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
Contents

Preface vii

Chapter 1 Introduction to Machine Language 1
A Bit About Numbers, 2
The Machine Language Monitor, 5
How Program 1 Works, 13
Instructions Used in Program 1, 17
Signed Numbers, 18
Summary, 22
Chapter Test, 23
Answers to Odd-Numbered Exercises in Chapter Test, 24

Chapter 2 Sound 25
Program 2—Making Your Own Sounds, 25
New Instructions in Program 2, 28
How Program 2 Works, 29
Condition Codes, 34
Using a Sound Table, 36
Explanation of Program 3, 38
New Instructions in Program 3, 39
Suggestions for Playing With Program 3, 40
Saving Machine Language Programs on Tape, 41
Summary, 42
Chapter Test, 43
Answers to Odd-Numbered Exercises in Chapter Test, 44
Chapter 3  Edit, Assemble, and ABUG  45
The Editor, 46
The Assembler, 53
The ABUG Monitor, 55
Assembly Language Used in Program 4, 57
Designing a Graphics Program, 58
Program 5—Color Bar, 61
New Instructions Used in Program 5, 63
Summary, 65
Chapter Test, 66
Answers to Odd-Numbered Exercises in Chapter Test, 67

Chapter 4  Color Graphics  69
Setting SAM, VDG, and Color Bytes, 70
Putting a Graphics Program Together, 74
Program 6—Drawing an Orange Rectangle, 76
How Program 6 Works, 78
Program 7—Drawing an Orange Rectangle in Mode 1C, 82
Screen Memory for Four-Color Graphic Modes, 86
Summary, 92
Chapter Test, 94
Answers to Odd-Numbered Exercises in Chapter Test, 95

Chapter 5  Animation  97
Two-Color Graphics, 97
Using Mode 1R, 100
Parts for Program 9, 102
How Program 9 Works, 104
Program 10—Moving the Piston, 107
Paging Screen Memory, 111
Program 11—Animation by Paging, 113
Description of Program 11, 116
Summary, 118
Chapter Test, 120
Answers to Odd-Numbered Exercises in Chapter Test, 122

Chapter 6  Sound and Graphics  125
Experimenting with Sound, 125
Program 12—Sound Explorer, 125
Using the Sound Explorer, 128
Adding Sound to Programs, 130
Program 13—Sound Bars, 131
Program 14—Rotating Bar, 133
Animation with Sound, 139
General Information on Programs in this
Chapter, 143
Summary, 144
Chapter Test, 145
Answers to Odd-Numbered Exercises in Chapter
Test, 149

Chapter 7
Joystick Animation  151
Designing a Joystick Program, 151
Using the Flying Saucer, 156
New Instructions and Data Forms, 156
Saucer around the Pylons, 157
Interrupts, 157
How Program 17 Works, 162
Using the Timer Function, 162
Timing your Flights, 166
Double Saucers, 168
Summary, 169
Chapter Test, 171
Answers to Odd-Numbered Exercises in Chapter
Test, 174

Chapter 8
Text  175
Using a Text Processor in the Text Mode, 175
Program 19—Word Processor, 176
Adding Backspace to the Word Processor, 182
Enlarging the Text Buffer, 186
Creating Text Characters, 188
Displaying Text in a Graphics Mode, 190
Program 20—Displaying Text, 190
Summary, 193
Chapter Test, 194
Answers to Odd-Numbered Exercises in Chapter
Test, 195

Chapter 9
Graphics with Text  197
Planning the Display, 198
Selecting a Character, 199
Placing the Character on the Screen, 201
CONTENTS

Program 23—Orange Text by Graphics, 202
How Program 23 Works, 206
Selecting Characters from the Keyboard, 206
Placement of Characters on the Screen, 207
Message to Select Inputs, 208
Using Keyboard Graphics, 213
Summary, 218
Chapter Test, 219
Answers to Odd-Numbered Exercises in Chapter Test, 223

Chapter 10 Vistas Beyond 225
EPROM Programmer, 225
Linking BASIC to Machine Language, 226
The Machine Language Subroutine, 226
Preparing the Machine Language Tape, 228
Using the Machine Language Subroutine, 235
Preparing the EPROM Tape Version, 235
Using the EPROM Programmer, 235
Designing Graphic Figures, 237
Program Commands, 238
Keyboard Sounds, 249

Appendix A Saving and Loading Programs Using Tape 253
Appendix B ASCII and Screen Codes 257
Appendix C SAM and VDG Settings 259
Appendix D Graphic Mode Description 261
Appendix E Screen Offsets 263
Appendix F Table to Determine Forward Branches 265
Appendix G Table to Determine Backward Branches 267
Appendix H 6809 Instruction Set 269
Index 277
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.
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 "personalities," 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

\[ \text{01011100} \]

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.

```
0 1 1 1 1 1 0 1
```

← block of 8 bits (1 byte)

```
  1
```

← one bit

```
  1101
```

← four bits (sometimes called a nybble)

```
  0111101
```

← eight bits (a byte)

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>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Binary Places</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^3$ $2^2$ $2^1$ $2^0$</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>$0 + 0 + 0 + 1 = 1$</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>$0 + 0 + 2 + 0 = 2$</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>$0 + 4 + 0 + 0 = 4$</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>$8 + 0 + 0 + 0 = 8$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Binary Places</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^7$ $2^6$ $2^5$ $2^4$</td>
<td>$16^1$</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>2</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>4</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>8</td>
</tr>
</tbody>
</table>
Next, look at the binary place values of the complete 16-bit number.

<table>
<thead>
<tr>
<th>Binary Places</th>
<th>Decimal Equivalent</th>
<th>Hex Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0 0 0 0 1 0 0 0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>0 0 0 1 0 0 0 0</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>0 1 0 0 0 0 0 0</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>64</td>
<td>40</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>128</td>
<td>80</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>binary</th>
<th>split</th>
<th>hex</th>
<th>shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111101</td>
<td>0111</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>11000011</td>
<td>1100</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>10101010</td>
<td>1010</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>10101010</td>
<td>1010</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

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

```
OK
EXEC 49152
```

(C) 1980 BY THE MICRO WORKS

CBUG:

this prompt lets you know you are in the monitor (other monitors will no doubt have a different prompt)

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

```
CBUG: M 0700
0700  FF FF FF FF FF FF FF FF
```

cursor

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

```
0700 86 FF FF FF FF FF FF
```

two spaces here

Type 3F

```
0700 86 3F FF FF FF FF FF
```

Type B7

```
0700 86 3F B7 FF FF FF FF
```

etc.

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

```
0700 86 3F B7 FF 23 1F 89 F7
0708 FF FF FF FF FF FF FF
```
Continue entering data until the complete program is in memory.

```
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
```

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).

```
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
```

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).

```
8-bit register

16-bit register
```

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.

```
CBUG: R
```

Type R to display registers

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

Here is the program counter

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

```
CBUG: 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.

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

Cursor is in front of program counter
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.
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 1

← 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  memory location $FF23

0704 23  just as before
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 $FF20 is the memory location
0709 20

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

```
1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | negative
0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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.
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**

\[
\begin{align*}
10000000 &= -128 \\
10000001 &= -127 \\
10000010 &= -126 \\
11111101 &= -3 \\
11111110 &= -2 \\
11111111 &= -1
\end{align*}
\]

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**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>33</td>
<td>21</td>
<td>65</td>
<td>41</td>
<td>97</td>
<td>61</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>34</td>
<td>22</td>
<td>66</td>
<td>42</td>
<td>98</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>35</td>
<td>23</td>
<td>67</td>
<td>43</td>
<td>99</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>36</td>
<td>24</td>
<td>68</td>
<td>44</td>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>37</td>
<td>25</td>
<td>69</td>
<td>45</td>
<td>101</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>38</td>
<td>26</td>
<td>70</td>
<td>46</td>
<td>102</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>39</td>
<td>27</td>
<td>71</td>
<td>47</td>
<td>103</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>40</td>
<td>28</td>
<td>72</td>
<td>48</td>
<td>104</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>41</td>
<td>29</td>
<td>73</td>
<td>49</td>
<td>105</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>42</td>
<td>2A</td>
<td>74</td>
<td>4A</td>
<td>106</td>
<td>6A</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>43</td>
<td>2B</td>
<td>75</td>
<td>4B</td>
<td>107</td>
<td>6B</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>44</td>
<td>2C</td>
<td>76</td>
<td>4C</td>
<td>108</td>
<td>6C</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>45</td>
<td>2D</td>
<td>77</td>
<td>4D</td>
<td>109</td>
<td>6D</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>46</td>
<td>2E</td>
<td>78</td>
<td>4E</td>
<td>110</td>
<td>6E</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>47</td>
<td>2F</td>
<td>79</td>
<td>4F</td>
<td>111</td>
<td>6F</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>48</td>
<td>30</td>
<td>80</td>
<td>50</td>
<td>112</td>
<td>70</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>49</td>
<td>31</td>
<td>81</td>
<td>51</td>
<td>113</td>
<td>71</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>50</td>
<td>32</td>
<td>82</td>
<td>52</td>
<td>114</td>
<td>72</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>51</td>
<td>33</td>
<td>83</td>
<td>53</td>
<td>115</td>
<td>73</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>52</td>
<td>34</td>
<td>84</td>
<td>54</td>
<td>116</td>
<td>74</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>53</td>
<td>35</td>
<td>85</td>
<td>55</td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>54</td>
<td>36</td>
<td>86</td>
<td>56</td>
<td>118</td>
<td>76</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>55</td>
<td>37</td>
<td>87</td>
<td>57</td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>56</td>
<td>38</td>
<td>88</td>
<td>58</td>
<td>120</td>
<td>78</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>57</td>
<td>39</td>
<td>89</td>
<td>59</td>
<td>121</td>
<td>79</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>58</td>
<td>3A</td>
<td>90</td>
<td>5A</td>
<td>122</td>
<td>7A</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>59</td>
<td>3B</td>
<td>91</td>
<td>5B</td>
<td>123</td>
<td>7B</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>60</td>
<td>3C</td>
<td>92</td>
<td>5C</td>
<td>124</td>
<td>7C</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>61</td>
<td>3D</td>
<td>93</td>
<td>5D</td>
<td>125</td>
<td>7D</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>62</td>
<td>3E</td>
<td>94</td>
<td>5E</td>
<td>126</td>
<td>7E</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>63</td>
<td>3F</td>
<td>95</td>
<td>5F</td>
<td>127</td>
<td>7F</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>64</td>
<td>40</td>
<td>96</td>
<td>60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1-5. Branch Bytes for Backward Branches

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

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0
\end{array}
\]

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

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\hline
9 & C
\end{array}
\]

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.

\[
\begin{array}{cccccccc}
\hline
\hline
\hline
\hline
A & B & \text{double accumulator D}
\end{array}
\]
• 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.

\[
\begin{align*}
01110110 & = \quad \quad \\
10011100 & = \quad \quad \\
11101111 & = \quad \quad \\
\end{align*}
\]

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 Chapter Two

1. switched closed is 1
   switch open is 0
2. 6809 or 6809E
3. D3 = 11010011
   8C = 10001100
   5A = 01011010
4. 8 bits
5. 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

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

Leave the memory mode (by pressing ENTER) and examine the registers to 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
```

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
```

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
```

change this to 80
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>
<thead>
<tr>
<th>Op Code</th>
<th>Mnemonic</th>
<th>Bytes</th>
<th>Address Mode</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6</td>
<td>LDB</td>
<td>2</td>
<td>Immediate</td>
<td>accumulator B is loaded with data that follows</td>
</tr>
<tr>
<td>8E</td>
<td>LDX</td>
<td>3</td>
<td>Immediate</td>
<td>register X is loaded with following data</td>
</tr>
<tr>
<td>30 1F</td>
<td>DEX</td>
<td>2</td>
<td>Inherent</td>
<td>the X register is decremented by 1</td>
</tr>
<tr>
<td>39</td>
<td>RTS</td>
<td>1</td>
<td>Inherent</td>
<td>return is made to CBUG</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>6809 MICROPROCESSOR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Decoder</td>
<td></td>
</tr>
<tr>
<td>Program Counter</td>
<td></td>
</tr>
<tr>
<td>Accumulator A</td>
<td></td>
</tr>
<tr>
<td>Accumulator B</td>
<td></td>
</tr>
<tr>
<td>Register X</td>
<td></td>
</tr>
</tbody>
</table>

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 pro-
gram? 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 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.

```
01111111
XXX0XXX
```

accumulator A positive

CC register

N flag = 0

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

```
10000000
XXX1XXX
```

accumulator A negative

CC register

N flag = 1

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.

```
00000001111111
XXX0XX
```

X register not zero

CC register

zero flag not set
If the result in the X register is zero, the zero flag will be set to one.

\[ \begin{align*}
0000000000000000 & \quad \text{X register} = 0 \\
XXXXX1XX & \quad \text{CC register}
\end{align*} \]

zero flag is set

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:

\[ \begin{align*}
A = 3F & \quad B = 00 \\
D = 00 & \quad X = 0000 \\
Y = AAF1 & \quad U = 0000 \\
P = 0718 & \quad S = 7F19 \\
E----Z-C & \\
\end{align*} \]

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

\[ E----Z-C \]

\[ \begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array} \]

- Carry is set to 1
- Overflow = 0
- Zero is set to 1
- Negative = 0
- Interrupt = 0
- Half carry = 0
- Fast interrupt = 0
- Entire is set to 1

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

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.
Now, make sure the program counter is set 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>
<thead>
<tr>
<th>Op Code</th>
<th>Mnemonic</th>
<th>Bytes</th>
<th>Address</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 8E</td>
<td>LDY</td>
<td>4</td>
<td>Immediate</td>
<td>16-bit register Y is loaded with 2 bytes that follow</td>
</tr>
<tr>
<td>E6 A0</td>
<td>LDB,Y+</td>
<td>2</td>
<td>Indexed</td>
<td>B register is loaded from address held in Y; Y increased by 1</td>
</tr>
<tr>
<td>C1</td>
<td>CMPB</td>
<td>2</td>
<td>Immediate</td>
<td>B register compared to byte following op code</td>
</tr>
<tr>
<td>27</td>
<td>BEQ</td>
<td>2</td>
<td>Relative</td>
<td>Branch made if zero flag is set to 1; 2nd byte tells how far and what direction to go</td>
</tr>
<tr>
<td>1F 98</td>
<td>TFR B,A</td>
<td>2</td>
<td>Inherent</td>
<td>data copied from B to A</td>
</tr>
</tbody>
</table>
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 DYNAMAX 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
  0758 7F 88 91 9A A3 AC B5 00
   \- add more notes
```

or

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

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

```
CBUG: S 0700 0758 0700 NOTES
```

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.

```
: OK
```

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

Type: CLOADM "NOTES"

```
: OK
CLOADM "NOTES"
```

Press ENTER

```
[NOTES blink on and off]
```

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 counter acts 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 stor-
age 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:

\[
\begin{array}{cccccccc}
E & F & H & I & N & Z & V & C \\
\end{array}
\]

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.

\[
\begin{array}{cccccccc}
& & & & & \hline & & \\
\end{array}
\]

location $FF23
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
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
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:

![Arrow keys diagram]

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.

<table>
<thead>
<tr>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label (optional)</td>
<td>Op code mnemonic</td>
<td>Operand (Not always required)</td>
<td>Comment (optional, like a REMARK in BASIC)</td>
</tr>
</tbody>
</table>

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

![Editor interface with LINE INSERT command]

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

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

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

   ![Cursor moves to mnemonic field](image)

2. Type: NAM

   ![Cursor moves to mnemonic field](image)

3. Press the space bar to leave the mnemonic field

   ![Cursor moves right](image)
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)*

```
■ NAM INVERT
LDX #$400 SCREEN ADDRESS
START LDA , X GET CHARACTER
LOOP 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
```
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 loca-
tion 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>
<thead>
<tr>
<th>Source Program</th>
<th>Object Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDX #$400 SCREEN ADDRESS</td>
<td>0767 8E 04 00</td>
</tr>
<tr>
<td>LDA X GET CHARACTER</td>
<td>076A A6 84</td>
</tr>
<tr>
<td>EORA #$40 INVERT COLOR</td>
<td>076C 88 40</td>
</tr>
<tr>
<td>STA ,X+ SAVE; INCREMENT X</td>
<td>076E A7 80</td>
</tr>
<tr>
<td>CMPX #$600 END OF SCREEN</td>
<td>0770 8C 06 00</td>
</tr>
<tr>
<td>BLO LOOP DO WHOLE SCREEN</td>
<td>0773 25 F5</td>
</tr>
<tr>
<td>SWI CALL ABUG</td>
<td>0775 3F</td>
</tr>
<tr>
<td>BRA START AND DO IT AGAIN</td>
<td>0776 20 EF</td>
</tr>
<tr>
<td>END START</td>
<td>0778</td>
</tr>
</tbody>
</table>

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.

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:

```
  1 1 0 1 0 1 0 0
   ↑ ↑ ↑ ↑ ↑ ↑
  E   F   H   I   N   Z   V   C
```

CBUG symbols

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; im-
mediate 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 ad-
dressing 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 Gener-
ator) 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
tion 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 $\text{FF22}$ is loaded with $\text{E0}$

   \[
   \begin{array}{cccccc}
   1 & 1 & 1 & 0 & 0 & 0 \\
   \end{array}
   \]

   - select mode
   - color set 0
   - don't care
   (green, yellow, blue, red)

2. Locations $\text{FFC3}$ and $\text{FFC5}$ 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 $\text{0400}$. It will extend from there up to, but not including, $\text{1C00}$.

   ![Memory Map](image)

   **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.

   ![Color Bar](image)
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:

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
\hline
\text{red} & \text{red} & \text{red} & \text{red}
\end{array}
\]

\[= \$FF\text{ when loaded gives four red points}\]

Four blue points would be

\[
\begin{array}{cccc}
1 & 0 & 1 & 0 \\
\hline
\text{blue} & \text{blue} & \text{blue} & \text{blue}
\end{array}
\]

\[= \$AA\]

Four yellow points

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
\hline
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow}
\end{array}
\]

\[= \$55\]

Four green points

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
\hline
\text{green} & \text{green} & \text{green} & \text{green}
\end{array}
\]

\[= \$00\]

You can mix colors within the byte also:

\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
\hline
\text{red} & \text{green} & \text{blue} & \text{yellow}
\end{array}
\]

\[= \$C9\]
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

<table>
<thead>
<tr>
<th>Block Number</th>
<th>Assembly Language Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ORG $1E00</td>
</tr>
<tr>
<td></td>
<td>LDA #$E0 SELECT MODE</td>
</tr>
<tr>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td>2</td>
<td>CLRA CLEAR SCREEN</td>
</tr>
<tr>
<td></td>
<td>CLR</td>
</tr>
<tr>
<td></td>
<td>LDX #$400</td>
</tr>
<tr>
<td>LOOP1</td>
<td>STD ,X + +</td>
</tr>
<tr>
<td></td>
<td>CMPX #$1C00</td>
</tr>
<tr>
<td></td>
<td>BLO LOOP1</td>
</tr>
</tbody>
</table>
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 #$38 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.

三行画出来，但没有空间隔开它们

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.

   
   0 0 1 1 0 1 1 0
   a b c d
   
   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
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>
<thead>
<tr>
<th>Resolution</th>
<th>Graphics Element</th>
<th>Memory Used</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 × 192</td>
<td>![Element Image]</td>
<td>$0400-$1BFF</td>
<td>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.</td>
</tr>
<tr>
<td>128 × 96</td>
<td>![Element Image]</td>
<td>$0400-$0FFF</td>
<td>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.</td>
</tr>
<tr>
<td>128 × 64</td>
<td>![Element Image]</td>
<td>$0400-$0BFF</td>
<td>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.</td>
</tr>
<tr>
<td>64 × 64</td>
<td>![Element Image]</td>
<td>$0400-$07FF</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.

![VDG mode select diagram]

$1110 = \text{mode } 6C$
$1100 = \text{mode } 3C$
$1010 = \text{mode } 2C$
$1000 = \text{mode } 1C$

Color set select:
- $0 = \text{green, yellow, blue, red}$
- $1 = \text{buff, cyan, magenta, orange}$

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 $V2$

<table>
<thead>
<tr>
<th>Mode</th>
<th>$V2$</th>
<th>$V1$</th>
<th>$V0$</th>
<th>Registers Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>6C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Set $FFC5$ and $FFC3$</td>
</tr>
<tr>
<td>3C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Set $FFC5$</td>
</tr>
<tr>
<td>2C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Set $FFC3$</td>
</tr>
<tr>
<td>1C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Set $FFC1$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Color Mode</th>
<th>Color Set</th>
<th>Instructions</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 6C         | 0         | LDA #$E0
             STA $FF22
             STA $FFC3
             STA $FFC5 | ← VDG mode, color set
|            | 1         | LDA #$EB
             STA $FF22
             STA $FFC3
             STA $FFC5 | ← set V1
| 3C         | 0         | LDA #$C0
             STA $FF22
             STA $FFC5 | ← set V2
|            | 1         | LDA #$C8
             STA $FF22
             STA $FFC5 | ← set V2
| 2C         | 0         | LDA #$A0
             STA $FF22
             STA $FFC3 | ← set V1
|            | 1         | LDA #$A8
             STA $FF22
             STA $FFC3 | ← set V1
| 1C         | 0         | LDA #$80
             STA $FF22
             STA $FFC1 | ← set V0
|            | 1         | LDA #$88
             STA $FF22
             STA $FFC1 | ← set V0

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**

```
0 0 0 1 1 0 1 1 ← data byte
```

- green or buff
- yellow or cyan
- blue or magenta
- red or orange

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

```
/ / / / ⬤ ⬤ ⬤ ⬤ X X X X 0 0 0 0
/ / / / ⬤ ⬤ ⬤ ⬤ X X X X 0 0 0 0
/ / / / ⬤ ⬤ ⬤ ⬤ X X X X 0 0 0 0
```

- green
- yellow
- blue
- red

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

```
/ / ⬤ ⬤ X X 0 0
/ / ⬤ ⬤ X X 0 0
/ / ⬤ ⬤ X X 0 0
```

- green
- yellow
- blue
- red
For mode 3C—color set 0, the data byte shown would produce:

```
/ / □ □ X X 0 0
/ / □ □ X X 0 0
```

green  blue  red
yellow

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

```
/ / □ □ X X 0 0
```

green  blue  red
yellow

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.

<table>
<thead>
<tr>
<th>Mode 6C</th>
<th>Mode 3C</th>
<th>Mode 2C</th>
<th>Mode 1C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1BFF</td>
<td>0FFF</td>
<td>0BFF</td>
<td>07FF</td>
</tr>
</tbody>
</table>

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.

```
1 byte
4 elements
```

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:

![Diagram showing rectangle placement]

We want to store FF at 1008, 1009, 100A, and 100B
C0 at 1028 102B 03 at
   102B
   104B
   106B
   108B
   10AB
   10CB
   10EB
   110B
   112B
   114B
   116B
   118B
   11AB
   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 #$EB ;SELECT 6C
   STA #$FF22
   STA #$FFC3
   STA #$FFC5

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

3) a) LDX #$1008
    LDY #TABLE
    FIRST
    LDA ,Y+
    CMPA #$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
   CMPX #$11E8
   BNE NEXT

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

4) XKEY
   JSR #$A1B1 ;LOOK FOR X KEY
   CMPA #$5B
   BNE XKEY

5) LOOP2
   LDA #$60
   LDX #$400
   STA ,X+
   CMPX #$600
   BLO LOOP2
   LDA #0
   STA #$FF22
   STA #$FFC2
   STA #$FFC4
   RTS ;RETURN TO MONITOR
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).

```
/// /// /// ///
```

memory location $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.

```
LDA ,Y +
```

load A from memory location in Y

then increment Y

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.

```
STA ,X +
```

store A into memory location in X

then increment X

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 86EB  LDA #$8E  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  STAD ,X++
0010 1E12 8C1C00  CMPX #$1C00
0011 1E15 25F9  BLO LOOP1
0012 1E17 8E1008  LDX #$1008
0013 1E1A 109E1E62  LDY #$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  BEO LAST
0031 1E43 BDA1B1  XKEY  JSR #$A1B1
0032 1E46 8158  CMPA #$58
0033 1E48 26F9  BNE XKEY
0034 1E4A B660  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 FFFFFFF  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
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.

<table>
<thead>
<tr>
<th>Mode 6C</th>
<th>Mode 1C</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Mode 6C Diagram" /></td>
<td><img src="image" alt="Mode 1C Diagram" /></td>
</tr>
</tbody>
</table>

sets 4 elements this size

sets 4 elements this size

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.
PROGRAM 7—AN ORANGE RECTANGLE IN MODE 1C

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 1E23 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 8C05DB  CMPX #$5DB
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 B660  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 B600  LDA #$00
0041 1E55 B7FF22  STA #$FF22
0042 1E5B B7FFC0  STA #$FFC0
0043 1E5F 39  RTS
0044 1E5F FFFF  TABLE  FCB #$FF,$FF
0045 1E5E CO03  FCB #$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.

![Blinking Cursor Diagram]

**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.

![Points Diagram]

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 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.
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0600</td>
<td>ORG $01E00</td>
</tr>
<tr>
<td>0002</td>
<td>1E00</td>
<td>LDA #FF88</td>
</tr>
<tr>
<td>0003</td>
<td>1E02</td>
<td>STA $FF22</td>
</tr>
<tr>
<td>0004</td>
<td>1E05</td>
<td>STA $FFC1</td>
</tr>
<tr>
<td>0005</td>
<td>1E08</td>
<td>JSR CLEARG</td>
</tr>
<tr>
<td>0006</td>
<td>1E0B</td>
<td>LDA #FF</td>
</tr>
<tr>
<td>0007</td>
<td>1E0D</td>
<td>STA $400</td>
</tr>
<tr>
<td>0008</td>
<td>1E10</td>
<td>STA $40F</td>
</tr>
<tr>
<td>0009</td>
<td>1E13</td>
<td>STA $7F0</td>
</tr>
<tr>
<td>0010</td>
<td>1E16</td>
<td>STA $7FF</td>
</tr>
<tr>
<td>0011</td>
<td>1E19</td>
<td>JSR XKEY</td>
</tr>
<tr>
<td>0012</td>
<td>1E1C</td>
<td>JSR CLEARA</td>
</tr>
<tr>
<td>0013</td>
<td>1E1F</td>
<td>STA $FFC0</td>
</tr>
<tr>
<td>0014</td>
<td>1E22</td>
<td>LDA #$A8</td>
</tr>
<tr>
<td>0015</td>
<td>1E24</td>
<td>STA $FF22</td>
</tr>
<tr>
<td>0016</td>
<td>1E27</td>
<td>STA $FFC3</td>
</tr>
<tr>
<td>0017</td>
<td>1E2A</td>
<td>JSR CLEARG</td>
</tr>
<tr>
<td>0018</td>
<td>1E2D</td>
<td>LDA #$FF</td>
</tr>
<tr>
<td>0019</td>
<td>1E2F</td>
<td>STA $400</td>
</tr>
<tr>
<td>0020</td>
<td>1E32</td>
<td>STA $41F</td>
</tr>
<tr>
<td>0021</td>
<td>1E35</td>
<td>STA $BBE0</td>
</tr>
<tr>
<td>0022</td>
<td>1E38</td>
<td>STA $BBFF</td>
</tr>
<tr>
<td>0023</td>
<td>1E3B</td>
<td>JSR XKEY</td>
</tr>
<tr>
<td>0024</td>
<td>1E3E</td>
<td>JSR CLEARA</td>
</tr>
<tr>
<td>0025</td>
<td>1E41</td>
<td>STA $FFC2</td>
</tr>
<tr>
<td>0026</td>
<td>1E44</td>
<td>LDA #$C8</td>
</tr>
<tr>
<td>0027</td>
<td>1E46</td>
<td>STA $FF22</td>
</tr>
<tr>
<td>0028</td>
<td>1E49</td>
<td>STA $FFC5</td>
</tr>
<tr>
<td>0029</td>
<td>1E4C</td>
<td>JSR CLEARG</td>
</tr>
<tr>
<td>0030</td>
<td>1E4F</td>
<td>LDA #$FF</td>
</tr>
<tr>
<td>0031</td>
<td>1E51</td>
<td>STA $400</td>
</tr>
<tr>
<td>0032</td>
<td>1E54</td>
<td>STA $41F</td>
</tr>
<tr>
<td>0033</td>
<td>1E57</td>
<td>STA $FE0</td>
</tr>
<tr>
<td>0034</td>
<td>1E5A</td>
<td>STA #$FF</td>
</tr>
<tr>
<td>0035</td>
<td>1E5D</td>
<td>JSR XKEY</td>
</tr>
<tr>
<td>0036</td>
<td>1E60</td>
<td>JSR CLEARA</td>
</tr>
<tr>
<td>0037</td>
<td>1E63</td>
<td>STA $FFC4</td>
</tr>
<tr>
<td>0038</td>
<td>1E66</td>
<td>LDA #$E8</td>
</tr>
<tr>
<td>0039</td>
<td>1E68</td>
<td>STA $FF22</td>
</tr>
<tr>
<td>0040</td>
<td>1E6B</td>
<td>STA $FFC3</td>
</tr>
<tr>
<td>0041</td>
<td>1E6E</td>
<td>STA $FFC5</td>
</tr>
<tr>
<td>0042</td>
<td>1E71</td>
<td>JSR CLEARG</td>
</tr>
<tr>
<td>0043</td>
<td>1E74</td>
<td>LDA #$FF</td>
</tr>
<tr>
<td>0044</td>
<td>1E76</td>
<td>STA $400</td>
</tr>
<tr>
<td>0045</td>
<td>1E79</td>
<td>STA $41F</td>
</tr>
<tr>
<td>0046</td>
<td>1E7C</td>
<td>STA $1BE0</td>
</tr>
<tr>
<td>0047</td>
<td>1E7F</td>
<td>STA $1BBF</td>
</tr>
<tr>
<td>0048</td>
<td>1E82</td>
<td>JSR XKEY</td>
</tr>
<tr>
<td>0049</td>
<td>1E85</td>
<td>JSR CLEARA</td>
</tr>
<tr>
<td>0050</td>
<td>1E88</td>
<td>STA $FFC2</td>
</tr>
<tr>
<td>0051</td>
<td>1E8B</td>
<td>STA $FFC4</td>
</tr>
<tr>
<td>0052</td>
<td>1E8E</td>
<td>RTS</td>
</tr>
</tbody>
</table>

90
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

<table>
<thead>
<tr>
<th>Instruction and Addressing Mode</th>
<th>Program Where Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>BEQ relative</td>
<td></td>
</tr>
<tr>
<td>BLO relative</td>
<td></td>
</tr>
<tr>
<td>BNE relative</td>
<td>X</td>
</tr>
<tr>
<td>BPL relative</td>
<td>X</td>
</tr>
<tr>
<td>BRA relative</td>
<td>X</td>
</tr>
<tr>
<td>CLRA inherent</td>
<td></td>
</tr>
<tr>
<td>CLRB inherent</td>
<td>X</td>
</tr>
<tr>
<td>CMPA immediate</td>
<td>X</td>
</tr>
</tbody>
</table>
### Table 4-3. (continued)

<table>
<thead>
<tr>
<th>Instruction and Addressing Mode</th>
<th>Program Where Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>CMPB</td>
<td>immediate</td>
</tr>
<tr>
<td>CMPX</td>
<td>immediate</td>
</tr>
<tr>
<td>DECB</td>
<td>inherent</td>
</tr>
<tr>
<td>DEX</td>
<td>inherent</td>
</tr>
<tr>
<td>END</td>
<td>pseudo-op</td>
</tr>
<tr>
<td>EORA</td>
<td>immediate</td>
</tr>
<tr>
<td>FCB</td>
<td>pseudo-op</td>
</tr>
<tr>
<td>INCA</td>
<td>inherent</td>
</tr>
<tr>
<td>INCB</td>
<td>inherent</td>
</tr>
<tr>
<td>JSR</td>
<td>extended</td>
</tr>
<tr>
<td>LDA</td>
<td>immediate</td>
</tr>
<tr>
<td>LDA ,X+</td>
<td>indexed</td>
</tr>
<tr>
<td>LDA ,Y+</td>
<td>indexed</td>
</tr>
<tr>
<td>LDB</td>
<td>immediate</td>
</tr>
<tr>
<td>LDB ,Y+</td>
<td>indexed</td>
</tr>
<tr>
<td>LDX</td>
<td>immediate</td>
</tr>
<tr>
<td>LDY</td>
<td>immediate</td>
</tr>
<tr>
<td>LEAX n,X</td>
<td>indexed</td>
</tr>
<tr>
<td>NOP</td>
<td>inherent</td>
</tr>
<tr>
<td>ORG</td>
<td>pseudo-op</td>
</tr>
<tr>
<td>RTS</td>
<td>inherent</td>
</tr>
<tr>
<td>STA</td>
<td>extended</td>
</tr>
<tr>
<td>STA ,X+</td>
<td>indexed</td>
</tr>
<tr>
<td>STB</td>
<td>extended</td>
</tr>
<tr>
<td>STD ,X+</td>
<td>indexed</td>
</tr>
<tr>
<td>SWI</td>
<td>inherent</td>
</tr>
<tr>
<td>TFR A,B</td>
<td>inherent</td>
</tr>
<tr>
<td>TFR B,A</td>
<td>inherent</td>
</tr>
</tbody>
</table>

**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.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Resolution</th>
<th>Element</th>
<th>Memory Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>6C</td>
<td>128 × 192</td>
<td></td>
<td>$400-1BFF 6K</td>
</tr>
<tr>
<td>3C</td>
<td>128 × 96</td>
<td></td>
<td>$400-FFF 3K</td>
</tr>
<tr>
<td>2C</td>
<td>128 × 64</td>
<td></td>
<td>$400-BFF 2K</td>
</tr>
<tr>
<td>1C</td>
<td>64 × 64</td>
<td></td>
<td>$400-7FF 1K</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mode</th>
<th>Set Registers</th>
<th>Data in $FF22</th>
</tr>
</thead>
<tbody>
<tr>
<td>6C</td>
<td>$FFC5, $FFC3</td>
<td>E0-color set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E8-color set 1</td>
</tr>
<tr>
<td>3C</td>
<td>$FFC5</td>
<td>C0-color set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C8-color set 1</td>
</tr>
<tr>
<td>2C</td>
<td>$FFC3</td>
<td>A0-color set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A8-color set 1</td>
</tr>
<tr>
<td>1C</td>
<td>$FFC1</td>
<td>80-color set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>88-color set 1</td>
</tr>
</tbody>
</table>

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

```
[ ] [ ] [ ] [ ]
```
four color bit pairs

<table>
<thead>
<tr>
<th>Bits</th>
<th>Color Set 0</th>
<th>Color Set 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>green</td>
<td>buff</td>
</tr>
<tr>
<td>01</td>
<td>yellow</td>
<td>cyan</td>
</tr>
<tr>
<td>10</td>
<td>blue</td>
<td>magenta</td>
</tr>
<tr>
<td>11</td>
<td>red</td>
<td>orange</td>
</tr>
</tbody>
</table>
• 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>
<thead>
<tr>
<th>Resolution</th>
<th>Graphic Element</th>
<th>Memory Used</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 x 192</td>
<td>[ ]</td>
<td>0400-1BFF</td>
<td>Mode 6R; highest resolution of all; each element is one unit wide and one unit high; 6K of memory used</td>
</tr>
<tr>
<td>128 x 192</td>
<td>[ ]</td>
<td>0400-0FFF</td>
<td>Mode 3R; each element is two units wide and one unit high; 3K of memory used</td>
</tr>
<tr>
<td>128 x 96</td>
<td>[ ]</td>
<td>0400-09FF</td>
<td>Mode 2R; each element is two units wide and two units high; 1.5K of memory used</td>
</tr>
<tr>
<td>128 x 64</td>
<td>[ ]</td>
<td>0400-07FF</td>
<td>Mode 1R; each element is two units wide and three units high; 1K of memory used</td>
</tr>
</tbody>
</table>
For each of these modes, green and black are the colors used in color set 0.

\[
\text{Color Set 0} \quad \begin{cases} 
0 &= \text{black} \\
1 &= \text{green}
\end{cases}
\]

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

\[
\text{Color Set 1} \quad \begin{cases} 
0 &= \text{black} \\
1 &= \text{buff}
\end{cases}
\]

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**

<table>
<thead>
<tr>
<th>Data byte</th>
<th>Color set 0</th>
<th>Mode 6R</th>
<th>Mode 1R</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00000000</td>
<td><img src="image" alt="Image showing all black" /></td>
<td><img src="image" alt="Image showing black in middle" /></td>
</tr>
<tr>
<td>81</td>
<td>10000001</td>
<td><img src="image" alt="Image showing green on ends" /></td>
<td><img src="image" alt="Image showing black in middle" /></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mode</th>
<th>Color Set</th>
<th>Instructions</th>
</tr>
</thead>
</table>
| 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 Program 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:

![Diagram of pistons and shaft]

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

![Diagram showing memory locations 0400 to 07FF]

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,$18,$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.

\[
\text{LDA } $FF00 \quad \text{used to read input port}
\]

load accumulator

\[
\text{with the value held in memory $FF00}
\]
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.

\[
\text{Keyboard input and output bits are enabled by a 0 and disabled by a 1.}
\]

Therefore

\[
\begin{align*}
\text{LDA } \#\$FE & \quad \text{write a 0 to bit 0} \quad 7 6 5 4 3 2 1 0 \\
\text{STA } \$FF02 & \quad \text{of output port} \quad 1 1 1 1 1 1 0
\end{align*}
\]

\[
\begin{align*}
\text{LDA } \$FF00 & \quad \text{read input port} \quad 7 6 5 4 3 2 1 0 \\
\text{CMPA } \#\$F7 & \quad \text{Is bit 3 = 0?} \quad 1 1 1 1 0 1 1 1
\end{align*}
\]

\[
\text{BNE DRAW} \quad \text{if X key not down, go back to draw piston}
\]

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 DRA W 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 etc.
       JSR OFF
       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 B690 LDA #$90
0003 1E02 B7FF22 STA $FF22
0004 1E05 B7FFC1 STA $FFC1
0005 1E08 4F CLRA
0006 1E09 5F CLR8
0007 1E0A BE0400 LDX #$400
0008 1E0D EDB1 LOOP1 STD ,X++
0009 1E0F BC1C00 CMPX #$1C00
0010 1E12 25F9 BLO LOOP1
0011 1E14 BE05C7 LDX #$5C7
0012 1E17 10BE1EA4 LDY #TABLE1
0013 1E1B C603 DOWN LDB #3
0014 1E1D A6A0 RIGHT LDA ,Y+
0015 1E1F A7B0 STA ,X+
0016 1E21 5A DECB
0017 1E22 26F9 BNE RIGHT
0018 1E24 300D LEAX $D,X
0019 1E26 BCO647 CMPX #$647
0020 1E29 26F0 BNE DOWN
0021 1E2B BE0598 DRAW LDX #$598 ;DRAW PISTON
0022 1E2E BD1E86 JSR ON
0023 1E31 BE0598 LDX #$598 ;ERASE PISTON
0024 1E34 BD1E99 JSR OFF
0025 1E37 BE05B0 LDX #$5B0 ;DRAW PISTON
0026 1E3A BD1E86 JSR ON
0027 1E3D BE05B8 LDX #$5B8 ;ERASE PISTON
0028 1E40 BD1E99 JSR OFF
0029 1E43 BE05DB LDX #$5DB ; ETC.
0030 1E46 BD1E86 JSR ON
0031 1E49 BE05DB LDX #$5DB
0032 1E4C BD1E99 JSR OFF
0033 1E4F BE05B0 LDX #$5B0
```
0034 1E52 BD1E86    JSR DN
0035 1E95 0E05B8    LDS #5B8
0036 1E58 BD1E99    JSR OFF
0037 1E5B B6FE    LDA #5FE
0038 1E5D B7FF02    STA #FF02
0039 1E60 B6FF00    STA #FF00
0040 1E63 81F7    CMPA #5F7
0041 1E65 26C4    BNE DRAW
0042 1E67 B660    TEXT LDA #560
0043 1E69 BE0400    LDS #400
0044 1E6C A7B0    LOOP2 STA ,X+
0045 1E6E B00600    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 BE4000    DELAY LDS #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 10BE1EBC    ON LDY #TABLE2
0059 1E8A C605    LDB #5
0060 1E8C A6A0    PIST LDA ,Y+
0061 1E8E A7B0    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 A7B0    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 1EA8 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,$18,$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>
<thead>
<tr>
<th>Write to Register</th>
<th>Sets Offset</th>
<th>Clears Offset</th>
<th>Offset Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFC6$</td>
<td>—</td>
<td>F0</td>
<td>—</td>
</tr>
<tr>
<td>$FFC7$</td>
<td>F0</td>
<td>—</td>
<td>$200$</td>
</tr>
<tr>
<td>$FFC8$</td>
<td>—</td>
<td>F1</td>
<td>—</td>
</tr>
<tr>
<td>$FFC9$</td>
<td>F1</td>
<td>—</td>
<td>$400$</td>
</tr>
<tr>
<td>$FFCA$</td>
<td>—</td>
<td>F2</td>
<td>—</td>
</tr>
<tr>
<td>$FFCB$</td>
<td>F2</td>
<td>—</td>
<td>$800$</td>
</tr>
<tr>
<td>$FFCC$</td>
<td>—</td>
<td>F3</td>
<td>—</td>
</tr>
<tr>
<td>$FFCD$</td>
<td>F3</td>
<td>—</td>
<td>$1000$</td>
</tr>
<tr>
<td>$FFCE$</td>
<td>—</td>
<td>F4</td>
<td>—</td>
</tr>
<tr>
<td>$FFCF$</td>
<td>F4</td>
<td>—</td>
<td>$2000$</td>
</tr>
<tr>
<td>$FFD0$</td>
<td>—</td>
<td>F5</td>
<td>—</td>
</tr>
<tr>
<td>$FFD1$</td>
<td>F5</td>
<td>—</td>
<td>$4000$</td>
</tr>
<tr>
<td>$FFD2$</td>
<td>—</td>
<td>F6</td>
<td>—</td>
</tr>
<tr>
<td>$FFD3$</td>
<td>F6</td>
<td>—</td>
<td>$8000$</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Starting Display Address</th>
<th>Write to Address(es)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>none</td>
</tr>
<tr>
<td>0200</td>
<td>$FFC7</td>
</tr>
<tr>
<td>0400</td>
<td>$FFC9</td>
</tr>
<tr>
<td>0600</td>
<td>$FFC7, $FFC9</td>
</tr>
<tr>
<td>0800</td>
<td>$FFCB</td>
</tr>
<tr>
<td>0A00</td>
<td>$FFCB, $FFC7</td>
</tr>
<tr>
<td>0C00</td>
<td>$FFCB, $FFC9</td>
</tr>
<tr>
<td>0E00</td>
<td>$FFCB, $FFC9, $FFC7</td>
</tr>
<tr>
<td>1000</td>
<td>$FFCD</td>
</tr>
<tr>
<td>1200</td>
<td>$FFCD, $FFC7</td>
</tr>
<tr>
<td>1400</td>
<td>$FFCD, $FFC9</td>
</tr>
<tr>
<td>1600</td>
<td>$FFCD, $FFC9, $FFC7</td>
</tr>
<tr>
<td>1800</td>
<td>$FFCD, $FFCB</td>
</tr>
<tr>
<td>1A00</td>
<td>$FFCD, $FFCB, $FFC7</td>
</tr>
<tr>
<td>1C00</td>
<td>$FFCD, $FFCB, $FFC9</td>
</tr>
<tr>
<td>1E00</td>
<td>$FFCD, $FFCB, $FFC9, $FFC7</td>
</tr>
<tr>
<td>2000</td>
<td>$FFCF</td>
</tr>
<tr>
<td>2200</td>
<td>$FFCF, $FFC7</td>
</tr>
<tr>
<td>2400</td>
<td>$FFCF, $FFC9</td>
</tr>
<tr>
<td>2600</td>
<td>$FFCF, $FFC9, $FFC7</td>
</tr>
<tr>
<td>2800</td>
<td>$FFCF, $FFCB</td>
</tr>
<tr>
<td>2A00</td>
<td>$FFCF, $FFCB, $FFC7</td>
</tr>
<tr>
<td>2C00</td>
<td>$FFCF, $FFCB, $FFC9</td>
</tr>
<tr>
<td>2E00</td>
<td>$FFCF, $FFCB, $FFC9, $FFC7</td>
</tr>
<tr>
<td>3000</td>
<td>$FFCF, $FFCD</td>
</tr>
<tr>
<td>3200</td>
<td>$FFCF, $FFCD, $FFC7</td>
</tr>
<tr>
<td>3400</td>
<td>$FFCF, $FFCD, $FFC9</td>
</tr>
<tr>
<td>3600</td>
<td>$FFCF, $FFCD, $FFC9, $FFC7</td>
</tr>
<tr>
<td>3800</td>
<td>$FFCF, $FFCD, $FFCB</td>
</tr>
<tr>
<td>3A00</td>
<td>$FFCF, $FFCD, $FFCB, $FFC7</td>
</tr>
<tr>
<td>3C00</td>
<td>$FFCF, $FFCD, $FFCB, $FFC9</td>
</tr>
<tr>
<td>3E00</td>
<td>$FFCF, $FFCD, $FFCB, $FFC9, $FFC7</td>
</tr>
<tr>
<td>4000</td>
<td>$FFD0</td>
</tr>
</tbody>
</table>
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
RMB 1
LDA #$98
STA $FF22
STA $FFC1
```

Part 2—Clear Screen

```
CLRA
CLRB
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  \text{ \hspace{1em} turn on block 1} \\
BRA AGIN  \text{ \hspace{1em} (FFC7, FFC9 set)} \\
\hspace{1em} \text{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 \} \text{ \hspace{1em} get ready for text mode} 
STA $FFC0 \} 
STA $FFC6 \} \text{ \hspace{1em} 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.

```
ROW  RMB  1
```

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>
<thead>
<tr>
<th>Instruction and Addressing Mode</th>
<th>Program and Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA extended</td>
<td>Program 9; accumulator A is loaded with the data in the extended address</td>
</tr>
<tr>
<td></td>
<td>Example: LDA $FF00</td>
</tr>
<tr>
<td>FCB pseudo-op</td>
<td>Program 9; used to indicate single byte items in a data table</td>
</tr>
<tr>
<td></td>
<td>Example: FCB 1,0,$80,1,0,$80</td>
</tr>
<tr>
<td>RMB n</td>
<td>Program 11; $80,1,0,$80</td>
</tr>
</tbody>
</table>

**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.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Resolution</th>
<th>Element</th>
<th>Memory Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>6R</td>
<td>256×192</td>
<td>□</td>
<td>6K</td>
</tr>
<tr>
<td>3R</td>
<td>128×192</td>
<td>□□</td>
<td>3K</td>
</tr>
<tr>
<td>2R</td>
<td>128×96</td>
<td>□□□□□□□□</td>
<td>1.5K</td>
</tr>
<tr>
<td>1R</td>
<td>128×64</td>
<td>□□□□□□□□</td>
<td>1K</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mode</th>
<th>Set Registers</th>
<th>Data in $FF22</th>
</tr>
</thead>
</table>
| 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.

<table>
<thead>
<tr>
<th>Offset Register</th>
<th>Write to Address</th>
<th>Offset Added</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>To Clear</td>
<td>To Set</td>
</tr>
<tr>
<td>F0</td>
<td>FFC6</td>
<td>FFC7</td>
</tr>
<tr>
<td>F1</td>
<td>FFC8</td>
<td>FFC9</td>
</tr>
<tr>
<td>F2</td>
<td>FFCA</td>
<td>FFCB</td>
</tr>
<tr>
<td>F3</td>
<td>FFCC</td>
<td>FFCD</td>
</tr>
<tr>
<td>F4</td>
<td>FFCE</td>
<td>FFCF</td>
</tr>
<tr>
<td>F5</td>
<td>FFD0</td>
<td>FFD1</td>
</tr>
<tr>
<td>F6</td>
<td>FFD2</td>
<td>FFD3</td>
</tr>
</tbody>
</table>

• 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.

   LDA
   STA
   STA
   STA

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)
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
PRINTIT 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
CLR B
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 B660 START LDA $60
0010 3002 B60400 LDX $400
0011 3005 A780 CLEAR STA ,X+
0012 3007 8C0600 CMPX $600
0013 300A 25F9 BLO CLEAR
0014 300C B660 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 CLR B
0022 3020 BDA9A2 JSR SELMUX
0023 3023 33BD0052 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 303B BDA7D3 JSR WAIT
0034 303B 20EA BRA LOOP
0035 303D 31F LOOP2 LEA Y -1,X
0036 305F 26E2 BNE LOOP1
0037 3041 20BD BRA START
0038 3043 3F QUIT SWI
0039 3044 B63F QUEST LDA #’?
0040 3046 BDA30A JSR PRINIT
0041 3049 BDA1B1 JSR POLCAT
0042 304C 8123 CMP #’#
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

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Delay</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>05</td>
<td>a short “pip”</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>lower “pip”</td>
</tr>
<tr>
<td>05</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>C0</td>
<td>very low</td>
</tr>
<tr>
<td>05</td>
<td>F0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>longer sound with 40 tone</td>
</tr>
<tr>
<td>20</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>40</td>
<td>very long</td>
</tr>
</tbody>
</table>

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  B6E0   Start LDA #4E0
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  353D004A  Leau COLOR,PCR
0020 1E1E  8E1208   Num2 Ldx #1208
0021 1E21  BD1E5C  Jsr Display
0022 1E24  8E1228  Ldx #1228
0023 1E27  BD1E5C  Jsr Display
0024 1E2A  8E1248  Ldx #1248
0025 1E2D  BD1E5C  Jsr Display
0026 1E30  3341   Leau I,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 A7B0  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  BEO QUIT
0047 1E62 A7B0  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 551530  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 33BD001E  LOOP1  LEAU TABLE,PCR
0060 1E81 A6C0  LOOP  LDA ,U+
0061 1E83 2712  BEO LOOP2
0062 1E85 48  LSLA
0063 1E86 48  LSLA
0064 1E87 BA02  ORA #2
0065 1E89 B7FF20  STA DA
0066 1E8C 4F  CLRA
0067 1E8D F60600  LDB DELAY
0068 1E90 1F01  FFR D,X
0069 1E92 BDA7D3  JSR WAIT
0070 1E95 20EA  BNA LOOP
0071 1E97 7A0601  LOOP2  DEC CYCLE
0072 1E99 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 1EAA 0D1116  FCB 13,17,22
0078 1EAB 1C232C  FCB 23,35,44
0079 1EAF 3F00-  FCB 63,0
0080 1EAD  END

AUDON 6976  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. 

   ![Red bar with low tone]

2. 

   ![Blue bar with higher tone]

3. 

   ![Yellow bar with still higher tone]

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 6G. 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.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F30</td>
<td>00000011</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F50</td>
<td>00000011</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F70</td>
<td>00000011</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F90</td>
<td>00000011</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0FB0</td>
<td>00000011</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0FD0</td>
<td>00000011</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0FF0</td>
<td>00000011</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>00000011</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
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.
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 $$40
0054 2075 B70600 STA CYCLE
0055 2079 BDA976 JSR AUDON
0056 207B 5F CLRB
0057 207C BDA9A2 JSR SELMUX
0058 207F 33BD007B LOOP1 LEAU TABLE3,PCR
0059 2083 A6C0 LOOP3 LDA $U+
0060 2085 2711 BEQ LOOP2
0061 2087 48 LSLA
0062 2088 48 LSLA
0063 2089 8A02 ORA #2
0064 208B B7FF20 STA DA
0065 208E 4F CLRA
0066 208F C610 LDB $$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 $$0F30
0076 20A5 4F CLRA
0077 20A6 5F CLRB
0078 20A7 EDB4 LOOP3 STD $X
0079 20A9 30B820 LEAX $20,X
0080 20AC 8C10E0 CMPX $$10E0
0081 20AF 25F6 BLO LOOP3
0082 20B1 3510 PULS X
0083 20B3 39 RTS
0084 20B4 EC01 DRAW LDD $X++
0085 20B6 EDB1 LOOP4 STD [,$Y++]
0086 20B9 7A0601 DEC COUNT
0087 20BB 26F9 BNE LOOP4
0088 20BD 39 RTS
0089 20BE 030003FC03 TABLE1 FCB $3,0,0,$FC,3,0,$FF,0
0090 20C5 020002AB02 FCB 2,0,2,$AB,2,0,$AA,0
0091 20CE 0100015401 FCB 1,0,1,$54,1,0,$55,0

137
0092 20D6 0F300F500F TABLE2 FCB $F, $50, $F, $50, $F, $70
0093 20DE 0F900FB00F FCB $F, $90, $F, $80, $F, $D0
0094 20E2 OFF01010 FCB $F, $FO, $10, $10
0095 20E6 OFF01010 FCB $F, $FO, $10, $10
0096 20EA OFF0101010 FCB $F, $FO, $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, $FO, $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 60 201A INDEX 0612
LOOP 2083 LOOP1 207F LOOP2 2098 LOOP3 20A7
LOOP4 20B6 NUM1 20A3 ROUND 20C2 SELMUX A942
SOUND 2071 START 2000 TABLE1 20BE TABLE2 20D6
TABLE3 20FE WAIT A7D3

Figure 6-3. Program 14—Rotating Bar

![Diagram of byte 1 and byte 2]

00 11 10 01
green red blue yellow

<table>
<thead>
<tr>
<th>red</th>
<th>blue</th>
<th>yellow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C0</td>
<td>0A 80</td>
<td>05 40</td>
</tr>
<tr>
<td>0F C0</td>
<td>0A 80</td>
<td>05 40</td>
</tr>
<tr>
<td>FF C0</td>
<td>AA 80</td>
<td>55 40</td>
</tr>
<tr>
<td>FF C0</td>
<td>AA 80</td>
<td>55 40</td>
</tr>
<tr>
<td>00 00</td>
<td>00 00</td>
<td>00 00</td>
</tr>
<tr>
<td>C0 C0</td>
<td>80 80</td>
<td>40 40</td>
</tr>
</tbody>
</table>

Figure 6-4. Data Bytes Defining Trucks

138
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.

![Diagram showing byte 1 and byte 2 with color codes]

<table>
<thead>
<tr>
<th>00</th>
<th>11</th>
<th>10</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>red</td>
<td>blue</td>
<td>yellow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>red</th>
<th>blue</th>
<th>yellow</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 00</td>
<td>20 00</td>
<td>10 00</td>
</tr>
<tr>
<td>30 C0</td>
<td>20 80</td>
<td>10 40</td>
</tr>
<tr>
<td>FF C0</td>
<td>AA 80</td>
<td>55 40</td>
</tr>
<tr>
<td>FF C0</td>
<td>AA 80</td>
<td>55 40</td>
</tr>
<tr>
<td>30 C0</td>
<td>20 80</td>
<td>10 40</td>
</tr>
<tr>
<td>30 00</td>
<td>20 00</td>
<td>10 00</td>
</tr>
</tbody>
</table>

*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.
DA   EQU  #$FF20
AUDON  EQU  #$A976
SELMUX  EQU  #$A9A2
WAIT  EQU  #$A7D3
ORG  #$2000

START LDA  #$E0
STA  #$FF22
STA  #$FFC3
STA  #$FFC5
STA  #$FFC7
JSR  ERASE
LDA  #$3
STA  COLOR
LDY  #TABLE
LDA  #$20
STA  INDEX
LDX  #$1CFE
JSR  MOVE
LEAY  $18,Y
DEC  COLOR
BNE  GO
LDA  #$0
STA  FF22
STA  FF22
STA  FF22
STA  FF22
STA  FF22
STA  FF22
STA  FF22
STA  FF22
SWI
PSHS  X
CLRA
CLRB
LDX  #$600
STD ,X++
CMPX  #$1E00
BLO  NUM1
PULS  X
RTS
LDA  #$6
STA  COUNT
JSR  DRAW
JSR  ERASE
DEC  INDEX
BNE  MOVE
RTS
LDD ,Y++
STD ,X
LDD ,Y++
STD  #1680,X
LEAX  #$20,X
DEC  COUNT
BNE  DRAW
LEAY  #$18,Y
LEAX  #$C1,X
PSHS  U,X,Y,D
LDA  #$3
STA  CYCLE
0056  207E  BDA976  JSR  AUDON
0057  2081  5F  CLR8
0058  2082  BDA9A2  JSR  SELMUX
0059  2085  33BD0069  LOOP1  LEAU  TABLE2,PCR
0060  2089  A6C0  LOOP  LDA  ,U+
0061  208B  2711  BE0  LOOP2
0062  208E  4B  LSLA
0063  208E  48  .  LSLA
0064  208F  B602  DRA  #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  20A5  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  FFCAAB0  FCB  $FF, $C0, $AA, $80
0082  20B6  FFCAAB0  FCB  $FF, $C0, $AA, $80
0083  20BA  00002080  FCB  0, 0, $20, $80
0084  20BE  C6CO2000  FCB  $C0, $C0, $20, 0
0085  20C2  0A801000  FCB  $A, $80, $10, 0
0086  20C6  0A801040  FCB  $A, $80, $10, $40
0087  20CA  AAB05540  FCB  $AA, $80, $55, $40
0088  20CE  A805540  FCB  $AA, $80, $55, $40
0089  20D2  00001040  FCB  0, 0, $10, $40
0090  20D6  08081000  FCB  $80, $80, $10, 0
0091  20DA  05403000  FCB  5, $40, $30, 0
0092  20DE  054030CC  FCB  5, $40, $30, $00
0093  20E2  5540FFCC  FCB  $55, $40, $FF, $00
0094  20E6  5540FFCC  FCB  $55, $40, $FF, $00
0095  20EA  000030CC  FCB  0, 0, $30, $00
0096  20EE  404030CC  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  1610213  FCB  22,28,35
0101  20FD  2C3F00  FCB  44,63,0
0102  2100  END

AUDON  A976  COLOR  20A6  COUNT  20A7  CYCLE  20AB
DA  FF20  DRAW  205E  ERASE  203C  60  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

141
In Program 15 (Figure 6-6), the MOVE and DRA W 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 DRA W 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    ;turns off previous $E0
STA $FF22
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  \{ setting up the mode
  STA $FF22 and color set
  STA $FFC3
  STA $FFC7  \rightarrow add $200 offset

NUM2
  LDX #$1208
  JSR DISPLA
  LDX #$1228
  JSR DISPLA \{ OMIT these lines since
  JSR DISPLA  \} mode 2C draws 3 times as
  LDX #$1248 mode 6C
  JSR DISPLA

NUM3

  STA $FF22  \rightarrow 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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Program Where Used and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFR D,Y</td>
<td>12 used to transfer data from the register D to register Y</td>
</tr>
<tr>
<td>LEAU TABLE,PCR</td>
<td>12 calculates distance (in bytes) to location of TABLE and inserts these values in the object code</td>
</tr>
<tr>
<td>LDA ,U +</td>
<td>12 loads accumulator A from stack, increases stack pointer by 1</td>
</tr>
<tr>
<td>LSLA</td>
<td>12 shifts all bits in A one place to the left, most significant bit is lost</td>
</tr>
</tbody>
</table>
ORA \#n
TFR D,X
LEAY n,Y
LDA #'?
CMPA #'
ADDA \#n
ANDA \#n
ABA
LEAU n,U
PSHS U,X,D
PULS U,X,D
STD ,Y++
LDD ,Y++
STD -$1680,X

12 logical OR with accumulator with the value n
12 transfer data from register D to register X
12 increment the value in Y register by the value n
12 load accumulator A with the ASCII code of the question mark
12 compare value in A with the ASCII code of the number sign
12 add the value n to accumulator A
12 logical AND accumulator A with n
12 add accumulator B to accumulator A
13 increment stack pointer by n
13 push values in U,X and D onto the stack
13 pull values of stack and place in U,X, and D
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
15 load register D from the location stored in Y and Y+1, increment Y by 2
15 store the value in D into $1680 less than the location held in X

Chapter Test

1. Describe the purpose of the statement
   \textsc{delay} RMB 1

2. Describe the purpose of the subroutines
   a. \textsc{audon}
   b. \textsc{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

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Data</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F30</td>
<td></td>
<td></td>
<td>0F31</td>
</tr>
<tr>
<td>0F50</td>
<td></td>
<td></td>
<td>0F51</td>
</tr>
<tr>
<td>0F70</td>
<td></td>
<td></td>
<td>0F71</td>
</tr>
<tr>
<td>0F90</td>
<td></td>
<td></td>
<td>0F91</td>
</tr>
<tr>
<td>0FB0</td>
<td></td>
<td></td>
<td>0FB1</td>
</tr>
<tr>
<td>0FD0</td>
<td></td>
<td></td>
<td>0FD1</td>
</tr>
<tr>
<td>0FF0</td>
<td></td>
<td></td>
<td>0FF1</td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td>1011</td>
</tr>
<tr>
<td>1030</td>
<td></td>
<td></td>
<td>1031</td>
</tr>
<tr>
<td>1050</td>
<td></td>
<td></td>
<td>1051</td>
</tr>
<tr>
<td>1070</td>
<td></td>
<td></td>
<td>1071</td>
</tr>
<tr>
<td>1090</td>
<td></td>
<td></td>
<td>1091</td>
</tr>
<tr>
<td>10B0</td>
<td></td>
<td></td>
<td>10B1</td>
</tr>
<tr>
<td>10D0</td>
<td></td>
<td></td>
<td>10D1</td>
</tr>
</tbody>
</table>
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:

\[
\begin{array}{c}
\text{\= blue} \\
\text{\= yellow} \\
\text{\= red}
\end{array}
\]

Use hex digits.
8. Revise Program 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.
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.  

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>02</td>
<td>80</td>
<td>09</td>
</tr>
<tr>
<td>60</td>
<td>25</td>
<td>58</td>
<td>97</td>
<td>D6</td>
</tr>
<tr>
<td>25</td>
<td>58</td>
<td>0A</td>
<td>A0</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>96</td>
<td>96</td>
<td>AA</td>
<td>96</td>
</tr>
<tr>
<td>96</td>
<td>96</td>
<td>AA</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>AA</td>
<td>96</td>
<td>96</td>
<td>AA</td>
<td>00</td>
</tr>
<tr>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Hex Values</th>
<th>Binary Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>0A A0</td>
<td>00010101 10100000</td>
</tr>
<tr>
<td>AA AA</td>
<td>10101010 10101010</td>
</tr>
<tr>
<td>AA AA</td>
<td>10101010 10101010</td>
</tr>
<tr>
<td>08 20</td>
<td>00001000 00100000</td>
</tr>
<tr>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>Byte 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 00 10 10</td>
<td>10 10 00 00</td>
</tr>
<tr>
<td>10 10 10 10</td>
<td>10 10 10 10</td>
</tr>
<tr>
<td>10 10 10 10</td>
<td>10 10 10 10</td>
</tr>
<tr>
<td>00 00 10 00</td>
<td>00 10 00 00</td>
</tr>
<tr>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

Figure 7-1. Flying Saucer Shape Table

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

```assembly
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 HORIZ. We will use the following program segment to read and store the joystick values.

```assembly
JSR JOYST  ← read joystick
LDA $15A  ← get horizontal data
STA HORIZ  ← 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 multiply vertical by 96
MUL TFR D,Y save result in Y
LDB HORIZ get horizontal position
LSRB LDB HORIZ 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 VIDE B,E 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 $A000
0002 0600 KEYIN EQU $A000
0003 0600 VIDE B,E EQU $600
0004 0600 ORG VIDE B,E+$1800
0005 1E00 B600 START LDA #8E0
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 CLR B
0012 1E10 8E0600 LDX #VID B,E
0013 1E13 EB81 LOOP STD , X++
0014 1E15 9C1E00 CMPX #VID B,E+$1800
0015 1E18 25F9 BLO LOOP
0016 1E1A F71EBF STB HORIZ
0017 1E1D F71E90  STB VERTJ
0018 1E20 AD9FA000  MOVE JSR [KEYINJ]
0019 1E24 B15B  CMPA #$X
0020 1E26 260F  BNE GO
0021 1E28 B600  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 BD31  BSR SCREEN
0029 1E3D B606  LDA #$6
0030 1E3F 6FB0  ERASE CLR ,X+
0031 1E41 6FB4  CLR ,X
0032 1E43 30B81F  LEAX 31,X
0033 1E46 4A  DECA
0034 1E47 26F6  BNE ERASE
0035 1E49 B6015A  LDA #$15A
0036 1E4C B71E9F  STA HORIZ
0037 1E4F B6015B  LDA #$15B
0038 1E52 B71E90  STA VERTJ
0039 1E55 8D17  BSR SCREEN
0040 1E57 33BD0036  LEAU TABLE,PCR
0041 1E5B B606  LDA #$6
0042 1E5D B71E8E  STA COUNT
0043 1E60 ECC1  DISPLA LDL ,U++
0044 1E62 ED84  STD ,X
0045 1E64 30BB20  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 B13D  CMPA #$61
0051 1E73 2502  BLO CALC
0052 1E75 B63D  LDA #$61
0053 1E77 C660  CALC LDB #$96
0054 1E79 3D  MUL
0055 1E7A 1F02  TFR D,Y
0056 1E7C F61E8F  LDB HORIZ
0057 1E7F 54  LSRB
0058 1E80 C11E  CMFB #$30
0059 1E82 2502  BLO DO
0060 1E84 C61E  LDB #$30
0061 1E86 4F  DO CLRA
0062 1E87 30AB  LEAX D,Y
0063 1E89 30B90600  LEAX VIDBEG,X
0064 1E8D 39  RTS
0065 1E8E  COUNT RMB 1
0066 1E8F  HORIZ RMB 1
0067 1E90  VERTJ RMB 1
0068 1E91 00000000  TABLE FCB 0,0,$A,$AO
0069 1E95 AAAAAAA  FCB $AA,$AA,$AA,$AA
0070 1E99 08200000  FCB 8,$20,0,0
0071 1E9D  END

155
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.

![Diagram of saucer flying around 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 pro-
ceeding 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 Re-Quest (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:

\[
\text{ANDCC} \ #\%11101111
\]

\[
\text{AND} \quad \text{Condition Code Register} \quad \text{with binary value that bit 4 will immediately be cleared follows}
\]

\[
\text{this zero assures that bit 4 will}
\]

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

\[
\text{ORCC} \ #\%00010000
\]

\[
\text{OR} \quad \text{Condition Code Register} \quad \text{with immediate binary value this one assures that bit 4 is set}
\]

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 $FFFF8 and $FFFF9. 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:

```assembly
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

```assembly
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

```assembly
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:

```
INRPT  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 interrupt 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
```
INTERRUPTS

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 accumulator 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 B600 G0 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 #111101111
0017 2026 A99FA000 KEVIN JSR [#A000]
0018 202A 27FA BEQ KEVIN
0019 202C 1A10 ORCC #200010000
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 B13C CMPA #$3C
0026 203C 2526 BLO SKIP
0027 203E 7C3001 INC $3001
0028 2041 B63001 LDA $3001
0029 2044 8B30 ADDA #$30
0030 2046 B13A CMPA #$3A
0031 2048 2612 BNE ON
```
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.
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    EDU $A000
0002  0600    KEYIN    EDU $A000
0003  0600    VIDBEG    EDU $600
0004  0600             ORG VIDBEG+$1800
0005  1E00  B6E0    START LDA #$E0
0006  1E02  B7FF22  STA $FF22
0007  1E05  B7FFC3  STA $FFC3
0008  1E08  B7FFC5  STA $FFC5
0009  1E0B  B7FFC7  STA $FFC7
0010  1E0E    4F          CLR A
0011  1E0F    5F          CLR B
0012  1E10    8E0600   LDX #VIDBEG
0013  1E13  ED81          STD ,X++
0014  1E15  8C1E00   CMPX #VIDBEG+$1800
0015  1E18  25F9          BLO LOOP
0016  1E1A  B6FF          PYLONS LDA #$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  B600          LDA #0
0024  1E30  B73000  STA $3000
0025  1E33  B73001  STA $3001
0026  1E36  B73002  STA $3002
0027  1E39  F71EF6  STB HORIZ
0028  1E3C  F71EF7  STB VERTJ
0029  1E3F  8E1EC1    TIME LDX #INTERRUPT
0030  1E42  BF010D  STX #$D0
0031  1E45  B635  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  B634  LDA #$34
0039  1E58  B7FF03  STA #$FF03
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.

<table>
<thead>
<tr>
<th>SHAPE</th>
<th>TABLE VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="SHAPE" /></td>
<td><strong>Binary</strong></td>
</tr>
<tr>
<td><img src="image2" alt="SHAPE" /></td>
<td>00000000 00000000</td>
</tr>
<tr>
<td><img src="image3" alt="SHAPE" /></td>
<td>00000000</td>
</tr>
<tr>
<td><img src="image4" alt="SHAPE" /></td>
<td>11111111</td>
</tr>
<tr>
<td><img src="image5" alt="SHAPE" /></td>
<td>11101010 01011111</td>
</tr>
<tr>
<td><img src="image6" alt="SHAPE" /></td>
<td>11111111</td>
</tr>
<tr>
<td><img src="image7" alt="SHAPE" /></td>
<td>00001111 11100000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SHAPE</th>
<th>TABLE VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image8" alt="SHAPE" /></td>
<td><strong>Binary</strong></td>
</tr>
<tr>
<td><img src="image9" alt="SHAPE" /></td>
<td>00000000 00000000</td>
</tr>
<tr>
<td><img src="image10" alt="SHAPE" /></td>
<td>00001010 10100000</td>
</tr>
<tr>
<td><img src="image11" alt="SHAPE" /></td>
<td>11110101 10101111</td>
</tr>
<tr>
<td><img src="image12" alt="SHAPE" /></td>
<td>11111111</td>
</tr>
<tr>
<td><img src="image13" alt="SHAPE" /></td>
<td>00000101 01010000</td>
</tr>
<tr>
<td><img src="image14" alt="SHAPE" /></td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SHAPE</th>
<th>TABLE VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image15" alt="SHAPE" /></td>
<td><strong>Binary</strong></td>
</tr>
<tr>
<td><img src="image16" alt="SHAPE" /></td>
<td>00000000 00000000</td>
</tr>
<tr>
<td><img src="image17" alt="SHAPE" /></td>
<td>00001111 11100000</td>
</tr>
<tr>
<td><img src="image18" alt="SHAPE" /></td>
<td>01010101 01010101</td>
</tr>
<tr>
<td><img src="image19" alt="SHAPE" /></td>
<td>01010101 01010101</td>
</tr>
<tr>
<td><img src="image20" alt="SHAPE" /></td>
<td>10100000 00001010</td>
</tr>
<tr>
<td><img src="image21" alt="SHAPE" /></td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

|= red | |= blue | |= yellow | |= green |

*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.
• 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.

<table>
<thead>
<tr>
<th>Bit Pair</th>
<th>Color Set 0</th>
<th>Color Set 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>red</td>
<td>orange</td>
</tr>
<tr>
<td>10</td>
<td>blue</td>
<td>magenta</td>
</tr>
<tr>
<td>01</td>
<td>yellow</td>
<td>cyan</td>
</tr>
<tr>
<td>00</td>
<td>green</td>
<td>buff</td>
</tr>
</tbody>
</table>

• One byte of data provides four colored elements.
binary   hex

11 10 01 11 = E7 gives

red  blue  yellow  red

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

<table>
<thead>
<tr>
<th>Hex Data</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>Location</td>
</tr>
<tr>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>0</td>
<td>601</td>
</tr>
<tr>
<td>A</td>
<td>620</td>
</tr>
<tr>
<td>A0</td>
<td>621</td>
</tr>
<tr>
<td>AA</td>
<td>640</td>
</tr>
<tr>
<td>AA</td>
<td>641</td>
</tr>
<tr>
<td>AA</td>
<td>640</td>
</tr>
<tr>
<td>AA</td>
<td>660</td>
</tr>
<tr>
<td>AA</td>
<td>661</td>
</tr>
<tr>
<td>A</td>
<td>680</td>
</tr>
<tr>
<td>A0</td>
<td>681</td>
</tr>
<tr>
<td>0</td>
<td>6A0</td>
</tr>
</tbody>
</table>
| 0   | 6A1     | 2 columns

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

JSR JOYST
LDA $15A
STA HORIZ
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 interrupt service routine.
  STX  $10D \} enables interrupt
  LDA  #$35 \} clears interrupt status bit
  STA  $FF03 \} disables interrupt
  ANDCC  #$11101111
  LDA  $FF02
  ORCC  #$00010000

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

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

   JSR JOYST
   LDA $15A
   STA HORIZ
   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 # 00001000

Answers to Odd-Numbered Exercises of Chapter Test

3. JSR JOYST reads the joystick
   LDA $15A gets horizontal joystick data
   STA HORIZ stores the horizontal data
   LDA $15B gets vertical joystick data
   STA VERTJ stores the vertical data

5. a. upper, right
    b. 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:

\[
\text{PSHS X}
\]

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.

![Stack Diagram](image)

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 CHAR
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.

\[ \text{CMPA #’#} \]

- compare the value in A
- immediate
- addressing
- mode
- compare the value
- with the string
- that follows
- compare it to
- the # sign

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:

\[ \text{PULS X} \]

- pull value
- off the stack
- place the value
- in the X register

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 BA1B1  BEGIN  JSR POLCAT GET A KEY
 0009 070B 8123  CMPA #0 CONTROL CODE?
 0010 070D 270E  BNE 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 BA30A  JSR PRINT
 0016 071B 20EB  BRA BEGIN
 0017 071D 8E0601  PRINT LDX #TEXT GET BUFFER
 0018 0720 F60600  LDB PRESS THIS MANY
 0019 0723 A690  LOOP LDA ,X+ PUT CHAR IN A
 0020 0725 3414  PSHS B,X SAVE BOTH
 0021 0727 BA2BF  JSR $A2BF PRINT "$2
 0022 072A 3514  PULS B,X RESTORE B & X
 0023 072C 5A  DECB
 0024 072D 28F4  BNE LOOP
 0025 072F 8600  LDA #0 GET CR
 0026 0731 BA2BF  JSR $A2BF PRINT #2
 0027 0734 3F  SWI GO BACK TO SDS80C
 0028 0735  END
```

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**

If 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 probably 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 be able to fill about half of the video screen before sending text to the printer.

Therefore, type only about 7 or 8 lines. Then send the results to the printer. The program will return to pabug 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          | —added          |
| INC PRESS| 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	changed
JSR PRINIT
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 #\n
0014 0716 260C	BNE ON
0015 0718 301F	LEAX -1,X
0016 071A 7A0600	DEC PRESS
0017 071B 3410	PSHS X
0018 071D BDA30A	BEGIN JSR PRINIT
0019 0720 220E	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 073B 3414	PSHS B,X
0029 073D BDA2BF	JSR $A2BF
0030 073F 3514	PULS B,X
0031 0740 5A	DECB
0032 0742 26F4	BNE LOOP
0033 0744 860D	LDA #$D
0034 0746 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 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

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0600</td>
<td>PRESS</td>
<td>0F</td>
</tr>
<tr>
<td>0601</td>
<td>TEXT</td>
<td>53</td>
</tr>
</tbody>
</table>
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:

SOMETIMES ERRORS ARE NOT
DISCOVERED IMMEDIATELY.

To correct, backspace until you see:

SOMETIMES ERROR

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:
ENLARGING THE TEXT BUFFER

0001 0600  POLCAT EQU $A1B1
0002 0600  PRINT 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 20EB  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 B60D  LDA ##D
0032 083F BDA2BF  JSR $A2BF
0033 0842 3F  SWI
0034 0843  END

BEGIN 080A LOOP 0830 ON 0822 POLCAT A1B1
PRINT 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 mode (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        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 \( \frac{128}{4} = 32 \) characters on each line. We could squeeze in \( \frac{192}{9} \) 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](image-url)
Table 8-1. Data Bytes for Text Characters

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>FF</td>
<td>FC</td>
<td>FF</td>
<td>FC</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FC</td>
<td>03</td>
<td>C3</td>
<td>C0</td>
<td>C3</td>
<td>C3</td>
<td>3C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>C3</td>
<td>C0</td>
<td>C3</td>
<td>C0</td>
<td>C0</td>
<td>C0</td>
<td>C3</td>
<td>30</td>
<td>03</td>
<td>CC</td>
<td>C0</td>
<td>FF</td>
<td>F3</td>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>C3</td>
<td>C0</td>
<td>C3</td>
<td>C0</td>
<td>C0</td>
<td>C0</td>
<td>C3</td>
<td>30</td>
<td>03</td>
<td>CC</td>
<td>C0</td>
<td>FF</td>
<td>F3</td>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>FC</td>
<td>FC</td>
<td>FC</td>
<td>FC</td>
<td>FC</td>
<td>FC</td>
<td>FF</td>
<td>03</td>
<td>03</td>
<td>FO</td>
<td>C0</td>
<td>C3</td>
<td>CF</td>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>C3</td>
<td>C0</td>
<td>C3</td>
<td>C0</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>30</td>
<td>03</td>
<td>CC</td>
<td>C0</td>
<td>C3</td>
<td>CF</td>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>C3</td>
<td>C0</td>
<td>C3</td>
<td>C0</td>
<td>C0</td>
<td>C0</td>
<td>C3</td>
<td>30</td>
<td>03</td>
<td>CC</td>
<td>C0</td>
<td>C3</td>
<td>CF</td>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>FC</td>
<td>FC</td>
<td>FC</td>
<td>FF</td>
<td>FC</td>
<td>FF</td>
<td>C3</td>
<td>FC</td>
<td>FF</td>
<td>C3</td>
<td>TC</td>
<td>FF</td>
<td>C3</td>
<td>3C</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>FC</td>
<td>3C</td>
<td>FC</td>
<td>3F</td>
<td>FC</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>CC</td>
<td>FF</td>
<td>3C</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>30</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>CC</td>
<td>03</td>
<td>C3</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>C0</td>
<td>30</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>3C</td>
<td>CC</td>
<td>0C</td>
<td>0C</td>
<td>03</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>FC</td>
<td>C3</td>
<td>FC</td>
<td>3C</td>
<td>30</td>
<td>C3</td>
<td>C3</td>
<td>C3</td>
<td>3C</td>
<td>30</td>
<td>3C</td>
<td>3C</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>C0</td>
<td>CF</td>
<td>CC</td>
<td>03</td>
<td>30</td>
<td>C3</td>
<td>C3</td>
<td>FF</td>
<td>C3</td>
<td>30</td>
<td>3C</td>
<td>3C</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>C0</td>
<td>CC</td>
<td>C3</td>
<td>C3</td>
<td>30</td>
<td>C3</td>
<td>3C</td>
<td>FF</td>
<td>C3</td>
<td>30</td>
<td>C0</td>
<td>00</td>
<td>3C</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>C0</td>
<td>3F</td>
<td>C3</td>
<td>FC</td>
<td>30</td>
<td>FF</td>
<td>3C</td>
<td>C3</td>
<td>C3</td>
<td>30</td>
<td>FF</td>
<td>30</td>
<td>0C</td>
<td>03</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>30</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

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**

```plaintext
ORG $1E00
LDA #$E8
STA $FF22  // Select 6C, color set 1
STA $FFC3
STA $FFC5
```
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,$FF,0
FCB 0,$C3,$C3,$3C,$3C X letters
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
ORG $1E00
LDA #$E8
STA $FF22
STA $FFC3
STA $FFC5
CLRA
CLRB
LDX #$400
STD X++
CMPX #$1C00
BLO LOOP1
LDX #$1268
LDB #4
LDA $9
STA $1F00
LDY #TABLE
BSR LET
LEAX $80, X
LDA #9
STA $1F00
DECB
BNE DRAW
JSR $A1B1
CMPA #'#
BNE KEY
LDA #0
STA KEY
STA B7F00
STA $1F00
DECB
BNE DRAW
LET LDA ,Y+
STA ,X
LEAX $20, X
DEC $1F00
BNE LET
RTS
TABLE FCB 0, FC, 30, 30, 30
FCB $30, $30, $30, $0
FCB 0, FF, C0, C0, FC
FCB C0, C0, FF, 0
FCB 0, C3, C3, C3, 3C
FCB C3, C3, C3, C3, 0
FCB 0, FC, 30, 30, 30
FCB $30, $30, $30, $0
END

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

![Diagram of 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.

![Diagram of screen resolution]
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.

<table>
<thead>
<tr>
<th></th>
<th>32 characters/row</th>
<th>21 rows</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4)128</td>
<td>9)192</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

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**

```
Table

9 bytes for A
9 bytes for B
9 bytes for C
etc.

1. Find B

2. Draw on screen where desired
```

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>41</td>
<td>K</td>
<td>4B</td>
<td>U</td>
<td>55</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
<td>L</td>
<td>4C</td>
<td>V</td>
<td>56</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
<td>M</td>
<td>4D</td>
<td>W</td>
<td>57</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
<td>N</td>
<td>4E</td>
<td>X</td>
<td>58</td>
</tr>
<tr>
<td>E</td>
<td>45</td>
<td>O</td>
<td>4F</td>
<td>Y</td>
<td>59</td>
</tr>
<tr>
<td>F</td>
<td>46</td>
<td>P</td>
<td>50</td>
<td>Z</td>
<td>5A</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
<td>Q</td>
<td>51</td>
<td>?</td>
<td>3F</td>
</tr>
<tr>
<td>H</td>
<td>48</td>
<td>R</td>
<td>52</td>
<td>,</td>
<td>2C</td>
</tr>
<tr>
<td>I</td>
<td>49</td>
<td>S</td>
<td>53</td>
<td>.</td>
<td>2E</td>
</tr>
<tr>
<td>J</td>
<td>4A</td>
<td>T</td>
<td>54</td>
<td>space</td>
<td>20</td>
</tr>
</tbody>
</table>

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

Start of table → Table + 9 → Table + 12 → Table + 1B → alphabet starts after 9 bytes
   | 9 bytes for A
   | 9 bytes for B
   | 9 bytes for C
etc.
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](image)

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

<table>
<thead>
<tr>
<th>Row</th>
<th>Column 01</th>
<th>Column 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>600</td>
<td>71F</td>
</tr>
<tr>
<td>02</td>
<td>720</td>
<td>83F</td>
</tr>
<tr>
<td>03</td>
<td>840</td>
<td>95F</td>
</tr>
<tr>
<td>04</td>
<td>960</td>
<td>A7F</td>
</tr>
<tr>
<td>05</td>
<td>A80</td>
<td>B9F</td>
</tr>
<tr>
<td>06</td>
<td>BA0</td>
<td>CBF</td>
</tr>
<tr>
<td>07</td>
<td>CC0</td>
<td>DDF</td>
</tr>
<tr>
<td>08</td>
<td>DE0</td>
<td>EFF</td>
</tr>
<tr>
<td>09</td>
<td>F00</td>
<td>101F</td>
</tr>
<tr>
<td>0A</td>
<td>1020</td>
<td>113F</td>
</tr>
<tr>
<td>0B</td>
<td>1140</td>
<td>125F</td>
</tr>
<tr>
<td>0C</td>
<td>1260</td>
<td>137F</td>
</tr>
<tr>
<td>0D</td>
<td>1380</td>
<td>149F</td>
</tr>
<tr>
<td>0E</td>
<td>14A0</td>
<td>15BF</td>
</tr>
<tr>
<td>0F</td>
<td>15C0</td>
<td>16DF</td>
</tr>
<tr>
<td>10</td>
<td>16E0</td>
<td>17FF</td>
</tr>
<tr>
<td>11</td>
<td>1800</td>
<td>191F</td>
</tr>
<tr>
<td>12</td>
<td>1920</td>
<td>1A3F</td>
</tr>
<tr>
<td>13</td>
<td>1A40</td>
<td>1B5F</td>
</tr>
<tr>
<td>14</td>
<td>1B60</td>
<td>1C7F</td>
</tr>
<tr>
<td>15</td>
<td>1C80</td>
<td>1D9F</td>
</tr>
</tbody>
</table>

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:

```
ORANGE TEXT
ON BUFF
```
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.
<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Memory Access</th>
<th>Value</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0039</td>
<td>1E51</td>
<td>E6A0</td>
<td>ADDD</td>
<td>#TABLE1</td>
</tr>
<tr>
<td>0040</td>
<td>1E53</td>
<td>C31E8A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0041</td>
<td>1E56</td>
<td>1F03</td>
<td>TFR</td>
<td>D,U</td>
</tr>
<tr>
<td>0042</td>
<td>1E5B</td>
<td>B609</td>
<td>LDA</td>
<td>#7</td>
</tr>
<tr>
<td>0043</td>
<td>1E5A</td>
<td>B71E89</td>
<td>STA</td>
<td>LINE</td>
</tr>
<tr>
<td>0044</td>
<td>1E5D</td>
<td>A6C0</td>
<td>LOOP5</td>
<td>LDA U+</td>
</tr>
<tr>
<td>0045</td>
<td>1E5F</td>
<td>A7B4</td>
<td>STA</td>
<td>X</td>
</tr>
<tr>
<td>0046</td>
<td>1E61</td>
<td>308820</td>
<td>LEAX</td>
<td>$20,X</td>
</tr>
<tr>
<td>0047</td>
<td>1E64</td>
<td>7A1E89</td>
<td>DEC</td>
<td>LINE</td>
</tr>
<tr>
<td>0048</td>
<td>1E67</td>
<td>26F4</td>
<td>BNE</td>
<td>LOOP5</td>
</tr>
<tr>
<td>0049</td>
<td>1E69</td>
<td>3089FEE2</td>
<td>LEAX</td>
<td>-$11E,X</td>
</tr>
<tr>
<td>0050</td>
<td>1E6D</td>
<td>7A1E8B</td>
<td>DEC</td>
<td>COUNT</td>
</tr>
<tr>
<td>0051</td>
<td>1E70</td>
<td>26DE</td>
<td>BNE</td>
<td>LOOP4</td>
</tr>
<tr>
<td>0052</td>
<td>1E72</td>
<td>BDA1B1</td>
<td>LOOP6</td>
<td>JSR POLCAT</td>
</tr>
<tr>
<td>0053</td>
<td>1E75</td>
<td>B123</td>
<td>CMPA</td>
<td>##</td>
</tr>
<tr>
<td>0054</td>
<td>1E77</td>
<td>26F9</td>
<td>BNE</td>
<td>LOOP6</td>
</tr>
<tr>
<td>0055</td>
<td>1E79</td>
<td>B600</td>
<td>LDA</td>
<td>#0</td>
</tr>
<tr>
<td>0056</td>
<td>1E7B</td>
<td>B7FF22</td>
<td>STA</td>
<td>$FF22</td>
</tr>
<tr>
<td>0057</td>
<td>1E7E</td>
<td>B7FFC2</td>
<td>STA</td>
<td>$FFC2</td>
</tr>
<tr>
<td>0058</td>
<td>1E81</td>
<td>B7FFC4</td>
<td>STA</td>
<td>$FFC4</td>
</tr>
<tr>
<td>0059</td>
<td>1E84</td>
<td>B7FFC6</td>
<td>STA</td>
<td>$FFC6</td>
</tr>
<tr>
<td>0060</td>
<td>1E87</td>
<td>3F</td>
<td>SWI</td>
<td></td>
</tr>
<tr>
<td>0061</td>
<td>1E88</td>
<td>COUNT</td>
<td>RMB</td>
<td>1</td>
</tr>
<tr>
<td>0062</td>
<td>1E89</td>
<td>LINE</td>
<td>RMB</td>
<td>1</td>
</tr>
<tr>
<td>0063</td>
<td>1E8A</td>
<td>000000000000</td>
<td>TABLE1</td>
<td>FCB 0,0,0,0,0</td>
</tr>
<tr>
<td>0064</td>
<td>1E8F</td>
<td>000000000</td>
<td>FCB</td>
<td>0,0,0,0</td>
</tr>
<tr>
<td>0065</td>
<td>1E93</td>
<td>00FFC3C3FF</td>
<td>FCB</td>
<td>$FF,$C3,$C3,$FF</td>
</tr>
<tr>
<td>0066</td>
<td>1E9B</td>
<td>C3C3C300</td>
<td>FCB</td>
<td>$C3,$C3,$C3,0</td>
</tr>
<tr>
<td>0067</td>
<td>1E9C</td>
<td>00FFC3C3FC</td>
<td>FCB</td>
<td>$FF,$C3,$C3,$FC</td>
</tr>
<tr>
<td>0068</td>
<td>1EA1</td>
<td>C3C3FC00</td>
<td>FCB</td>
<td>$C3,$C3,$FC,0</td>
</tr>
<tr>
<td>0069</td>
<td>1EA5</td>
<td>00FFC0C000</td>
<td>FCB</td>
<td>$FF,$C0,$C0,$C0</td>
</tr>
<tr>
<td>0070</td>
<td>1EAA</td>
<td>C0C0FF00</td>
<td>FCB</td>
<td>$C0,$C0,$FF,0</td>
</tr>
<tr>
<td>0071</td>
<td>1EAE</td>
<td>00FFC3C3C3</td>
<td>FCB</td>
<td>$FF,$C3,$C3,$C3</td>
</tr>
<tr>
<td>0072</td>
<td>1EB3</td>
<td>C3C3FC00</td>
<td>FCB</td>
<td>$C3,$C3,$FC,0</td>
</tr>
<tr>
<td>0073</td>
<td>1EB7</td>
<td>00FFC0C0FC</td>
<td>FCB</td>
<td>$FF,$C0,$C0,$FC</td>
</tr>
<tr>
<td>0074</td>
<td>1EBC</td>
<td>C0C0FF00</td>
<td>FCB</td>
<td>$C0,$C0,$FF,0</td>
</tr>
<tr>
<td>0075</td>
<td>1EC0</td>
<td>00FFC0C0FC</td>
<td>FCB</td>
<td>$FF,$C0,$C0,$FC</td>
</tr>
<tr>
<td>0076</td>
<td>1EC5</td>
<td>C0C0C000</td>
<td>FCB</td>
<td>$C0,$C0,$C0,0</td>
</tr>
<tr>
<td>0077</td>
<td>1EC9</td>
<td>00FFC0C0CF</td>
<td>FCB</td>
<td>$FF,$C0,$C0,$CF</td>
</tr>
<tr>
<td>0078</td>
<td>1ECE</td>
<td>C3C3FF00</td>
<td>FCB</td>
<td>$C3,$C3,$FF,0</td>
</tr>
<tr>
<td>0079</td>
<td>1ED2</td>
<td>00C3C3C3FF</td>
<td>FCB</td>
<td>$C3,$C3,$C3,$FF</td>
</tr>
<tr>
<td>0080</td>
<td>1ED7</td>
<td>C3C3C300</td>
<td>FCB</td>
<td>$C3,$C3,$C3,0</td>
</tr>
<tr>
<td>0081</td>
<td>1EDB</td>
<td>00FC300C30</td>
<td>FCB</td>
<td>$FC,$30,$30,$30</td>
</tr>
<tr>
<td>0082</td>
<td>1EEO</td>
<td>3050FC00</td>
<td>FCB</td>
<td>$30,$30,$FC,0</td>
</tr>
<tr>
<td>0083</td>
<td>1EE4</td>
<td>0003050303</td>
<td>FCB</td>
<td>0,3,3,3</td>
</tr>
<tr>
<td>0084</td>
<td>1EE9</td>
<td>03C3FF00</td>
<td>FCB</td>
<td>$3,$C3,$FF,0</td>
</tr>
</tbody>
</table>
COUNT 1E88 LINE 1E89 LOOP1 1E13 LOOP2 1E26
LOOP0 1E33 LOOP4 1E50 LOOP5 1E50 LOOP6 1E72
POLCAT A181 SCREE1 1024 SCREE2 1268 START 1E00
TABLE1 1E8A TABLE2 1F98 VIDBEG 0600

Figure 9-2. Program 23—Text by Graphics

205
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:

    OFFSET = (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
    BEQ CONVT
    AND #1
    LDB #$10
    MUL
    STB ROW
get ten’s digit
of the character
convert ASCII to 1 or 0
multiply by 16 (10 hex)
save the ten’s digit

CONVU
    JSR POLCAT
    BEQ CONVU
    CMPA #$41
    BLO ON
    CMPA #$46
    BGT CONVU
    SUBA #7
get one’s digit
if in range of 0-9
go to ON
if greater than F, bad input
go back for another input
if A-F, subtract 7

ON
    SUBA #$30
    ADDA ROW
change ASCII to hex 0-F
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

\[
\text{LOCATION} = (\text{ROW}-1) \times 120 + (\text{COL}-1) \times 2 + \text{VIDBEG}
\]

\[
\text{VIDBEG} = \$600
\]

120\times (\text{ROW}-1)
gives row offset

2\times (\text{COL}-1)
gives column offset

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
MESSAGE TO SELECT INPUTS

LDX #VIDBEG + $124

$600

The listing of Program 24, Keyboard Graphic Text, is shown in Figure 9-3.

```assembly
0091 09600  VIDBEG  EQU  $600
0092 09600  POLCAT  EQU  $A1B1
0093 09600  ORG  VIDBEG+$1800
0094 1E00 B6EB  START  LDA  #$E8
0095 1E02 B7FF22  STA  #$FF22
0096 1E05 B7FFC3  STA  #$FFC3
0097 1E08 B7FFC5  STA  #$FFC5
0098 1E0B B7FFC7  STA  #$FFC7
0099 1E0E 4F  CLRA
0100 1E0F 5F  CLRB
0101 1E10 8E0600  LOOP1  STD  ,x++
0102 1E13 8C1E00  CMPIX  #VIDBEG+$1800
0103 1E18 25FF9  BLO  LOOP1
0104 1E1A 8E0724  LDX  #VIDBEG+$124
0105 1E1D 108E2077  LDY  #TABLE2
0106 1E21 8604  LDA  #$4
0107 1E23 B71F67  STA  COUNT
0108 1E26 1700FB  LBSR  DISPLA
0109 1E29 17011B  LBSR  CONVT
0110 1E2C B71F61  STA  ROW
0111 1E2F 17011F  LBSR  CONVU
0112 1E32 80030  SUBA  #$30
0113 1E34 BB1F61  ADDA  ROW
0114 1E37 8001  SUBA  #$1
0115 1E39 B71F61  STA  ROW
0116 1E3C 8E0724  LDX  #VIDBEG+$124
0117 1E3F 8604  LDA  #$4
0118 1E41 B71F67  STA  COUNT
0119 1E44 1700DA  LBSR  DISPLA
0120 1E47 1700FA  LBSR  CONVT
0121 1E4A B71F62  STA  COL
0122 1E4D 170101  LBSR  CONVU
0123 1E50 80030  SUBA  #$30
0124 1E52 BB1F62  ADDA  COL
0125 1E55 8001  SUBA  #$1
0126 1E57 B71F62  STA  COL
0127 1E5A 8E0724  LDX  #VIDBEG+$124
0128 1E5B 8605  LDA  #$5
0129 1E5F B71F67  STA  COUNT
0130 1E62 1700BC  LBSR  DISPLA
0131 1E65 1700DC  LBSR  CONVT
0132 1E68 B71F63  STA  CHRS
0133 1E6B 1700E3  LBSR  CONVU
```
0098 1EE5 A7B4 STA ,X
0099 1EE7 308B20 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 B600 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 C00105 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 B609 LDA #9
0130 1F2B B71F68 STA LINE
0131 1F2E A6C0 LOOP LDA ,U+
0132 1F30 A7B4 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 B401 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 B141 CMPA ##41
0150 1F58 2506 BLO ON
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

<table>
<thead>
<tr>
<th>Input</th>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>05</td>
<td>5</td>
</tr>
<tr>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>07</td>
<td>7</td>
</tr>
<tr>
<td>08</td>
<td>8</td>
</tr>
<tr>
<td>09</td>
<td>9</td>
</tr>
<tr>
<td>0A</td>
<td>A</td>
</tr>
<tr>
<td>0B</td>
<td>B</td>
</tr>
<tr>
<td>0C</td>
<td>C</td>
</tr>
<tr>
<td>0D</td>
<td>D</td>
</tr>
<tr>
<td>0E</td>
<td>E</td>
</tr>
<tr>
<td>0F</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

After you have entered the row value, the screen will display:

\[ \text{COL?} \]

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

\[ \text{CHRS?} \]

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>05</td>
<td>5</td>
</tr>
<tr>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>07</td>
<td>7</td>
</tr>
<tr>
<td>08</td>
<td>8</td>
</tr>
<tr>
<td>09</td>
<td>9</td>
</tr>
<tr>
<td>0A</td>
<td>A</td>
</tr>
<tr>
<td>0B</td>
<td>B</td>
</tr>
<tr>
<td>0C</td>
<td>C</td>
</tr>
<tr>
<td>0D</td>
<td>D</td>
</tr>
<tr>
<td>0E</td>
<td>E</td>
</tr>
<tr>
<td>0F</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

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. **CHRSH?**
   - We typed 08

5. **screen blanks**

6. We typed the following message:
   a. **K**
      - K \(\rightarrow\) row A, column 5
   b. **E**
      - KE \(\rightarrow\) column 6
   c. **Y**
      - KEY \(\rightarrow\) column 7
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

![Image of 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.

\[
OFFSET = (ASCII \text{ CODE} - 40) \times 9
\]

The memory location where the text is to begin on the screen can also be calculated for a selected row and column by

\[
LOCATION = (ROW-1) \times 120 + (COLUMN-1) \times 2 + VIDBEG
\]

allows for
9 rows per character
32 locations per row
allows for space between letters
beginning of graphic memory

New Instructions in This Chapter

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Program Where Used and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD #$120</td>
<td>24 add $120 to the value in D</td>
</tr>
<tr>
<td>BGT KEY</td>
<td>24 used after a compare—if value is greater than the one it is compared to, a branch is made</td>
</tr>
<tr>
<td>LBSR DISPLAY</td>
<td>24 used when subroutine is more than 255 bytes away (Long Branch SubRoutine)</td>
</tr>
<tr>
<td>LDX #VIDBEG + $124</td>
<td>24 X is loaded from the location (VIDBEG + 124)</td>
</tr>
<tr>
<td>LSLB</td>
<td>24 Logical Shift Left register B—shifts all bits in B one place left</td>
</tr>
<tr>
<td>SUBA #$30</td>
<td>24 subtracts $30 from value in A</td>
</tr>
<tr>
<td>TFR D,U</td>
<td>23 transfers data from U to D—data is also left in U</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>A in orange</th>
<th>A in magenta</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>row</th>
<th>1-8</th>
<th>9-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
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

\[ X = \text{USRO}(20 \cdot 256 + 20) \]

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  VIDESEG EDU `$600
0002 0600  POLCAT EQU `$A1B1
0003 0600  ORG `$3000
0004 3C00  BDB3ED  JSR `$B3ED
0005 3C03  B73CC1  STA COL
0006 3C06  F73CC0  STB ROW
0007 3C09  4F    CLR A
0008 3C0A  1F01  TFR D,X
0009 3C0C  5F    CLR B
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    CLR A
0015 3C18  F63CC1  LDB COL
0016 3C1B  F33CC2  ADDD OFFSET
0017 3C1E  C30600  ADDD `VIDSEG
0018 3C21  1F01  TFR D,X
0019 3C23  59B1  KEY JSR POLCAT
0020 3C26  27FB  RED KEY
0021 3C28  B120  CMPA `##20
0022 3C2A  102700B8  LBEQ ZERO
0023 3C2E  B12E  CMPA `##2E
0024 3C30  102700B1  LBEQ DOT
0025 3C34  B12C  CMPA `##2C
0026 3C36  277B  BEQ COMMA
0027 3C38  B13F  CMPA `##3F
0028 3C3A  276F  BEO QUEST
0029 3C3C  B140  CMPA `##40
0030 3C3E  2531  BLO NUM
0031 3C40  27E1  BEO KEY
0032 3C42  B15A  CMPA `##5A
0033 3C44  224A  BHI LOWC
0034 3C46  B040  SUBA `##40
0035 3C48  B73CBE  STA HOLD
0036 3C4B  4B    LSLA
0037 3C4C  4B    LSLA
0038 3C4D  4B    LSLA
0039 3C4E  B3CBE  ADDA HOLD
0040 3C51  1F09  TFR A,B
0041 3C53  4F    CLR A
0042 3C54  C33CC4  UP ADDD `TABLE1
0043 3C57  1F03  GO TFR D,U
0044 3C59  B609  LDA `9
0045 3C5B  B73CBE  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 30B9FEE1 LEAX -$11F,X
0052 3C6E 20B3   BRA KEY
0053 3C70 39    BACK RTS
0054 3C71 810D   NUM CMPA #40D
0055 3C73 17FB   BEQ BACK
0056 3C75 8130   CMPA #330
0057 3C77 25AA   BLD KEY
0058 3C79 8139   CMPA #339
0059 3C7B 22A6   BHI KEY
0060 3C7D 8030   SUBA #30
0061 3C7F B73CBE STA HOLD
0062 3CB2 48    LSLA
0063 3CB3 48    LSLA
0064 3CB4 48    LSLA
0065 3CB5 BB3CBE ADDA HOLD
0066 3CB8 1FB9   TFR A,B
0067 3CBA 4F    CLR A
0068 3CBB C33EBC ADDD #TABLE3
0069 3CBE 20C7   BRA GO
0070 3C90 8161   LOWC CMPA #61
0071 3C92 258F   BLD KEY
0072 3C94 817A   CMPA #7A
0073 3C96 22BB   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 1FB9   TFR A,B
0081 3CA5 4F    CLR A
0082 3CA6 C33DD2 ADDD #TABLE2
0083 3CA9 20AC   BRA GO
0084 3CAB C6F3   QUEST LDB #F3
0085 3CAD 4F    CLR A
0086 3CAE 20A4   BRA UP
0087 3CB0 C6FC   COMMA LDB #FC
0088 3CB2 4F    CLR A
0089 3CB3 209F   BRA UP
0090 3CB5 CC0105 DOT LDD #105
0091 3CB8 209A   BRA UP
0092 3CBA 4F     ZERO CLR A
0093 3CBB 5F     CLR B
0094 3CBC 2096  BRA UP
0095 3CBE     HOLD RMB 1
<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
<th>Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0143</td>
<td>3DB6</td>
<td>FCB</td>
<td>$22, $22, $3E, 0</td>
</tr>
<tr>
<td>0144</td>
<td>3DBA</td>
<td>FCB</td>
<td>$22, $22, $22, $22</td>
</tr>
<tr>
<td>0145</td>
<td>3DBF</td>
<td>FCB</td>
<td>$22, $14, $B, 0</td>
</tr>
<tr>
<td>0146</td>
<td>3D93</td>
<td>FCB</td>
<td>$22, $22, $22, $22</td>
</tr>
<tr>
<td>0147</td>
<td>3D9B</td>
<td>FCB</td>
<td>$2A, $36, $22, 0</td>
</tr>
<tr>
<td>0148</td>
<td>3D9C</td>
<td>FCB</td>
<td>$22, $22, $14, 8</td>
</tr>
<tr>
<td>0149</td>
<td>3DA1</td>
<td>FCB</td>
<td>$14, $22, $22, 0</td>
</tr>
<tr>
<td>0150</td>
<td>3DA5</td>
<td>FCB</td>
<td>$22, $22, $22, $22</td>
</tr>
<tr>
<td>0151</td>
<td>3DAA</td>
<td>FCB</td>
<td>$8, $8, 0</td>
</tr>
<tr>
<td>0152</td>
<td>3DAE</td>
<td>FCB</td>
<td>$3E, 2, 4, 8</td>
</tr>
<tr>
<td>0153</td>
<td>3DB3</td>
<td>FCB</td>
<td>$10, $20, $3E, 0</td>
</tr>
<tr>
<td>0154</td>
<td>3DB7</td>
<td>FCB</td>
<td>$3E, 2, 2, $E</td>
</tr>
<tr>
<td>0155</td>
<td>3DBC</td>
<td>FCB</td>
<td>8, 0, 8</td>
</tr>
<tr>
<td>0156</td>
<td>3DC0</td>
<td>FCB</td>
<td>0, 0, 0, 0</td>
</tr>
<tr>
<td>0157</td>
<td>3DC5</td>
<td>FCB</td>
<td>$C, 4, 4, 8</td>
</tr>
<tr>
<td>0158</td>
<td>3DC9</td>
<td>FCB</td>
<td>0, 0, 0, 0</td>
</tr>
<tr>
<td>0159</td>
<td>3DCE</td>
<td>FCB</td>
<td>$C, $E, 0, 0</td>
</tr>
<tr>
<td>0160</td>
<td>3DD2</td>
<td>TABLE2</td>
<td>0, 0, 0, $1E, 2</td>
</tr>
<tr>
<td>0161</td>
<td>3DD7</td>
<td>FCB</td>
<td>$E, $12, $E, 0</td>
</tr>
<tr>
<td>0162</td>
<td>3DBB</td>
<td>FCB</td>
<td>0, 0, $20, $20, $3C</td>
</tr>
<tr>
<td>0163</td>
<td>3DE0</td>
<td>FCB</td>
<td>$22, $22, $3C, 0</td>
</tr>
<tr>
<td>0164</td>
<td>3DE4</td>
<td>FCB</td>
<td>0, 0, 0, $1E, $20</td>
</tr>
<tr>
<td>0165</td>
<td>3DE9</td>
<td>FCB</td>
<td>$20, $20, $1E, 0</td>
</tr>
<tr>
<td>0166</td>
<td>3DED</td>
<td>FCB</td>
<td>0, 0, 2, 2, $1E</td>
</tr>
<tr>
<td>0167</td>
<td>3DF2</td>
<td>FCB</td>
<td>$22, $22, $1E, 0</td>
</tr>
<tr>
<td>0168</td>
<td>3DF6</td>
<td>FCB</td>
<td>0, 0, 0, $1C, $22</td>
</tr>
<tr>
<td>0169</td>
<td>3DFB</td>
<td>FCB</td>
<td>$3C, $20, $1E, 0</td>
</tr>
<tr>
<td>0170</td>
<td>3DFD</td>
<td>FCB</td>
<td>0, 0, $E, 8, $1C</td>
</tr>
<tr>
<td>0171</td>
<td>3E04</td>
<td>FCB</td>
<td>8, 8, 0</td>
</tr>
<tr>
<td>0172</td>
<td>3E08</td>
<td>FCB</td>
<td>0, 0, 0, $1C, $22</td>
</tr>
<tr>
<td>0173</td>
<td>3E0D</td>
<td>FCB</td>
<td>$22, $1E, 2, $C</td>
</tr>
<tr>
<td>0174</td>
<td>3E11</td>
<td>FCB</td>
<td>0, 0, $20, $20, $3C</td>
</tr>
<tr>
<td>0175</td>
<td>3E16</td>
<td>FCB</td>
<td>$22, $22, $22, 0</td>
</tr>
<tr>
<td>0176</td>
<td>3E1A</td>
<td>FCB</td>
<td>0, 0, $20, 0, $20</td>
</tr>
<tr>
<td>0177</td>
<td>3E1F</td>
<td>FCB</td>
<td>$20, $20, $1C, 0</td>
</tr>
<tr>
<td>0178</td>
<td>3E23</td>
<td>FCB</td>
<td>0, 0, 2, 0, 2</td>
</tr>
<tr>
<td>0179</td>
<td>3E2B</td>
<td>FCB</td>
<td>2, 2, $22, $1C</td>
</tr>
<tr>
<td>0180</td>
<td>3E2C</td>
<td>FCB</td>
<td>0, 0, $20, $22, $24</td>
</tr>
<tr>
<td>0181</td>
<td>3E31</td>
<td>FCB</td>
<td>$3B, $24, $22, 0</td>
</tr>
<tr>
<td>0182</td>
<td>3E35</td>
<td>FCB</td>
<td>0, 0, $20, $20, $20</td>
</tr>
<tr>
<td>0183</td>
<td>3E3A</td>
<td>FCB</td>
<td>$20, $20, $20, 0</td>
</tr>
<tr>
<td>0184</td>
<td>3E3E</td>
<td>FCB</td>
<td>0, 0, 0, $34, $2A</td>
</tr>
<tr>
<td>0185</td>
<td>3E43</td>
<td>FCB</td>
<td>$2A, $2A, $2A, 0</td>
</tr>
<tr>
<td>0186</td>
<td>3E47</td>
<td>FCB</td>
<td>0, 0, 0, $3C, $22</td>
</tr>
<tr>
<td>0187</td>
<td>3E4C</td>
<td>FCB</td>
<td>$22, $22, $22, 0</td>
</tr>
<tr>
<td>0188</td>
<td>3E50</td>
<td>FCB</td>
<td>0, 0, 0, $1C, $22</td>
</tr>
<tr>
<td>0189</td>
<td>3E55</td>
<td>FCB</td>
<td>$22, $22, $1C, 0</td>
</tr>
<tr>
<td>0190</td>
<td>3E59</td>
<td>FCB</td>
<td>0, 0, 0, $3C, $22</td>
</tr>
<tr>
<td>0191</td>
<td>3E5E</td>
<td>FCB</td>
<td>$22, $3C, $20, $20</td>
</tr>
</tbody>
</table>
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 000000081C   FCB 0,0,8,8,$1C
0199 3E82 08008000   FCB 8,8,6,0
0200 3E86 0000002222   FCB 0,0,0,$22,$22
0201 3E8B 222221E00   FCB $22,$22,$1E,0
0202 3E8F 0000002222   FCB 0,0,0,$22,$22
0203 3E94 22140800   FCB $22,$14,8,0
0204 3E98 0000002222A   FCB 0,0,0,$22,$2A
0205 3E9D 2A2A3600   FCB $2A,$2A,$A3,0
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 002028B3E   FCB 0,$20,$2B,$2B,$3E
0221 3EE5 08080800   FCB 8,8,8,0
0222 3EE9 003E20203C   FCB 0,$3E,$20,$20,$3C
0223 3EEF 02221C00   FCB 2,$22,$1C,0
0224 3EF2 001C22203C   FCB 0,$1C,$22,$20,$3C
0225 3EF7 22221C00   FCB $22,$22,$1C,0
0226 3EF8 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 GD 3C57 HOLD 3CB8 KEY 3C23
LINE 3CBF LOOP 3C5E LOWC 3C9A NUM 3C71
OFFSET 3C72 POLCAT A181 QUEST 3CAB ROW 3C60
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

$$\text{ABUG: } S \ 3C00 \ 3F15 \ 3C00 \text{ CHARS}$$

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 DEFUSRO=&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"E5U13D265D4E5"
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

![Diagram of a chair](image)
Now type: CHAIR

"CHAIR" appears

Now type: HOOK

"HOOK" appears

Now type: CHAIR RAISER

Title appears

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

```
PA0,7FF,3C00
```

Display

```
PA0,7FF,3C00
```

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
PV3C00,16

verify program command
beginning at original data in memory
type of EPROM used (2716 in our case; value may be any integer 0-16 for a 2716 EPROM)

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

\[ 90 \text{DEFUSRO} = 49152 \]

decimal equivalent of $C000

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 $12 \times 8$ 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:

<table>
<thead>
<tr>
<th>Color Set 0</th>
<th>Color Set 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = green</td>
<td>5 = buff</td>
</tr>
<tr>
<td>2 = yellow</td>
<td>6 = cyan</td>
</tr>
<tr>
<td>3 = blue</td>
<td>7 = magenta</td>
</tr>
<tr>
<td>4 = red</td>
<td>8 = orange</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>DATA BYTES (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXXX + 00</td>
<td>00 00 00</td>
</tr>
<tr>
<td>XXXXX + 20</td>
<td>00 00 00</td>
</tr>
<tr>
<td>XXXXX + 40</td>
<td>00 00 00</td>
</tr>
<tr>
<td>XXXXX + 60</td>
<td>00 00 00</td>
</tr>
<tr>
<td>XXXXX + 80</td>
<td>00 00 00</td>
</tr>
<tr>
<td>XXXXX + A0</td>
<td>00 00 00</td>
</tr>
<tr>
<td>XXXXX + C0</td>
<td>00 00 00</td>
</tr>
<tr>
<td>XXXXX + E0</td>
<td>00 00 00</td>
</tr>
</tbody>
</table>

**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 XXXXX 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.

\[ \text{Figure 10-5. Figure Creator (Assembler)} \]

```
ORG $3000
NAM FIGURE

SCREEN EQU $400 ;Start address of text screen memory
KEY EQU $A1B1 ;ROM Keyboard routine
START CLRA ;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
    ;to background color (green)
LOOP1 STA ,X+
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:
BEO 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    
BNE NEXT3
LDA YCUR    ;Yes, see if user tried to move
INCA        ;cursor outside of box:
CMPA #9     ;Yes, ignore command
BEO GETCOM  ;Update array
INC YCUR    ;Move cursor down
LBSR READA  ;Update old color (OLDC)
BRA SHOW    ;Re-display
NEXT3       CMPA #8     
BNE NEXT4
LDA XCUR    ;See if user tried to move
DECA        ;cursor outside of box:
BEO 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     
BNE NEXT5
LDA XCUR    ;See if user tried to move
INCA        ;cursor outside of box:
CMPA #13    ;Yes, ignore command
BEO GETCOM  ;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 #520   ;Key = space bar?
BEO SET     ;Yes, set point at cursor position
CMPA #'-'   ;Key = dash (-)?
LBEOQ RESET ;Yes, erase point at cursor
CMPA #'R'   ;Key = R ?
LBEOQ START ;Yes, restart with clear array
CMPA #'S'   ;Key = S ?
BEO CHSET   ;Yes, change color set
CMPA #'C'   ;Key = C ?
BEO CHCOL   ;Yes, change drawing color
CMPA #'T'   ;Key = T ?
BEO TEST    ;Yes, display figure in Hi-res
CMPA #'D'   ;Key = D ?
LBEOQ 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 DLDC
LDA CCOL
LBSR SWITCH
STA CCOL
LDX #DISP
; Change current color
; 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
CMFA #1
; If key is less than '1' or key
BLO CHCOL
; is greater than '8' then it is
CMFA #8
; not a valid color. Ask for new
BHI CHCOL
; color code
ANDA $00001111
; AND off ASCII bits leaving 1-8
CLR8
CMPB CSET
; Is Color Set = 0?
BNE NXTST
CMFA #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
; color, redisplay box
Lbra SHOW

NXTST
CMFA #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
INC8
; OLDC = 4 * CSET + 1
STB OLDC
Lbra GETCOM
; Get new command
TEST
LDA CSET
; *** Display the figure in mode 6C ***
LDB #8
; Set up $FF22 byte according to CSET
MUL
ADD8 #$E0
; Set up VDG for mode 6C
STB $FFC5
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
CLR8
; B=0. A=0 from result of MUL above
NEXT7
STD ,X++
; Clear screen by storing zeroes
CMPX #$1E00
; Done with whole screen?
BNE NEXT7 ;No, continue
LBSR MAKEB ;Yes, create output data list
LDX #DATA ;Point X to start of data list above
LDY #$C0CD ;Point Y to screen address to start at
BACK2 ;Put three bytes from table onto screen horizontally
LDA X,+ STA Y,+ LDA X,+ STA Y,+ LDA X,+ STA Y,+ CMPY #$0CF0 ;Done with whole figure?
BEQ NEXTB ;Yes, go onward
LEAY 29,Y ;No, point Y to next screen row
BRA BACK2 ;Draw next three data bytes
NEXTB ;Wait for a key to be pressed
JSR KEY ;Reset display offset to start of
CLRA ;text memory ($400)
STA #$FC6 ;Change VDG mode back to text mode
STA #$FC0
STA #$FC2
STA #$FC4
STA #$FF22 ;Change $FF22 back to text mode
LBRA SHOW, ;Re-display screen
TABLE ;*** Display table of data bytes ***
LBSR MAKEB ;Create output data list, clear screen
BSR CLS
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 ;Display message '"XXX+'
BSR TEXT
PSHS B ;Save B (counter) on stack
LDA #32 ;Compute offset byte to be
MUL ;displayed = Counter * 32
TFR B,A ;Display result, and a space
BSR ASCII ;Display three more spaces
LDA ,X+ ;Display byte from output data table
STA ,X+ ;Pointed to by U
STA ,X+ ;Display next table byte
BSR ASCII
LDA ,U+ ;Display next table byte
BSR ASCII
PULS B ;Restore counter
INCB ;Increment counter
CMPX #$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 '"XXX+’ again
BRA BACK3 ;Do next row
DONE4    LDX #SCREEN+416 ;Display 'Hit Enter' message
        LDY #MSG7
        BSR TEXT
BACK4    JSR KEY
        CMPA #00D ;Get key
        BNE BACK4
        LBRA REDRAW ;Is it the ENTER key?
CLS      LDX #SCREEN
        LDA #$60
        ;This routine loads space ($60) codes into the text memory,
        ;clearing the screen
        CMPX #SCREEN+$200
        BNE WIPE
        RTS
TEXT     LDA ,Y+
        ;This routine displays text on the
        ;screen. On entry, X points to the
        ;screen address to display data at,
        ;Y points to the start address of the
        ;message data, and the data ends with
        ;a zero byte. The routine also adds
        ;$40 to any codes that need it so that
        ;they are not displayed as inverted on
        ;the screen
        CMPA #0
        BEQ DONE1
        CMPA #$41
        BLO FIX
        CMPA #$7F
        BHI AOK
        CMPA #$5A
        BHI FIX
        STA ,X+
        BRA TEXT
        FIX      ADDA #$40
        BRA AOK
DONE1    RTS
SETA     BSR FINDG
        LDA OLDC
        STA ,X
        ;Set array value = OLDC
        ;Point X to Disp array address of
        ;cursor position
        RTS
READA    BSR FINDG
        LDA ,X
        ;cursor position, get array value,
        ;and store it in OLDC
        STA OLDC
        RTS
SWITCH   CMPA #5
        BLO THENA
        SUBA #4
        RTS
        ;This routine switches the color code
        ;in A-reg. from one color set to the
        ;other. If color > 4 then color =
        ;color - 4. If color < 5 then
        ;color = color + 4
        THENA ADDA #4
        RTS
ASCII    TFR A,B
        ;Save hex number to be displayed in B
        RORA
        RORA
        RORA
        RORA
        ANDA %000001111
        ANDB %000001111
        BSR CREATE
        STA ,X+
        ;Keep binary form of first digit
        ;Convert first digit to ASCII
        ;Display it on screen at X
        TFR B,A
        BSR CREATE
        STA ,X+
        ;Convert second digit to ASCII
        ;Display it on screen
        BLO NUMBER
        LDA #$60
        ;Display a space
        STA ,X+
        RTS
CREATE   CMPA #$A
        ;Return with X point to next loc.
        ;Is data 0-9?
        BNE CREATE
        RTS
        ;Yes, go convert it
ADD A $37
RTS

NUMBER ADD A $70
RTS

FINDG LDA YCUR
DECA
LDB #12
MUL
ADD D #DISP
TFR D,X
LDB XCUR
DECB
ABX
RTS

MAKEB CLR COUNT
LDX #DISP
LDY #DATA
MORE BSR BITEM
LDB #6
AGAIN ASLA
DECB
BNE AGAIN
TFR A,B
BSR BITEM
ASLA
ASLA
ASLA
ASLA
PSHS A
ADD B ,S+
BSR BITEM
ASLA
ASLA
PSHS A
ADD B ,S+
BSR BITEM
PSHS A
ADD B ,S+
STB ,Y+
INC COUNT
LDA #24
CMPA COUNT
BNE MORE
RTS

BITEM LDA ,X+
DECA
ANDA #$00000011
RTS

DRAW CLR COUNT
*** Display inside of box ***
LDX #SCREEN+138
LDY #DISP
ZOOM LDA ,Y+
DECA
LDB #16
MUL
ADD B #143
STB ,X+
INC COUNT

;No, add $37 to display A-F

;Add $70 to display 0-9

;This routine points X to the memory address where the color data corresponding to the cursor position (XCUR,YCUR) can be found.

;This routine takes bytes from the color data array (Disp), four at a time, converts the color data from 1-8 to 0-3, temporarily ignoring the color set, crams these 4 bytes into one byte which represents the data to be stored to the screen memory in graphics mode 6c, and puts this byte into the 'output data table' called DATA.

;Store byte in DATA array

;Have we done all 24 bytes?

;No, go back and do next set

;Get color data, increment X

;Color data = 0-3 or 4-7

;Strip off all but last two bits

;Now A = 0-3 for either color set

;Clear counter, point X to inside box

;Point Y to Disp array of color data

;In screen memory to display a block of that color. CODE = (COLOR - 1) * 16 + 143.

;Display block on screen

;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?
BGL 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
LDX #8
MUL
ADDD #COLORS
TFR D,Y
LDX #SCREEN+280
LBSR TEXT
LDA YCUR
DECA
LDX #32
MUL
TFR D,X
LDX XCUR
DECB
ABX
LEAX SCREEN+138,X
LDA #7* ;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"
FCC 0
TOPGR FCB 158,156,156,156,156,156
FCB 156,156,156,156,156
FCB 156,156,156,156,157,0

246
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$(8)
22 RESTORE: FOR X=0 TO 8: READ C$(X): NEXT X
24 DATA BLACK, GREEN, YELLOW, BLUE, RED, BURG, CYAN, MAGENTA, ORANGE
25 CS=0: CC=1: OC=1: CX=1: CY=1
26 FOR X=1 TO 12: F0R Y=1 TO 8: G(X,Y)=OC: NEXT Y, X
30 CLS: PRINT "FIGURE CREATION PROGRAM";
30 FOR Y=0 TO 7: PRINT 128+9+32*Y,CHR$(154);
35 PRINT 128+22+32*Y,CHR$(149);: NEXT Y
40 PRINT 150,CHR$(159): PRINT 118,CHR$(157);
45 PRINT 393,CHR$(155): PRINT 406,CHR$(151);
50 "LINES 40-60 DRAW BORDER
55 GOSUB 2000 "DISPLAY ARRAY
60 PRINT "449"; "COMMAND?"; STRING$(15," ");
65 A$=INKEY": IFA$="THEN 95
70 IFA$="THEN 200
75 IFCY<11 THEN 90
80 B(X,CY)=OC: CY=CY-1: OC=OC-8: CY+8: OC=0: G(X,CY): GOT085
85 IFA$<CHR$(10): NEXT 300 "DOWN-ARROW
90 IFCY+1<11 THEN 90
95 B(X,CY)=OC: CY=CY+1: OC=OC+8: CY-8: OC=0: G(X,CY): GOT085
100 IFA$<CHR$(8): NEXT 400 "LEFT-ARROW
105 IFCX-1<11 THEN 90
110 B(X,CX)=OC: CX=CX-1: OC=OC-8: CX+8: OC=0: G(X,CX): GOT085
115 IFA$<CHR$(9): NEXT 500 "RIGHT-ARROW
120 IFCX+1<12 THEN 90
125 B(X,CX)=OC: CX=CX+1: OC=OC+8: CX-8: OC=0: G(X,CX): GOT085
130 PRINT 1495, A$: IFA$="THEN 90: PRINT 1499: "SET POINT (SPC)
135 IFA$="THEN 45: IF C$+1<60: GOTO 90: "RESET TO BACKGROUND (-)
140 IFA$="R": THEN 22
145 IFA$="S": THEN 600
150 IFA$="C": THEN 700
155 IFA$="T": THEN 800
160 IFA$="D": THEN 900
165 GOTO 900
170 "CHANGE COLOR SET: CHANGE OC AND CS AND G(ARRAY)
175 OC=CS$+8: OC+4: CC=-CS$+CS$+CC+4: FOR X=1 TO 12
180 F0R Y=1 TO 8: G(X,Y)=CS$+6: G(X,Y)+4: NEXT Y, X: CS=-CS$+GOT085
185 FOR X=1 TO 12: F0R Y=1 TO 8: G(X,Y)=CS$+8: G(X,Y)+4: NEXT Y, X
190 "CHANGE COLOR OF SET: CHANGE CC/CHECK IF VALID
195 PRINT 1497; "TYPE NEW COLOR CODE: ";
200 C$=INKEY": IFC$="GOT0720
205 PRINT 14870, C$; IFC$="1" OR C$="B"GOT0710
210 C=VAL(C$): IFC$=0 AND C>4 THEN 710
215 IFCS=1 AND C<5 THEN 710
220 CC=C: GOT085
225 "TEST FIGURE ON REAL DISPLAY
230 PMODE 3, 1: SCREEN 1, CS: PCLS
235 GOSUB 1000 "GET DISPLAY ARRAY
240 FOR X=0 TO 7: FOR Y=1 TO 8: G(X,Y)=3072+14+X+32*Y, M(X,Y): NEXT Y, X
245 C$=INKEY": IFC$="THEN 840
250
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
EQU $A1B1
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
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
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

```
::
W
```

4-character name
press ENTER to record

253
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

```
: R

L

TEST □ press ENTER
```

To Generate an Object Program to Cassette

The assembler T option is used.

Example

```
LSMT → generate an object program to tape
       generate object code to memory
       produce a sorted symbol table
       produce a listing
```

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

```
S 3000 3115 3000 TEST
```

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

```
L 2000 TEST
```

The L command is the same as the BASIC command:

```
CLOADM "TEST",2000
```
The codes used for the keyboard and video screen for the Color Computer differ. The values shown in this appendix are used.

<table>
<thead>
<tr>
<th>ASCII CODE (Keyboard)</th>
<th>SCREEN CODE</th>
<th>CHAR.</th>
<th>ASCII CODE (Keyboard)</th>
<th>SCREEN CODE</th>
<th>CHAR.</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>60</td>
<td>space</td>
<td>35</td>
<td>75</td>
<td>5</td>
</tr>
<tr>
<td>21</td>
<td>61</td>
<td>!</td>
<td>36</td>
<td>76</td>
<td>6</td>
</tr>
<tr>
<td>22</td>
<td>62</td>
<td>&quot;</td>
<td>37</td>
<td>77</td>
<td>7</td>
</tr>
<tr>
<td>23</td>
<td>63</td>
<td>#</td>
<td>38</td>
<td>78</td>
<td>8</td>
</tr>
<tr>
<td>24</td>
<td>64</td>
<td>$</td>
<td>39</td>
<td>79</td>
<td>9</td>
</tr>
<tr>
<td>25</td>
<td>65</td>
<td>%</td>
<td>3A</td>
<td>7A</td>
<td>:</td>
</tr>
<tr>
<td>26</td>
<td>66</td>
<td>&amp;</td>
<td>3B</td>
<td>7B</td>
<td>;</td>
</tr>
<tr>
<td>27</td>
<td>67</td>
<td>'</td>
<td>3C</td>
<td>7C</td>
<td>&lt;</td>
</tr>
<tr>
<td>28</td>
<td>68</td>
<td>(</td>
<td>3D</td>
<td>7D</td>
<td>=</td>
</tr>
<tr>
<td>29</td>
<td>69</td>
<td>)</td>
<td>3E</td>
<td>7E</td>
<td>&gt;</td>
</tr>
<tr>
<td>2A</td>
<td>6A</td>
<td>*</td>
<td>3F</td>
<td>7F</td>
<td>?</td>
</tr>
<tr>
<td>2B</td>
<td>6B</td>
<td>+</td>
<td>40</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>2C</td>
<td>6C</td>
<td>,</td>
<td>41</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2D</td>
<td>6D</td>
<td>-</td>
<td>42</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>2E</td>
<td>6E</td>
<td>.</td>
<td>43</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>2F</td>
<td>6F</td>
<td>/</td>
<td>44</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>30</td>
<td>70</td>
<td>0</td>
<td>45</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>31</td>
<td>71</td>
<td>1</td>
<td>46</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>32</td>
<td>72</td>
<td>2</td>
<td>47</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>33</td>
<td>73</td>
<td>3</td>
<td>48</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>34</td>
<td>74</td>
<td>4</td>
<td>49</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>ASCII CODE (Keyboard)</td>
<td>SCREEN CODE</td>
<td>CHAR.</td>
<td>ASCII CODE (Keyboard)</td>
<td>SCREEN CODE</td>
<td>CHAR.</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------</td>
<td>-------</td>
<td>----------------------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>4A</td>
<td>4A</td>
<td>J</td>
<td>65</td>
<td>05</td>
<td>e</td>
</tr>
<tr>
<td>4B</td>
<td>4B</td>
<td>K</td>
<td>66</td>
<td>06</td>
<td>f</td>
</tr>
<tr>
<td>4C</td>
<td>4C</td>
<td>L</td>
<td>67</td>
<td>07</td>
<td>g</td>
</tr>
<tr>
<td>4D</td>
<td>4D</td>
<td>M</td>
<td>68</td>
<td>08</td>
<td>h</td>
</tr>
<tr>
<td>4E</td>
<td>4E</td>
<td>N</td>
<td>69</td>
<td>09</td>
<td>i</td>
</tr>
<tr>
<td>4F</td>
<td>4F</td>
<td>O</td>
<td>6A</td>
<td>0A</td>
<td>j</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>P</td>
<td>6B</td>
<td>0B</td>
<td>k</td>
</tr>
<tr>
<td>51</td>
<td>51</td>
<td>Q</td>
<td>6C</td>
<td>0C</td>
<td>l</td>
</tr>
<tr>
<td>52</td>
<td>52</td>
<td>R</td>
<td>6D</td>
<td>0D</td>
<td>m</td>
</tr>
<tr>
<td>53</td>
<td>53</td>
<td>S</td>
<td>6E</td>
<td>0E</td>
<td>n</td>
</tr>
<tr>
<td>54</td>
<td>54</td>
<td>T</td>
<td>6F</td>
<td>0F</td>
<td>o</td>
</tr>
<tr>
<td>55</td>
<td>55</td>
<td>U</td>
<td>70</td>
<td>10</td>
<td>p</td>
</tr>
<tr>
<td>56</td>
<td>56</td>
<td>V</td>
<td>71</td>
<td>11</td>
<td>q</td>
</tr>
<tr>
<td>57</td>
<td>57</td>
<td>W</td>
<td>72</td>
<td>12</td>
<td>r</td>
</tr>
<tr>
<td>58</td>
<td>58</td>
<td>X</td>
<td>73</td>
<td>13</td>
<td>s</td>
</tr>
<tr>
<td>59</td>
<td>59</td>
<td>Y</td>
<td>74</td>
<td>14</td>
<td>t</td>
</tr>
<tr>
<td>5A</td>
<td>5A</td>
<td>Z</td>
<td>75</td>
<td>15</td>
<td>u</td>
</tr>
<tr>
<td>5B</td>
<td>5B</td>
<td>[</td>
<td>76</td>
<td>16</td>
<td>v</td>
</tr>
<tr>
<td>5C</td>
<td>5C</td>
<td>\</td>
<td>77</td>
<td>17</td>
<td>w</td>
</tr>
<tr>
<td>5D</td>
<td>5D</td>
<td>]</td>
<td>78</td>
<td>18</td>
<td>x</td>
</tr>
<tr>
<td>5E</td>
<td>5E</td>
<td>Λ</td>
<td>79</td>
<td>19</td>
<td>y</td>
</tr>
<tr>
<td>5F</td>
<td>5F</td>
<td>_</td>
<td>7A</td>
<td>1A</td>
<td>z</td>
</tr>
<tr>
<td>60</td>
<td>—</td>
<td></td>
<td>7B</td>
<td>—</td>
<td>{</td>
</tr>
<tr>
<td>61</td>
<td>01</td>
<td>a</td>
<td>7C</td>
<td>—</td>
<td>:</td>
</tr>
<tr>
<td>62</td>
<td>02</td>
<td>b</td>
<td>7D</td>
<td>—</td>
<td>}</td>
</tr>
<tr>
<td>63</td>
<td>03</td>
<td>c</td>
<td>7E</td>
<td>—</td>
<td>~</td>
</tr>
<tr>
<td>64</td>
<td>04</td>
<td>d</td>
<td>7F</td>
<td>—</td>
<td>rub</td>
</tr>
</tbody>
</table>

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.
## SAM and VDG Settings

<table>
<thead>
<tr>
<th>Graphics Mode</th>
<th>Color Set</th>
<th>Register Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>To Set</strong></td>
</tr>
<tr>
<td>6R</td>
<td>0</td>
<td>LDA #$F0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LDA #$F8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td>6C</td>
<td>0</td>
<td>LDA #$E0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LDA #$E8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td>3R</td>
<td>0</td>
<td>LDA #$D0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LDA #$D8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td>Graphics Mode</td>
<td>Color Set</td>
<td>Register Instructions</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>3C</td>
<td>0</td>
<td>LDA #$C0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LDA #$C8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC5</td>
</tr>
<tr>
<td>2R</td>
<td>0</td>
<td>LDA #$B0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LDA #$B8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td>2C</td>
<td>0</td>
<td>LDA #$A0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LDA #$A8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC3</td>
</tr>
<tr>
<td>1R</td>
<td>0</td>
<td>LDA #$90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LDA #$98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC1</td>
</tr>
<tr>
<td>1C</td>
<td>0</td>
<td>LDA #$80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LDA #$88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FF22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STA $FFC1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Graphic Mode</th>
<th>Resolution</th>
<th>Element Size</th>
<th>Code</th>
<th>Color Set Set 0</th>
<th>Color Set Set 1</th>
<th>Memory Used (with no offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6R 256 x 192</td>
<td></td>
<td></td>
<td>0</td>
<td>black</td>
<td>black</td>
<td>0400-1BFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>green</td>
<td>buff</td>
<td></td>
</tr>
<tr>
<td>6C 128 x 192</td>
<td></td>
<td></td>
<td>00</td>
<td>green</td>
<td>buff</td>
<td>0400-1BFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01</td>
<td>yellow</td>
<td>cyan</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>blue</td>
<td>magenta</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>red</td>
<td>orange</td>
<td></td>
</tr>
<tr>
<td>3R 128 x 192</td>
<td></td>
<td></td>
<td>0</td>
<td>black</td>
<td>black</td>
<td>0400-0FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>green</td>
<td>buff</td>
<td></td>
</tr>
<tr>
<td>3C 128 x 96</td>
<td></td>
<td></td>
<td>00</td>
<td>green</td>
<td>buff</td>
<td>0400-0FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01</td>
<td>yellow</td>
<td>cyan</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>blue</td>
<td>magenta</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>red</td>
<td>orange</td>
<td></td>
</tr>
<tr>
<td>2R 128 x 96</td>
<td></td>
<td></td>
<td>0</td>
<td>black</td>
<td>black</td>
<td>0400-09FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>green</td>
<td>buff</td>
<td></td>
</tr>
<tr>
<td>2C 128 x 64</td>
<td></td>
<td></td>
<td>00</td>
<td>green</td>
<td>buff</td>
<td>0400-0BFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01</td>
<td>yellow</td>
<td>cyan</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>blue</td>
<td>magenta</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>red</td>
<td>orange</td>
<td></td>
</tr>
<tr>
<td>1R 128 x 64</td>
<td></td>
<td></td>
<td>0</td>
<td>black</td>
<td>black</td>
<td>0400-07FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>green</td>
<td>buff</td>
<td></td>
</tr>
<tr>
<td>1C 64 x 64</td>
<td></td>
<td></td>
<td>00</td>
<td>green</td>
<td>buff</td>
<td>0400-07FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01</td>
<td>yellow</td>
<td>cyan</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>blue</td>
<td>magenta</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>red</td>
<td>orange</td>
<td></td>
</tr>
</tbody>
</table>
## Screen Offsets

<table>
<thead>
<tr>
<th>Starting Address</th>
<th>To Set Write to Address(es)</th>
<th>To Reset to Zero Write to Address(es)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>none</td>
<td>$FFC6</td>
</tr>
<tr>
<td>0200</td>
<td>$FFC7</td>
<td></td>
</tr>
<tr>
<td>0400</td>
<td>$FFC9</td>
<td>$FFC8</td>
</tr>
<tr>
<td>0600</td>
<td>$FFC7,FFC9</td>
<td>$FFC6,FFC8</td>
</tr>
<tr>
<td>0800</td>
<td>$FFCB</td>
<td>$FFCA</td>
</tr>
<tr>
<td>0A00</td>
<td>$FFCB,FFC7</td>
<td>$FFCA,FFC6</td>
</tr>
<tr>
<td>0C00</td>
<td>$FFCB,FFC9</td>
<td>$FFCA,FFC8</td>
</tr>
<tr>
<td>0E00</td>
<td>$FFCB,FFC9,FFC7</td>
<td>$FFCA,FFC8,FFC6</td>
</tr>
<tr>
<td>1000</td>
<td>$FFCD</td>
<td>$FFCC</td>
</tr>
<tr>
<td>1200</td>
<td>$FFCD,FFC7</td>
<td>$FFCC,FFC6</td>
</tr>
<tr>
<td>1400</td>
<td>$FFCD,FFC9</td>
<td>$FFCC,FFC8</td>
</tr>
<tr>
<td>1600</td>
<td>$FFCD,FFC9,FFC7</td>
<td>$FFCC,FFC8,FFC6</td>
</tr>
<tr>
<td>1800</td>
<td>$FFCD,FFCB</td>
<td>$FFCC,FFCA</td>
</tr>
<tr>
<td>1A00</td>
<td>$FFCD,FFCB,FFC7</td>
<td>$FFCC,FFCA,FFC6</td>
</tr>
<tr>
<td>1C00</td>
<td>$FFCD,FFCB,FFC9</td>
<td>$FFCC,FFCA,FFC8</td>
</tr>
<tr>
<td>1E00</td>
<td>$FFCD,FFCB,FFC9,FFC7</td>
<td>$FFCC,FFCA,FFC8,FFC6</td>
</tr>
<tr>
<td>2000</td>
<td>$FFCF</td>
<td>$FFCE</td>
</tr>
</tbody>
</table>
The starting Address is changed by writing to the following registers.

<table>
<thead>
<tr>
<th>Offset Added</th>
<th>Write to Register</th>
<th>To Clear Offset Write to Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>$200</td>
<td>$FFC7</td>
<td>$FFC6</td>
</tr>
<tr>
<td>$400</td>
<td>$FFC9</td>
<td>$FFC8</td>
</tr>
<tr>
<td>$800</td>
<td>$FFCB</td>
<td>$FFCA</td>
</tr>
<tr>
<td>$1000</td>
<td>$FFCD</td>
<td>$FFCC</td>
</tr>
<tr>
<td>$2000</td>
<td>$FFCF</td>
<td>$FFCE</td>
</tr>
<tr>
<td>$4000</td>
<td>$FFD1</td>
<td>$FFD0</td>
</tr>
<tr>
<td>$8000</td>
<td>$FFD3</td>
<td>$FFD2</td>
</tr>
</tbody>
</table>

The computer is set to the text mode when it is turned on. The offset is automatically set to offset $400 at that time.
### Table to Determine Forward Branches

<table>
<thead>
<tr>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>49</td>
<td>31</td>
<td>97</td>
<td>61</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>50</td>
<td>32</td>
<td>98</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>51</td>
<td>33</td>
<td>99</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>52</td>
<td>34</td>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>53</td>
<td>35</td>
<td>101</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>54</td>
<td>36</td>
<td>102</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>55</td>
<td>37</td>
<td>103</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>56</td>
<td>38</td>
<td>104</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>57</td>
<td>39</td>
<td>105</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>58</td>
<td>3A</td>
<td>106</td>
<td>6A</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>59</td>
<td>3B</td>
<td>107</td>
<td>6B</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>60</td>
<td>3C</td>
<td>108</td>
<td>6C</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>61</td>
<td>3D</td>
<td>109</td>
<td>6D</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>62</td>
<td>3E</td>
<td>110</td>
<td>6E</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>63</td>
<td>3F</td>
<td>111</td>
<td>6F</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>64</td>
<td>40</td>
<td>112</td>
<td>70</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>65</td>
<td>41</td>
<td>113</td>
<td>71</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>66</td>
<td>42</td>
<td>114</td>
<td>72</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>67</td>
<td>43</td>
<td>115</td>
<td>73</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>68</td>
<td>44</td>
<td>116</td>
<td>74</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>69</td>
<td>45</td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>70</td>
<td>46</td>
<td>118</td>
<td>76</td>
</tr>
</tbody>
</table>
## APP. F. TABLE TO DETERMINE FORWARD BRANCHES

<table>
<thead>
<tr>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>17</td>
<td>71</td>
<td>47</td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>72</td>
<td>48</td>
<td>120</td>
<td>78</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>73</td>
<td>49</td>
<td>121</td>
<td>79</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>74</td>
<td>4A</td>
<td>122</td>
<td>7A</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>75</td>
<td>4B</td>
<td>123</td>
<td>7B</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>76</td>
<td>4C</td>
<td>124</td>
<td>7C</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>77</td>
<td>4D</td>
<td>125</td>
<td>7D</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>78</td>
<td>4E</td>
<td>126</td>
<td>7E</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>79</td>
<td>4F</td>
<td>127</td>
<td>7F</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>81</td>
<td>51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>82</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>83</td>
<td>53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>84</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>85</td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>86</td>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>87</td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>88</td>
<td>58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>89</td>
<td>59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>90</td>
<td>5A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>91</td>
<td>5B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>92</td>
<td>5C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>93</td>
<td>5D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>94</td>
<td>5E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>95</td>
<td>5F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>96</td>
<td>60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Appendix G

### Table to Determine Backward Branches

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

6809 Instruction Set
### 6809 Instruction Set (cont’d)

<table>
<thead>
<tr>
<th>Instr./Forms</th>
<th>Addressing Modes</th>
<th>Description</th>
<th>Flags Eff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABX</td>
<td>Inherent</td>
<td>B + X → X</td>
<td>none</td>
</tr>
<tr>
<td>ADC</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>A + M + C → A</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ADCB</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>B + M + C → B</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ADD</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>A + M → A</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ADDA</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>B + M → B</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ADDB</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>D + M, M + 1 → D</td>
<td>NZVC</td>
</tr>
<tr>
<td>ADDD</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>AAM → A</td>
<td>NZV</td>
</tr>
<tr>
<td>AND</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>BAM → B</td>
<td>NZV</td>
</tr>
<tr>
<td>ANDA</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>CCAIMM → CC</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ANDB</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>AAM → A</td>
<td>NZV</td>
</tr>
<tr>
<td>ANDCC</td>
<td>Immediate</td>
<td>CCAIMM → CC</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ASL</td>
<td>Inherent</td>
<td>A</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ASLA</td>
<td>Inherent</td>
<td>B</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ASLB</td>
<td>Direct, Extended, Indexed</td>
<td>M</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ASL</td>
<td>Direct, Extended, Indexed</td>
<td>A</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ASRA</td>
<td>Inherent</td>
<td>B</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ASRB</td>
<td>Direct, Extended, Indexed</td>
<td>M</td>
<td>HNZVC</td>
</tr>
<tr>
<td>ASR</td>
<td>Direct, Extended, Indexed</td>
<td>A</td>
<td>HNZC</td>
</tr>
<tr>
<td>BCC</td>
<td>Relative</td>
<td>Branch C = 0</td>
<td>none</td>
</tr>
<tr>
<td>BCS</td>
<td>Relative</td>
<td>Branch C = 1</td>
<td>none</td>
</tr>
<tr>
<td>LBCC</td>
<td>Relative</td>
<td>Long branch C = 0</td>
<td>none</td>
</tr>
<tr>
<td>LBCS</td>
<td>Relative</td>
<td>Long branch C = 1</td>
<td>none</td>
</tr>
<tr>
<td>BEQ</td>
<td>Relative</td>
<td>Branch Z = 0</td>
<td>Long branch Z = 0</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>--------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>LBEQ</td>
<td>Relative</td>
<td>Branch ≥ 0</td>
<td>Long branch ≥ 0</td>
</tr>
<tr>
<td>BGE</td>
<td>Relative</td>
<td>Branch &gt; 0</td>
<td>Long branch &gt; 0</td>
</tr>
<tr>
<td>LBGE</td>
<td>Relative</td>
<td>Branch higher</td>
<td>Long branch higher</td>
</tr>
<tr>
<td>BGT</td>
<td>Relative</td>
<td>Branch higher or =</td>
<td>Long branch higher or =</td>
</tr>
<tr>
<td>LBGT</td>
<td>Relative</td>
<td>Bit test A</td>
<td>Bit test B</td>
</tr>
<tr>
<td>BHI</td>
<td>Relative</td>
<td>Branch ≤ 0</td>
<td>Long branch ≤ 0</td>
</tr>
<tr>
<td>LBHI</td>
<td>Relative</td>
<td>Branch lower</td>
<td>Long branch lower</td>
</tr>
<tr>
<td>BHS</td>
<td>Relative</td>
<td>Branch lower or same</td>
<td>Long branch lower/same</td>
</tr>
<tr>
<td>LBHS</td>
<td>Relative</td>
<td>Branch &lt; 0</td>
<td>Long branch &lt; 0</td>
</tr>
<tr>
<td>BMI</td>
<td>Relative</td>
<td>Branch minus</td>
<td>Long branch minus</td>
</tr>
</tbody>
</table>
## 6809 Instruction Set (cont’d)

<table>
<thead>
<tr>
<th>Instr./Forms</th>
<th>Addressing Modes</th>
<th>Description</th>
<th>Flags Eff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE</td>
<td>Relative</td>
<td>Branch not =</td>
<td>none</td>
</tr>
<tr>
<td>LBNE</td>
<td>Relative</td>
<td>Long branch not =</td>
<td>none</td>
</tr>
<tr>
<td>BPL</td>
<td>Relative</td>
<td>Branch plus</td>
<td>none</td>
</tr>
<tr>
<td>LBPL</td>
<td>Relative</td>
<td>Long branch plus</td>
<td>none</td>
</tr>
<tr>
<td>BRA</td>
<td>Relative</td>
<td>Branch always</td>
<td>none</td>
</tr>
<tr>
<td>LBRA</td>
<td>Relative</td>
<td>Long branch always</td>
<td>none</td>
</tr>
<tr>
<td>BRN</td>
<td>Relative</td>
<td>Branch never</td>
<td>none</td>
</tr>
<tr>
<td>LBRN</td>
<td>Relative</td>
<td>Long branch never</td>
<td>none</td>
</tr>
<tr>
<td>BSR</td>
<td>Relative</td>
<td>Branch sub.</td>
<td>none</td>
</tr>
<tr>
<td>LBSR</td>
<td>Relative</td>
<td>Long branch subroutine</td>
<td>none</td>
</tr>
<tr>
<td>BVC</td>
<td>Relative</td>
<td>Branch V = 0</td>
<td>none</td>
</tr>
<tr>
<td>LBVC</td>
<td>Relative</td>
<td>Long branch V = 0</td>
<td>none</td>
</tr>
<tr>
<td>BVS</td>
<td>Relative</td>
<td>Branch V = 1</td>
<td>none</td>
</tr>
<tr>
<td>LBVS</td>
<td>Relative</td>
<td>Long branch V = 1</td>
<td>none</td>
</tr>
<tr>
<td>CLR</td>
<td>Inherent</td>
<td>0 → A</td>
<td>NZVC</td>
</tr>
<tr>
<td>CLRA</td>
<td>Inherent</td>
<td>0 → B</td>
<td>NZVC</td>
</tr>
<tr>
<td>CLRB</td>
<td>Direct, Extended, Indexed</td>
<td>0 → M</td>
<td>NZVC</td>
</tr>
<tr>
<td>CLR</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>A-M</td>
<td>HNZVC</td>
</tr>
<tr>
<td>CMP</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>B-M</td>
<td>HNZVC</td>
</tr>
<tr>
<td>CMPA</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>D-M, M + 1</td>
<td>NZVC</td>
</tr>
<tr>
<td>CMPB</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>S-M, M + 1</td>
<td>NZVC</td>
</tr>
<tr>
<td>CMPD</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>U-M, M + 1</td>
<td>NZVC</td>
</tr>
<tr>
<td>CMPS</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>X-M, M + 1</td>
<td>NZVC</td>
</tr>
<tr>
<td>CMPU</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>Y-M, M + 1</td>
<td>NZVC</td>
</tr>
<tr>
<td>CMPX</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPY</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COM</td>
<td>COMA</td>
<td>COMB</td>
<td>COMC</td>
</tr>
<tr>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>CWAI</td>
<td>DAA</td>
<td>DEC</td>
<td>DECA</td>
</tr>
<tr>
<td></td>
<td>Inherent</td>
<td>Inherent</td>
<td>Inherent</td>
</tr>
<tr>
<td></td>
<td>CCA, M - CC</td>
<td>AY - M - A</td>
<td>BY - M - B</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>R1 - R2</td>
<td>A + 1 - A</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>A - A</td>
<td>B - B</td>
<td>M - M</td>
<td>none</td>
</tr>
</tbody>
</table>

273
<table>
<thead>
<tr>
<th>Instr./Forms</th>
<th>Addressing Modes</th>
<th>Description</th>
<th>Flags Eff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>Indexed</td>
<td>EA→S</td>
<td>none</td>
</tr>
<tr>
<td>LEAU</td>
<td>Indexed</td>
<td>EA→U</td>
<td>none</td>
</tr>
<tr>
<td>LEAX</td>
<td>Indexed</td>
<td>EA→X</td>
<td>Z</td>
</tr>
<tr>
<td>LEAY</td>
<td>Indexed</td>
<td>EA→Y</td>
<td>Z</td>
</tr>
<tr>
<td>LSL</td>
<td>Inherent</td>
<td></td>
<td>NZVC</td>
</tr>
<tr>
<td>LSLA</td>
<td>Inherent</td>
<td></td>
<td>NZVC</td>
</tr>
<tr>
<td>LSLB</td>
<td>Direct, Extended, Indexed</td>
<td></td>
<td>NZVC</td>
</tr>
<tr>
<td>LSL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>Inherent</td>
<td></td>
<td>NZC</td>
</tr>
<tr>
<td>LSRA</td>
<td>Inherent</td>
<td></td>
<td>NZC</td>
</tr>
<tr>
<td>LSRB</td>
<td>Inherent</td>
<td></td>
<td>NZC</td>
</tr>
<tr>
<td>LSR</td>
<td>Direct, Extended, Indexed</td>
<td></td>
<td>NZC</td>
</tr>
<tr>
<td>MUL</td>
<td>Inherent</td>
<td>A x B→D</td>
<td>ZC</td>
</tr>
<tr>
<td>NEG</td>
<td>Inherent</td>
<td>A + 1→A</td>
<td>HNZVC</td>
</tr>
<tr>
<td>NEGA</td>
<td>Inherent</td>
<td>B + 1→B</td>
<td>HNZVC</td>
</tr>
<tr>
<td>NEGB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td>Direct, Extended, Indexed</td>
<td></td>
<td>HNZVC</td>
</tr>
<tr>
<td>NOP</td>
<td>Inherent</td>
<td>no operation</td>
<td>none</td>
</tr>
<tr>
<td>OR</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>AvM→A</td>
<td>NZV</td>
</tr>
<tr>
<td>ORA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORB</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>BvM→B</td>
<td>NZV</td>
</tr>
<tr>
<td>ORCC</td>
<td>Immediate</td>
<td>CCvIMM→CC</td>
<td>HNZVC</td>
</tr>
<tr>
<td>PSH</td>
<td>Immediate</td>
<td>Reg. to S</td>
<td>none</td>
</tr>
<tr>
<td>PSHS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSHU</td>
<td>Immediate</td>
<td>Reg. to U</td>
<td>none</td>
</tr>
<tr>
<td>PUL</td>
<td>PULS</td>
<td>Immediate</td>
<td>Reg. from S</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>PULU</td>
<td></td>
<td>Immediate</td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>ROLA</td>
<td>Inherent</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>ROLB</td>
<td>Inherent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROL</td>
<td>Direct, Extended, Indexed</td>
<td>M</td>
</tr>
<tr>
<td>ROR</td>
<td>RORA</td>
<td>Inherent</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>RORB</td>
<td>Inherent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROR</td>
<td>Direct, Extended, Indexed</td>
<td>M</td>
</tr>
<tr>
<td>RTI</td>
<td></td>
<td>Inherent</td>
<td>Ret. from Int</td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td>Inherent</td>
<td>Ret. from Sub.</td>
</tr>
<tr>
<td>SBC</td>
<td>SBCA</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>A(\cdot)M\cdot)C\cdot)A</td>
</tr>
<tr>
<td></td>
<td>SBCB</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>B(\cdot)M\cdot)C\cdot)B</td>
</tr>
<tr>
<td>SEX</td>
<td></td>
<td>Inherent</td>
<td>sign ext. B\cdot)A</td>
</tr>
<tr>
<td>ST</td>
<td>STA</td>
<td>Direct, Extended, Indexed</td>
<td>A(\cdot)M</td>
</tr>
<tr>
<td></td>
<td>STB</td>
<td>Direct, Extended, Indexed</td>
<td>B(\cdot)M</td>
</tr>
<tr>
<td></td>
<td>STD</td>
<td>Direct, Extended, Indexed</td>
<td>D(\cdot)M\cdot)M + 1</td>
</tr>
<tr>
<td></td>
<td>STS</td>
<td>Direct, Extended, Indexed</td>
<td>S(\cdot)M\cdot)M + 1</td>
</tr>
<tr>
<td></td>
<td>STU</td>
<td>Direct, Extended, Indexed</td>
<td>U(\cdot)M\cdot)M + 1</td>
</tr>
<tr>
<td></td>
<td>STX</td>
<td>Direct, Extended, Indexed</td>
<td>X(\cdot)M\cdot)M + 1</td>
</tr>
<tr>
<td></td>
<td>STY</td>
<td>Direct, Extended, Indexed</td>
<td>Y(\cdot)M\cdot)M + 1</td>
</tr>
<tr>
<td>Instr./Forms</td>
<td>Addressing Modes</td>
<td>Description</td>
<td>Flags Eff.</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>SUB</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>A-M A</td>
<td>HNZVC</td>
</tr>
<tr>
<td>SUBA</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>B-M B</td>
<td>HNZVC</td>
</tr>
<tr>
<td>SUBB</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td>D-M, M+1 D</td>
<td>NZVC</td>
</tr>
<tr>
<td>SUBD</td>
<td>Direct, Extended, Immediate, Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWI</td>
<td>Inherent</td>
<td>Softw. Int.1</td>
<td>none</td>
</tr>
<tr>
<td>SWI2</td>
<td>Inherent</td>
<td>Softw. Int.2</td>
<td>none</td>
</tr>
<tr>
<td>SWI3</td>
<td>Inherent</td>
<td>Softw. Int.3</td>
<td>none</td>
</tr>
<tr>
<td>SYNC</td>
<td>Inherent</td>
<td>Synch to Int.</td>
<td>none</td>
</tr>
<tr>
<td>TFR</td>
<td>R1, R2</td>
<td>R1 R2</td>
<td>none</td>
</tr>
<tr>
<td>TST</td>
<td>Inherent</td>
<td>Test A</td>
<td>NZV</td>
</tr>
<tr>
<td>TSTA</td>
<td>Inherent</td>
<td>Test B</td>
<td>NZV</td>
</tr>
<tr>
<td>TSTB</td>
<td>Inherent</td>
<td>Test M</td>
<td>NZV</td>
</tr>
<tr>
<td>TST</td>
<td>Direct, Extended, Indexed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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  
A is logical AND  
V is logical OR  
Y is logical EXCLUSIVE OR
Index

ABUG monitor, 46, 55, 65, 232
Accumulator A, 13, 22, 29, 31, 43
Accumulator B, 15, 22, 29, 31, 43
Accumulator D, 15, 22
Adding backspace to word processor, 182
Adding sound to programs, 130
Amplitude for sound, 130
Animation by joystick, 151
Animation by paging, 113
Animation with sound, 139
ASCII codes for characters, 199
Assembler, 45, 53, 57, 65
Assembler options, 53, 65
Assembler program, 46, 53, 65
Assembler, single-step, 53
Assembler symbolism, 25, 42
Automatic key repeat in Editor, 48
Backward branches, 21
BASIC to machine language link, 226
Binary/hex/decimal equivalents, 3
Binary/hex relationships, 4, 22
Binary number system, 4, 22
Binary symbols, 1
Bit, 1
Branch bytes backward, 21
Branch bytes forward, 20
Branch destinations, 20, 21
Buffer, 48, 177
Buffer address, 178
Burn an EPROM, 225
Byte, 2
Cartridge slot, 225
CBUG monitor, 6
Central Processing Unit (CPU), 2
CLEAR, 233
CLOADM, 42, 233
Clock interrupt, 159
Color codes, 60, 73, 93, 98, 119
Color Computer News, 175
Color set 0, 60, 73, 98, 119, 169
Color set 1, 73, 98, 119, 169
Commands for Figure Creator, 238
Comment field for assembler, 48
Condition codes, 34, 56
Condition Code Register, 34, 43, 56, 158
Condition flags, 36
Creating graphic figures, 237
Creating text characters, 188
Cursor in Editor, 48
Data table for Figure Creator, 239
Decimal/binary/hex equivalents, 3
Decimal use with assembler, 156
Define user statement (DEFUSR), 237
Designing a graphics program, 58
Designing a joystick program, 151
Disable IRQ, 158
Displaying text by graphics, 190, 198, 218
Display offset registers, 111, 120
Double accumulator D, 15, 22

Editor, 46, 65
Enable IRQ, 158
EPROM, 189, 225
EPROM programmer, 225, 235
Equate statements, 125, 176, 293
Execute from ABUG, 55
Exit from ABUG, 57

Facts, The, 125
Fast interrupt (FJIRQ), 158
Fields for assembler, 48
Flags, 35
Flowchart of program 1, 14
Flowchart of program 2, 30
Flying saucer program, 154, 164
Forward branches, 20
Four-color mode 6C, 58, 66, 70

Graphic characters from keyboard, 206
Graphic elements for 4-color modes, 70, 73, 93, 198
Graphic elements for 2-color modes, 97, 119
Graphic mode 1R, 100, 104
Graphic program design, 58

Hand assembly, 45
Hexadecimal number system, 3
Hex/binary/decimal equivalents, 3
Hex/binary relationships, 4, 22
Horizontal joystick, 153

Index register, 31
Information on programs in
Chapter 6, 143
INPUT port $FF00, 105, 120
Instruction classification, 17
Instruction decoder, 2
Instructions for 4-color modes, 72
Instructions for 2-color modes, 99
Instruction summary for Chapters 1-4, 91

Instructions used
in Chapter 5, 118
in Chapter 6, 144
in Chapter 7, 171
in Chapter 8, 193
in Chapter 9, 219
in Program 1, 17
in Program 2, 28
in Program 3, 39
in Program 4, 57
in Program 5, 63
in Program 6, 78

Integer convert routine (INTCNV), 226
Internal components, Color Computer, 125
Interrupt request, 158
Interrupts, 157
Interrupt service routine, 157, 160

Interrupt, software, 32
IRQ, 158
IRQ vector, 159

JOYST, 152, 170
Joystick animation, 151
Joystick horizontal control, 152, 170
Joystick program, 151
Joystick vertical control, 152, 170

Keyboard graphic characters, 206
Keyboard graphic text, 209, 213
INDEX

Keyboard matrix, 106
Keyboard read, 105, 120
Keyboard scanning routine ($A1B1), 175
Keyboard sounds, 249

Label field of assembler, 48
Line insert command of editor, 48
Linking BASIC to machine language, 226
Loading taped programs, 42, 233

Machine language, 1
Machine language capabilities and shortcomings, 1
Machine language instructions—POKEd, 7
Machine language monitor, 5, 22, 46, 55, 65
Machine language operation codes, 25, 29
Machine language tape, 232
Matrix, keyboard, 105
Memory for 4-color modes, 70, 74, 86, 93
Memory for mode 6C, 59, 66, 70
Memory for 2-color modes, 97, 119
Message prompt for inputs, 208, 249
Micro Works, The, 6, 45
Mnemonic field for assembler, 49
Mode 6C, 58

Negative flag, 35
Non-maskable interrupt (NMI), 158

Object program, 53, 55
Object program, recording, 232
Op code field for assembler, 48
Operand field for assembler, 48
Operation codes, 25, 29
Origin of program (ORG), 63, 126
OUTPUT port $FF02, 105, 120

Output to printer routine ($A2BF), 175, 193
Pages of memory, 59
Paging, 111
Peripheral interface adapter (PIA), 69, 159
Placing graphic text on screen, 201, 207
POKE machine language program, 7
POLCAT, 175, 193
Preparing a machine language tape, 232
PRINT, 175, 193
Printer output routine ($A2BF), 175, 193
Print to screen routine ($A30A), 175, 193
Program counter, 10, 29, 43

RAM controller, 69
Read keyboard, 105, 120
Recording on tape, 41, 232
Registers, 10, 22, 29, 43
A, 13, 22, 29, 31, 43
B, 15, 22, 29, 31, 43
CC, 34, 43, 56, 158
D, 15, 22
X, 29, 31, 43
Y, 37, 43
Registers for 4-color graphics, 71, 93
Registers for 2-color graphics, 99, 119
Reserve memory bytes, 116, 126, 176, 193
Resolution for 4-color graphics, 70, 189, 198
Resolution for 2-color graphics, 97, 119
ROM cartridge, 225
ROM subroutines for integer conversion, 226
joysticks, 152
keyboard scan, 175
print to printer, 175
print to video, 175
SAM, 69, 71, 97, 119
Save command, 41
Saving programs on tape, 41, 61, 232
Screen for Figure Creator, 238
Screen positions for graphic text, 201
Screen positions for joysticks, 153
Second byte of relative branches, 20
Selecting graphic characters from keyboard, 206
Signed numbers, 18, 23
Signed number wheel, 19
Single-step mode of assembler, 53
6809 microprocessor, 2, 10, 22
Size elements for 4-color modes, 70, 73, 93, 198
Size elements for 2-color modes, 97, 119
Software Development System, 45
Software interrupt (SWI), 32
Sound experiments, 125, 128
Sounds from the keyboard, 249
Sound table, 37
Source buffer, 48
Source program, 50, 53
Spectral Associates, 225
Status of computer, 35
SWI (software interrupt), 32
Tape saves, 41, 61, 232
Text buffer, 46, 177, 186
Text characters by graphics, 188, 197
Text processor, 175
Timing using IRQ, 162
Two-color modes, 97
User stack, 177
Using an EPROM programmer, 235
USR function, 225
VDG (Video Display Generator), 69, 71, 97, 119
Vertical, joystick, 153
Word processor, 180
X register, 29, 31, 43
Y register, 37, 43
Zero flag, 35
Zero force insertion socket (ZIF), 226
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:
- Introduction to Machine Language
- Sound
- Edit, Assemble, and ABUG
- Color Graphics
- Animation
- Sound and Graphics
- Joystick Animation
- Text
- Graphics with Text
- Vistas Beyond

Cover Design by Debbie Balboni

RESTON PUBLISHING COMPANY, INC.
A Prentice Hall Company
Reston, Virginia

0-8359-0317-6