	Inherent	$\overline{A}$ + 1→A	HNZVC
	NEGB	Inherent	$\overline{B}$ + 1→B	HNZVC
	NEG	Direct,Extended,Indexed	$\overline{M}$ + 1→M	HNZVC
NOP		Inherent	no operation	none
OR	ORA	Direct,Extended,Immediate,Indexed	AvM→A	NZV
	ORB	Direct,Extended,Immediate,Indexed	BvM→B	NZV
	ORCC	Immediate	CCvIMM→CC	HNZVC
PSH	PSHS	Immediate	Reg. to S	none
	PSHU	Immediate	Reg. to U	none

PUL	PULS	Immediate	Reg. from S	none
	PULU	Immediate	Reg. from U	none
ROL	ROLA	Inherent	A	NZVC
	ROLB	Inherent	B	NZVC
	ROL	Direct,Extended,Indexed	M	NZVC
ROR	RORA	Inherent	A	NZC
	RORB	Inherent	B	NZC
	ROR	Direct,Extended,Indexed	M	NZC
RTI		Inherent	Ret. from Int	HNZVC
RTS		Inherent	Ret. from Sub.	none
SBC	SBCA	Direct,Extended,Immediate,Indexed	A-M-C→A	HNZVC
	SBCB	Direct,Extended,Immediate,Indexed	B-M-C→B	HNZVC
SEX		Inherent	sign ext. B→A	NZV
ST	STA	Direct,Extended,Indexed	A→M	NZV
	STB	Direct,Extended,Indexed	B→M	NZV
	STD	Direct,Extended,Indexed	D→M,M + 1	NZV
	STS	Direct,Extended,Indexed	S→M,M + 1	NZV
	STU	Direct,Extended,Indexed	U→M,M + 1	NZV
	STX	Direct,Extended,Indexed	X→M,M + 1	NZV
	STY	Direct,Extended,Indexed	Y→M,M + 1	NZV

6809 Instruction Set (cont'd)

Instr./Forms		*Addressing Modes*	*Description*	*Flags Eff.*
SUB	SUBA	Direct,Extended,Immediate,Indexed	A-M A	HNZVC
	SUBB	Direct,Extended,Immediate,Indexed	B-M B	HNZVC
	SUBD	Direct,Extended,Immediate,Indexed	D-M,M + 1 D	NZVC
SWI	SWI	Inherent	Softw. Int.1	none
	SWI2	Inherent	Softw. Int.2	none
	SWI3	Inherent	Softw. Int.3	none
SYNC		Inherent	Synch to Int.	none
TFR	R1,R2	Inherent	R1 R2	none
TST	TSTA	Inherent	Test A	NZV
	TSTB	Inherent	Test B	NZV
	TST	Direct,Extended,Indexed	Test M	NZV

Notes: EA is Effective Address
S is Stack S
U is Stack U
M is Memory
A is Accumulator A
B is Accumulator B
D is Accumulator D
X is Register X
Y is Register Y
CC is Condition Code register
IMM is Immediate value
PC is Program Counter
∧ is logical AND
V is logical OR
⊻ is logical EXCLUSIVE OR

Index

ASSEMBLY LANGUAGE GRAPHICS for the TRS-80 COLOR COMPUTER

Don Inman Kurt Inman
with Dymax

Written specifically for the TRS-80 Color Computer, this dynamic new book uses sound and graphics to show you how 6809 assembly language can be used to perform tasks that would be difficult or impossible with BASIC. All of the techniques included in this book are explained in a hands-on approach so that you begin to learn as soon as you pick up the book. Learn how to tailor your own programming style, from editing, assembling, executing, and even debugging, to make your programs run quickly and efficiently. **Assembly Language Graphics** is packed with video screen diagrams, which explain each step of the process of creating your own graphics. By combining the extraordinary capabilities of the TRS-80 Color Computer, your curiosity and imagination, and **Assembly Language Graphics**, a powerful new programming skill will be yours.

The Table of Contents:

Cover Design by Debbie Balboni
RESTON PUBLISHING COMPANY, INC.
A Prentice Hall Company
Reston, Virginia
0-8359-0317-6