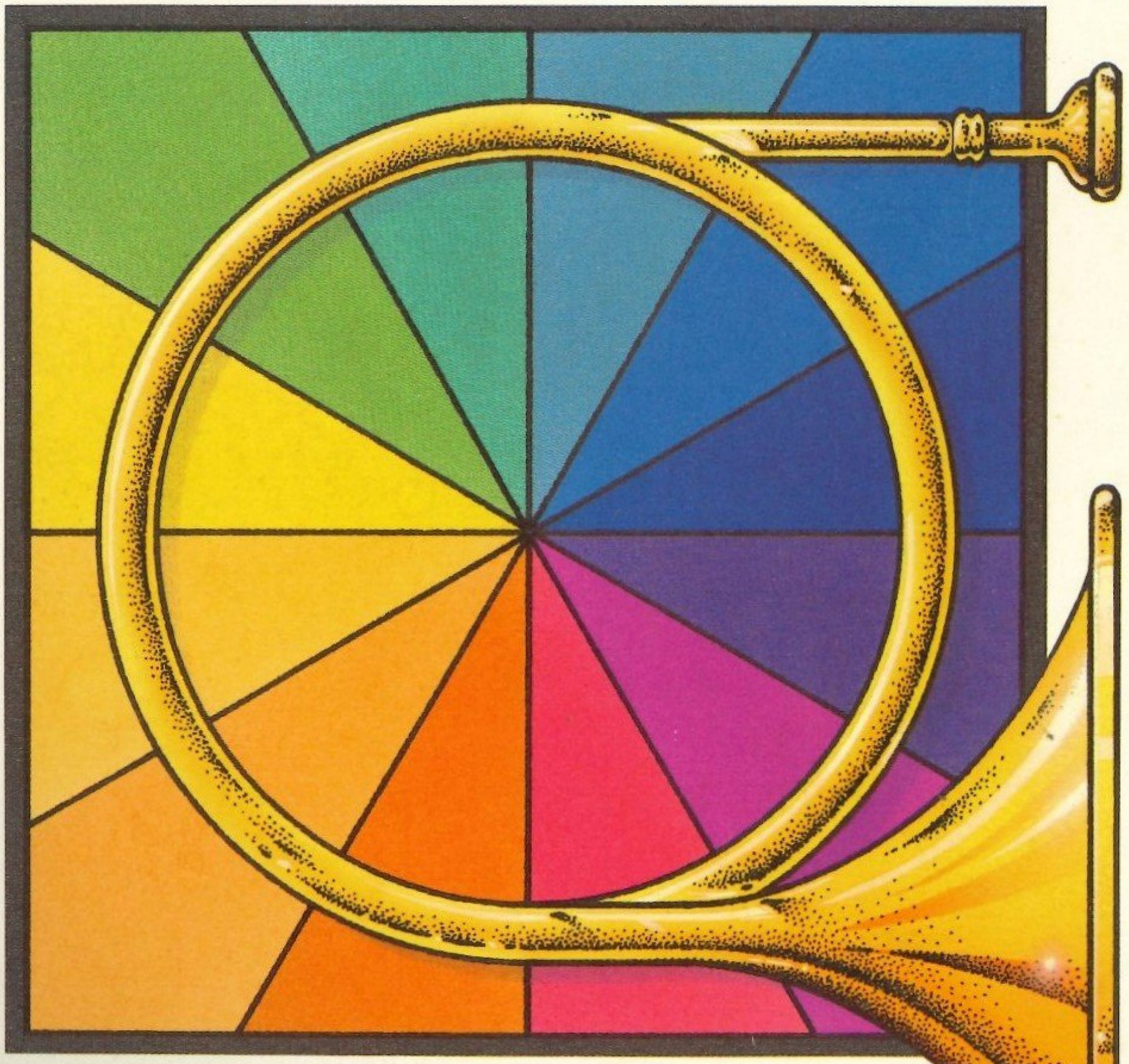
COMPUTE!'s GUIDE TO SOUND & GRAPHICS

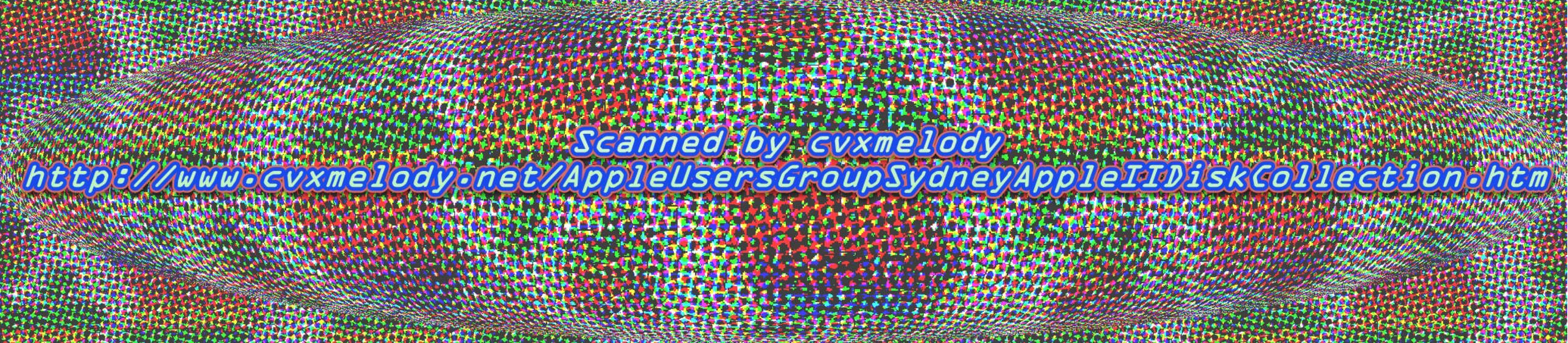


ON THE APPLE IIGS

William B. Sanders

The perfect introduction to programming sound and graphics on the dazzling new Apple IIGS.

COMPUTE! LIBRARY SELECTION


Scanned by cvxmelody
http://www.cvxmelody.net/AppleUsersGroupSydneyAppleIIDiskCollection.htm

COMPUTE!'s GUIDE TO SOUND AND GRAPHICS on the Apple IIGS

William B. Sanders

COMPUTE! Publications, Inc.
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-096-3

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Apple is a registered trademark and Apple IIGS is a trademark of Apple Computer, Inc.

Table of Contents

Foreword

Years ago, Apple promised "Apple II forever." The Apple IIGS is the proof that Apple wasn't kidding. The newest addition to the ten-year-old Apple II line, the IIGS is a powerful computer with new features and capabilities, yet one that is compatible with most Apple software.

That combination—added power and compatibility—extends into the programming arena as well. If you've programmed on the Apple II+, IIe, or IIc, you won't have trouble with the standard graphics modes on the Apple IIGS. But the IIGS-specific graphics and sound features, like super high-resolution graphics and sophisticated sounds, are new to everyone. You may be confused at first. How can you create super hi-res pictures, for instance? How can you get to the IIGS's Toolbox?

COMPUTE!'s Guide to Sound and Graphics on the Apple IIGS shows you how. Written by William Sanders, well-known author of such books as *The Elementary Apple* and *The Elementary Apple IIGS*, it's full of programming examples, tips, and techniques that illustrate the power of your new computer.

From the first chapter to the last, Sanders takes you on a guided programming tour of the Apple IIGS. You'll be introduced to the text screen and shown how that simple mode can generate useful graphics. You'll discover the fundamentals of animation. You'll explore the lo-res and hi-res Apple graphics modes, and will even see how to create shapes and bitmapped graphics. You'll see practical programming applications, such as drawing charts and graphs. You'll find out how to access the super hi-res mode and how to make superb music on the IIGS.

Scores of BASIC and machine language programs and routines amply demonstrate the programming techniques Sanders offers. Step-by-step instructions outline what you need to do and when to do it. The clear writing leaves nothing to puzzle you.

COMPUTE!'s Guide to Sound and Graphics on the Apple IIGS is your introduction and instructor to the world of sound and graphics programming on the newest, hottest Apple II computer ever. With it and your imagination, you can turn out your most dazzling programs ever.

Acknowledgments

Just about every project requires the acknowledged author to ritualistically thank others for their help. While several of those who lent their assistance to this project suggested I send them lots of money instead of a mere acknowledgment, it was impossible for me to bestow filthy lucre upon such noble altruism. Therefore, it is with pure sincerity, and in this case more than ritualistic gratitude, that these acknowledgments are made.

The first three people who made it possible for this book to be completed with some semblance of schedule were Roger Wagner, Roger Wagner and Roger Wagner. First of all, Roger helped with enlightening me to the mysteries of the new Apple IIGS Toolbox. For the super high-resolution graphics and DOC sounds, this was absolutely required. Secondly, he provided me with prerelease versions of the *Merlin Assembler* for the 65816 chip that's the heart of the Apple IIGS. Third, Roger was generally encouraging and helpful in organizing excuses to our long-suffering editor who regularly, but patiently, inquired as to the completion date.

Martha Steffen of Apple Computer, Inc. provided a prototype Apple IIGS along with a ton of documentation and a great deal of invaluable software. With all of the demands placed upon Martha by developers and writers, her assistance is doubly appreciated. Eric Goez, Bill Rupp, and many others in the Apple world in San Diego provided different types of information right at the moment it seemed to be most needed. My aforementioned, long-suffering editor, Stephen Levy, is a case study of what a good editor should be: helpful, encouraging, and understanding in the complexity of preparing books for new computers. Likewise I am grateful to Bill Gladstone of Waterside Productions for arranging things to make the book possible.

Finally, my family has been wonderful in this occupation—where one is at home writing but not at home. My sons Bill and David were considerate in keeping typical teenaged sounds down to a bearable din and allowing me occasional use of the phone. My wife Eli provided an intangible warmth, and our brave dog Jingle kept robbers and ax murderers at bay while the work proceeded.

William Sanders
April, 1987

Chapter 1
Text Graphics

It might seem strange to begin a book on graphics with a discussion of *text-graphics*—but there *is* a good reason. First of all, a number of concepts introduced in this chapter will later be applied to various graphics applications, and in order to focus on the concept and not the graphic element, text will serve better. Second, there are many graphic elements which can be introduced using text. This is especially true with combinations of inverse and regular video. Using full graphics screens, you loose certain types of built-in text capabilities. However, using full text screens, you can arrange various inverse blocks of text to create graphic elements along with all of the built-in Apple IIGS screen fonts and characters. Finally, it will be a relatively easy introduction to *screen mapping*. Screen mapping refers to addresses in your GS's memory that correspond to locations on the screen. If you insert a given code in a given address, some figure will appear in the corresponding location on the screen.

Normal, Inverse, and Flash

To get started, initialize a blank disk. While learning new concepts and techniques, it's a good idea to use a disk which you can accidentally destroy without losing vital data. Keep a second disk handy to transfer and save those programs that you want to keep. Then you'll have both a disk of expendable experimental programs and another of hold-and-keep programs.

Spaces

An important thing to understand about text graphics is the *space*. Not just *space*, but *the space*—the empty room between words. In Applesoft BASIC, there are two ways to make a space. First, you

can define a space by placing two quotation marks with a space between them, as the next line illustrates.

```
S$=" "  (One or more spaces between quotation marks)
```

The second method is to use the SPC function, which places a specified number of spaces sequentially on the screen. It's different from the defined space. First, you cannot define SPC as part of a variable; second, SPC places spaces sequentially when there's no formatting symbol (semicolor or comma) following it. The following shows SPC used in a typical format:

```
30 PRINT SPC(5)
```

That produces the same result as

```
30 PRINT "     ";: REM 5 spaces between quote marks
```

INVERSE

It's a bit difficult to see a space unless it's against a contrasting background. You can easily reverse the screen on your Apple with the INVERSE statement. Let's take a look at a simple program to get going.

Program 1-1.

```
10 TEXT : HOME
20 S$="     " : REM 5 SPACES
30 INVERSE
40 PRINT S$
50 PRINT
60 PRINT SPC(10)
70 NORMAL
```

This program puts a couple of bars on your screen. Conceivably, you could put anything you wanted on your Apple's screen simply by adjusting the number of spaces between the quotes or the value of SPC. That's exactly what you'll do, but you'll let your Apple do most of the thinking.

Creating graphics programs is much the same as creating any other kind of program. If you were writing a checkbook program, for example, you wouldn't write it so that the user has to do all the calculations. You want the computer to do the work. Similarly,

with graphics programming you don't want to do all the calculations manually when your computer can do them, and do them much faster. This not only saves you programming time, but helps you think of mathematically-generated designs.

To see an example of this concept, let's create a stairway. You'll do it two ways—the hard way first, then the easy (and smart) way.

Program 1-2.

```
10 TEXT : HOME
20 INVERSE
30 PRINT " " : REM 1 SPACE
40 PRINT "  " : REM 2 SPACES
50 PRINT "   " : REM 3 SPACES
60 PRINT "    " : REM 4 SPACES
```

. . . Continue adding one space to each PRINT statement until you reach 22.

```
240 PRINT "                      " : REM 22 SPACES
250 NORMAL
```

That's a lot of work to get a staircase. Try it again, using the next program.

Program 1-3.

```
10 TEXT : HOME
20 INVERSE
30 FOR X = 1 TO 22
40 S$ = S$ + " "
50 PRINT S$
60 NEXT
70 NORMAL
```

This program builds the size of the bars by concatenating S$ with a space each time through the loop. Notice that the second version took only one-third as many lines as the first. In other words, it was three times as much work to get the same results using the first method. That's what was meant by the *hard way* and the *smart way.*

Let's do the same thing with SPC. You can do it in only seven lines since the loop variable (X) increases the length of each bar,

avoiding the necessity of concatenating a string variable. You also don't have to define S$, though a line to provide the *line feed* or *carriage return* after printing SPC must be added.

Program 1-4.

```
10 TEXT : HOME
20 INVERSE
30 FOR X = 1 TO 22
40 PRINT SPC( X)
50 PRINT
60 NEXT
70 NORMAL
```

These stairs look pretty steep. By changing a single line in the program—line 30—you can make them easier to negotiate.

Program 1-5.

```
10 TEXT : HOME
20 INVERSE
30 FOR X = 1 TO 40 STEP 2
40 PRINT SPC( X)
50 PRINT
60 NEXT
70 NORMAL
```

A simple change in the program alters the entire graphic design. In later chapters, you'll see again and again how to let your Apple do the thinking and calculating for you.

FLASH

The FLASH statement is used sparingly in programming since it's distracting and hard to read. However, it has some useful purposes in text graphics. For example, you could make a flashing poster with the next program.

Program 1-6.

```
10 TEXT : HOME
20 FLASH
30 FOR X = 1 TO 40
40 S$ = S$ + " "
```



```
50 NEXT
60 FOR X = 1 TO 22
70 PRINT S$;
80 NEXT
90 VOTE$ = "VOTE FOR SENATOR SNORT"
100 VTAB 12: HTAB 20 - LEN (VOTE$) / 2
110 PRINT VOTE$
120 TEXT
130 NORMAL
```

See if you can change the program to put a border of asterisks (*) around the campaign poster.

By combining inverse and normal backgrounds, and adding a little sound, you can make something that will really get people's attention. (The following program gets a little complicated, but it uses the same principles you've just seen.)

Program 1-7.

```
10 TEXT : HOME :F$ = "=": PRINT "ENTER MESSAGE HERE:":
   PRINT "(MUST BE EVEN # OF CHARACTERS.)": INPUT "==>
   ";YP$:YP$ = "**" + YP$ + "**":P = LEN (YP$)
20 HOME :LM = 20 - LEN (YP$) / 2
30 IF LEN (F$) < > 40 THEN F$ = F$ + "=": GOTO 30
40 PRINT F$;: FOR I = 1 TO 15: PRINT "I"; SPC( 38);"I";: NEXT :
   PRINT F$
50 INVERSE : FOR I = 2 TO 16: HTAB 2: VTAB I: PRINT SPC( 38):
   NEXT : NORMAL : FOR PAUSE = 1 TO 1000: NEXT
60 FOR K = 2 TO 16: FOR W = 20 TO 21: VTAB K: HTAB W: PRINT
   SPC( 1): NEXT : NEXT : FOR I = 2 TO LEN (YP$) / 2
70 FOR J = 2 TO 16: VTAB J: HTAB I + 20: PRINT SPC( 1): VTAB
   J: HTAB 21 - I: PRINT SPC( 1): NEXT : NEXT
80 VTAB 22
90 SS$ = "*"
100 IF LEN (SS$) < 40 THEN SS$ = SS$ + "*": GOTO 100
110 FLASH : HTAB 1: VTAB 18: PRINT SS$: NORMAL
120 SPEED= 150
130 L$ = LEFT$ (YP$, LEN (YP$) / 2)
140 R$ = RIGHT$ (YP$, LEN (YP$) / 2)
150 FOR V = 1 TO ( LEN (YP$) / 2)
160 VTAB 9: HTAB 20 + V: PRINT MID$ (R$,V,1): GOSUB 250
```

```
170 IF V = ( LEN (L$)) + 1 THEN 190
180 VTAB 9: HTAB 21 - V: PRINT MID$ (L$, LEN (L$) - (V - 1),1):
    GOSUB 250
190 NEXT : SPEED= 255
200 INVERSE : VTAB 22:H$ = " <HIT ANY KEY TO CONTINUE> ":
    HTAB 20 - LEN (H$) / 2: PRINT H$: NORMAL
210 WAIT - 16384,128: POKE - 16368,0
220 VTAB 16: HTAB LM: POKE 32,LM: POKE 35,16: POKE 33,P:
    POKE 34,2
230 FOR I = 1 TO 16: FOR J = 1 TO 50: NEXT J: CALL - 912: NEXT
240 VTAB 22: TEXT : FOR I = 1 TO 24: FOR J = 1 TO 50: NEXT J:
    CALL - 912: NEXT : END
250 BZ = 49200: FOR I = 1 TO 15: FOR J = 1 TO I * (J - 1): NEXT :B =
    PEEK (BZ): NEXT
260 RETURN
```

A couple of things worth noting in this program are the sound generation and the WAIT routine. Line 250 generates a buzzing sound by PEEKing the speaker address, but it doesn't use the full sound power of your IIGS. That will come later in this book, and involves the Toolbox and the Ensoniq chip.

The WAIT routine on line 210 does two things. First, it holds everything until a key is hit, then it clears the keyboard buffer. Secondly, it waits for this keypress without a cursor or prompt to spoil your graphics display. You may be more familiar with the GET statement. Change line 210 to

```
210 GET A$
```

and see the difference.

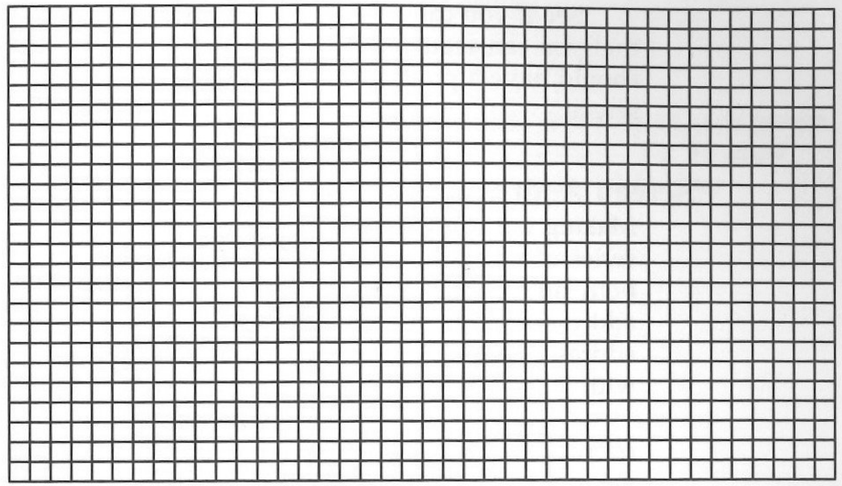
Mapping the Text Screen

The easiest way to begin learning about mapping the IIGS screen is with the 40-column text screen. The Apple IIGS text screen is a grid of 24 rows and 40 columns.

The upper left corner is *column 1, row 1* and the lower right corner is *column 40, row 24*. Thus, if you think of the screen in terms of *x* coordinates for the horizontal position and *y* coordinates

for the vertical position, you can define an *X,Y* coordinate system. The 1,1 position is the upper left, and the 40,24 position is the lower right. Here's a grid with all of the points on the 40-column screen.

Figure 1-1. 40-Column Text Screen



Under this same coordinate scheme, the middle of the screen is defined as position 20,12. The entire coordinate system is simple to use, since you can plot anywhere you want.

To become familiar with the system, use the following program to place inverse spaces on the screen. The program uses CHR$(32), the ASCII code for a space.

Program 1-8.

```
10 TEXT : HOME : COUNT = 0
20 INPUT "HOW MANY PLOTS ";N
30 HOME
40 HTAB 1: VTAB 1
50 PRINT "X POSITION=";
60 INPUT X
70 HTAB 1: VTAB 1
80 PRINT "Y POSITION=";
90 INPUT Y
```

```
100 HTAB X: VTAB Y
110 INVERSE
120 PRINT CHR$ (32)
130 NORMAL
140 COUNT = COUNT + 1
150 IF COUNT < > N THEN 40
```

See if you can draw a box with this program. Plan ahead so that you can sequentially plot your box. Once you can do that, you should have a pretty good idea where everything goes on the screen.

Screen Addresses

Making the conceptual jump from understanding the screen as a series of *x,y* coordinates to understanding the screen's addresses is both simple and confounding at the same time. It's simple since the addresses in a row are sequential, but it's confounding since the addresses are not sequential from one row to the next. Let's start with the simple part.

While thinking of your screen as a series of column and row coordinates, also think of it as a series of addresses. If you place a value into a screen address with a POKE statement, a character will appear on the screen.

The upper left corner is address 1024 ($400 in *hexadecimal*). The first row is made up of addresses 1024–1063. To see how to POKE a character in the text screen, use an inverse space. POKE the value 32 into an address in the first row to create an inverse space.

The 32 you POKEd into memory is a screen code. It is different from the CHR$(32) we used before. (CHR$(32) is a normal space, the screen code 32 is an inverse space.)

Program 1-9.

```
10 TEXT : HOME
20 FOR X=1024 TO 1024 + 39
30 POKE X,32
40 NEXT X
```

That small program put an inverse bar across the top of your screen. Now let's try going another row or 40 addresses farther (for

40 columns) and see what happens. Change line 20 to read

```
20 FOR X = 1024 TO 1024 + 79
```

Now when you run the revised program, you'll see two bars separated by a considerable space, not stacked one on top of another as you'd expect. This is because the Apple's screen memory is not sequential from one row to the next.

Bars Together

On the Apple IIGS, the address of the first character of each screen line is 128 ($80), higher than that of the previous line. To draw two adjacent bars, then, the program would look like this:

Program 1-10.

```
10 TEXT : HOME
20 FOR X = 1024 TO 1024 + 39
30 POKE X,32
40 NEXT
50 FOR X = 1152 TO 1152 + 39
60 POKE X,32
70 NEXT
```

Now you can begin to discern the pattern of screen addresses on the IIGS. Refer to the table below for the address of each row's beginning.

Row	Address
1	1024
2	1152
3	1280
4	1408
5	1536
6	1664
7	1792
8	1920

After the eighth row, the sequence begins again with address 1064. Before continuing, however, type in and run the following program.

Program 1-11.

```
10 TEXT : HOME
20 FOR X = 1024 TO 1920 STEP 128
30 POKE X,32
40 NEXT
```

If there's a vertical bar, you know you're on the right track. Just for fun and practice, try POKEing any screen address, and then that address plus 128. You'll get stacked bars all over the screen.

Character Patterns

Generating inverse and normal screen characters can make interesting patterns. By POKEing normal and inverse spaces to the screen, you can draw low-resolution figures in black and white. The difference in screen code values between an inverse and a normal space is 128, so a normal space's value is 160 (128 + 32).

This next program draws alternating normal and inverse spaces.

Program 1-12.

```
10 TEXT : HOME : V = 0
20 FOR X = 1024 TO 1024 + 39
30 POKE X,32 + V
40 IF V = 0 THEN V = 128 : NEXT
50 IF V = 128 THEN V = 0 : NEXT
```

In addition to using the POKE statement to put something on the screen, you can use the PEEK command to see what's there. For example, if you used statements like

```
PRINT PEEK (1024)
PRINT PEEK (1025)
PRINT PEEK (1026)
```

and so on, to examine the top line after running Program 1-12, you'd find alternating values of 32 and 160. Using this information, you could write a program that would switch light and dark spaces, giving the sensation of movement. Program 1-13 does just that.

Program 1-13.

```
 10 TEXT : HOME : V = 32
 20 FOR X = 1024 TO 1063
 30 POKE X, V
 40 IF V = 32 THEN V = 160 : NEXT
 50 V = 32 : NEXT
 60 FOR W = 1 TO 20
 70 FOR X = 1024 TO 1063
80 POKE X,V : V = V + 128 : IF V
   > 160 THEN V = 32
90 NEXT X : V = V + 128 : IF V
   > 160 THEN V = 32
100 NEXT W
```

The program draws a sequence of normal and inversed spaces across the top of the screen. It then enters a loop which switches the normal spaces to inversed ones, and the inversed spaces to normal ones. This is what produces the animated affect. The following program scans the screen memory sequentially and inverses whatever it finds.

Program 1-14.

```
 10 FOR PA = 1024 TO 1104 STEP 40
 20 FOR X = PA TO 2039 STEP 128
 30 FOR SCREEN = 0 TO 39
 40 N = PEEK (X + SCREEN)
 50 IF N > = 192 THEN F = N - 192
 60 IF N < 192 THEN F = N - 128
 70 IF N < 160 AND N > 31 THEN F = N + 128
 80 IF N < 32 THEN F = N + 192
 90 POKE X + SCREEN,F
100 NEXT SCREEN
110 NEXT X: NEXT PA
```

Machine Language Speed

You can greatly enhance the effect of text graphics by speeding up the process. This can be done by using the native language of your Apple IIGS, *machine language*. The POKEs and PEEKs you've been using are actually simple machine language routines included

within BASIC. In later chapters, when you get into the Toolbox, you'll be using more machine language programming.

For now, though, let's keep it simple. You'll load a space (the value 160) into a register, and then will move it from that register to screen memory. You'll do it sequentially from top to bottom, half from right to left and the other half from left to right.

The process will happen so fast that it will seem like an invisible hand is wiping the screen clean with two swipes—It's so fast that you won't be able to see the addresses filled with spaces.

The first listing below is in BASIC, and the second is in assembly language source code. (If you don't have an assembler, use the mini-assembler built into your Apple IIGS. Just type **CALL −151** and press Return; then when you see the asterisk prompt, enter an exclamation point (!) and press Return again. When the exclamation point prompt appears, you're in the mini-assembler. See your Apple IIGS reference manual for an explanation of how to use the mini-assembler.)

Program 1-15.

```
10 FOR X = 32768 TO 32945
20 READ D: POKE X,D: NEXT X
30 CALL 32768
9000 DATA 169, 160, 162, 39, 157
9010 DATA 0, 4, 157, 128, 4, 157
9020 DATA 0, 5, 157, 128, 5, 157
9030 DATA 0, 6, 157, 128, 6, 157
9040 DATA 0, 7, 157, 128, 7, 157
9050 DATA 40, 4, 157, 168, 4, 157
9060 DATA 40, 5, 157, 168, 5, 157
9070 DATA 40, 6, 157, 168, 6, 157
9080 DATA 40, 7, 157, 168, 7, 157
9090 DATA 80, 4, 157, 208, 4, 157
9100 DATA 80, 5, 157, 208, 5, 157
9110 DATA 80, 6, 157, 208, 6, 157
9120 DATA 80, 7, 157, 208, 7, 169
9130 DATA 127, 32, 168, 252, 169, 160
9140 DATA 202, 224, 19, 208, 172, 162
9150 DATA 0, 157, 0, 4, 157, 128
9160 DATA 4, 157, 0, 5, 157, 128
```



```
9170 DATA 5, 157, 0, 6, 157, 128
9180 DATA 6, 157, 0, 7, 157, 128
9190 DATA 7, 157, 40, 4, 157, 168
9200 DATA 4, 157, 40, 5, 157, 168
9210 DATA 5, 157, 40, 6, 157, 168
9220 DATA 6, 157, 40, 7, 157, 168
9230 DATA 7, 157, 80, 4, 157, 208
9240 DATA 4, 157, 80, 5, 157, 208
9250 DATA 5, 157, 80, 6, 157, 208
9260 DATA 6, 157, 80, 7, 157, 208
9270 DATA 7, 169, 127, 32, 168, 252
9280 DATA 169, 160, 232, 224, 20, 208
9290 DATA 172, 76, 3, 224, 96
```

Program 1-16.

```
                    10          ORG  $8000
                    11          OBJ  $8000
                    12
                    13
8000: A9 A0         14          LDA  #$A0
8002: A2 27         15          LDX  #$27
8004: 9D 00 04 16   START1      STA  $400,X
8007: 9D 80 04 17               STA  $480,X
800A: 9D 00 05 18               STA  $500,X
800D: 9D 80 05 19               STA  $580,X
8010: 9D 00 06 20               STA  $600,X
8013: 9D 80 06 21               STA  $680,X
8016: 9D 00 07 22               STA  $700,X
8019: 9D 80 07 23               STA  $780,X
801C: 9D 28 04 24               STA  $428,X
801F: 9D A8 04 25               STA  $4A8,X
8022: 9D 28 05 26               STA  $528,X
8025: 9D A8 05 27               STA  $5A8,X
8028: 9D 28 06 28               STA  $628,X
802B: 9D A8 06 29               STA  $6A8,X
802E: 9D 28 07 30               STA  $728,X
8031: 9D A8 07 31               STA  $7A8,X
8034: 9D 50 04 32               STA  $450,X
8037: 9D D0 04 33               STA  $4D0,X
```

```
803A: 9D 50 05 34          STA  $550,X
803D: 9D D0 05 35          STA  $5D0,X
8040: 9D 50 06 36          STA  $650,X
8043: 9D D0 06 37          STA  $6D0,X
8046: 9D 50 07 38          STA  $750,X
8049: 9D D0 07 39          STA  $7D0,X
804C: A9 7F    40          LDA  #$7F
804E: 20 A8 FC 41          JSR  $FCA8
8051: A9 A0    42          LDA  #$A0
8053: CA       43          DEX
8054: E0 13    44          CPX  #$13
8056: D0 AC    45          BNE  START1
8058: A2 00    46          LDX  #$0
805A: 9D 00 04 47  START5  STA  $400,X
805D: 9D 80 04 48          STA  $480,X
8060: 9D 00 05 49          STA  $500,X
8063: 9D 80 05 50          STA  $580,X
8066: 9D 00 06 51          STA  $600,X
8069: 9D 80 06 52          STA  $680,X
806C: 9D 00 07 53          STA  $700,X
806F: 9D 80 07 54          STA  $780,X
8072: 9D 28 04 55          STA  $428,X
8075: 9D A8 04 56          STA  $4A8,X
8078: 9D 28 05 57          STA  $528,X
807B: 9D A8 05 58          STA  $5A8,X
807E: 9D 28 06 59          STA  $628,X
8081: 9D A8 06 60          STA  $6A8,X
8084: 9D 28 07 61          STA  $728,X
8087: 9D A8 07 62          STA  $7A8,X
808A: 9D 50 04 63          STA  $450,X
808D: 9D D0 04 64          STA  $4D0,X
8090: 9D 50 05 65          STA  $550,X
8093: 9D D0 05 66          STA  $5D0,X
8096: 9D 50 06 67          STA  $650,X
8099: 9D D0 06 68          STA  $6D0,X
809C: 9D 50 07 69          STA  $750,X
809F: 9D D0 07 70          STA  $7D0,X
80A2: A9 7F    71          LDA  #$7F
80A4: 20 A8 FC 72          JSR  $FCA8
80A7: A9 A0    73          LDA  #$A0
```



```
80A9: E8       74          INX
80AA: E0 14    75          CPX  #$14
80AC: D0 AC    76          BNE  START5
80AE: 4C 03 E0 77          JMP  $E003
80B1: 60       78          RTS
```

To use either of the two programs, put a lot of text on the screen, run the program, and watch the screen clear.

Summary

This chapter represents a conceptual beginning to understanding graphics. Since the text screen is the least difficult to manipulate, it serves as a good beginning to understanding the concept of placing information on the screen. By arranging inverse and normal blocks of light on the screen, it's possible to produce a form of low-resolution graphics while maintaining all the text screen capabilities. More important, though, is learning the concept of how the screen is mapped to memory.

Once the screen memory is understood, it's possible to better understand the concept of screen addressing. The screen serves as a place where information can be stored. Since information can be stored there, it can be changed and manipulated (as you saw in several program examples).

The crucial element is the fundamental simplicity of what's happening. By building programs around these fundamental concepts, you can do a great deal in manipulating text and graphics.

Chapter 2

Fundamentals of Animation

This chapter continues to use text characters to explain concepts you'll apply to various graphics. For the moment, let's keep things simple.

The Illusion of Movement

Movies are optical illusions, and so is animation on your computer. Both involve showing a sequence of still photographs so rapidly that your eyes are tricked into believing you see *movement*, not a series of still pictures. With animation, you use this sequence:

- Put figure on the screen
- Erase figure
- Place figure in different location
- Erase figure
- And so on . . .

You've already done some of this—in chapter 1, you created a moving effect by changing spaces from normal to inverse and back again.

Horizontal Movement

The first thing we'll do is move a character horizontally. Let's move it across the top of the screen using VTAB and HTAB to control placement.

Program 2-1.

```
10 TEXT : HOME
20 FOR X = 1 TO 40
30 VTAB 1: REM ROW
```

```
 40 HTAB X: REM COLUMN
 50 PRINT "*": REM CHARACTER
 60 FOR PAUSE = 1 TO 20: REM SPEED CONTROL
 70 NEXT PAUSE
 80 VTAB 1: HTAB X
 90 PRINT " "
100 NEXT X
```

By changing the value of the loop in lines 60 and 70, you can increase or decrease the speed of the asterisk. To make a "trail" behind your moving object, put something other than a space after it. For example, change line 90 so that a period (.) is within the quotation marks instead of a space.

Besides using HTAB and VTAB, you can use the screen addresses to create the illusion of movement. The addresses in the top row of your screen range from 1024 to 1063 (1024 + 39). By alternating the value 170 (an asterisk) with 160 (a space), you can do the same thing.

Program 2-2.

```
10 TEXT : HOME
20 FOR X = 0 TO 39
30 POKE 1024 + X,170
40 FOR PAUSE = 1 TO 20
50 NEXT PAUSE
60 POKE 1024 + X,160
70 NEXT X
```

The second program took fewer lines than the first. But since the screen addresses are not consistently sequential from row to row, the first method of horizontal movement is easier and quicker to figure out when using vertical movement. With vertical movement, POKEing addresses is more difficult.

Vertical Movement

Vertical movement with HTAB and VTAB is essentially the same as with horizontal movement except that the maximum VTAB is 24 instead of 40. By changing lines 20–40 and line 80 in the horizontal movement program, you can create vertical movement.

Program 2-3.

```
 10 TEXT : HOME
 20 FOR X = 1 TO 24
 30 HTAB 1: REM ROW
 40 VTAB X: REM COLUMN
 50 PRINT "*": REM CHARACTER
 60 FOR PAUSE = 1 TO 20: REM SPEED CONTROL
 70 NEXT PAUSE
 80 HTAB 1: VTAB X
 90 PRINT " "
100 NEXT X
```

When you run the program, the asterik will appear to bounce. That's a problem with vertical movement caused by scrolling. By placing a semicolon before the colon in line 50 and at the end of line 90, it works without the bounce.

It takes a little more planning to move vertically through the screen memory addresses, but it can be done. The first two lines in the short program below set up a sequential arrangement for vertical movement. When you run the program, notice how the cursor appears to bounce back to the top of the screen. That's because it never left the bottom of the screen. If you add a line to print TEXT, it will be at the bottom of the screen.

Program 2-4.

```
10 TEXT : HOME
20 FOR A = 1024 TO 1104 STEP 40
30 FOR X = A TO 2039 STEP 128
40 POKE X,170
50 FOR PAUSE = 1 TO 20
60 NEXT PAUSE
70 POKE X,160
80 NEXT X
90 NEXT A
```

You may think that it's impractical to conduct movement in any place other than the left column when POKEing the screen. However, by using an *offset* from each address, you can place it in any column you want.

Program 2-5.

```
 10 TEXT : HOME
 20 INPUT "OFFSET (0-39) ";OF
 30 FOR A = 1024 TO 1104 STEP 40
 40 FOR X = A TO 2039 STEP 128
 50 POKE X + OF,170
 60 FOR PAUSE = 1 TO 20
 70 NEXT PAUSE
 80 POKE X + OF,160
 90 NEXT X
100 NEXT A
```

Diagonal Movement

Now that you've seen how to show horizontal and vertical movement, take a look at moving something diagonally.

With HTAB and VTAB it's easy, since all you need to do is simultaneously change each value of the *x* and *y* position. For instance, the following program traces a diagonal path from the top left corner to the bottom of the screen at its midpoint.

Program 2-6.

```
 10 TEXT : HOME
 20 FOR XY = 1 TO 22
 30 HTAB XY
 40 VTAB XY
 50 INVERSE
 60 PRINT CHR$ (32)
 70 FOR PAUSE = 1 TO 100
 80 NEXT PAUSE
 90 NORMAL
100 HTAB XY
110 VTAB XY
120 PRINT CHR$ (32)
130 NEXT
```

To bounce the ball to the upper right corner, add the following lines to the program above:

```
140 FOR X = 20 TO 40
150 Y = (41 - X)
160 HTAB X
```



```
170 VTAB Y
180 INVERSE
190 PRINT CHR$ (32)
200 FOR PAUSE = 1 TO 100
210 NEXT PAUSE
220 NORMAL
230 HTAB X
240 VTAB Y
250 PRINT CHR$ (32)
260 NEXT X
```

Notice in line 150 how the value for *y* (vertical position) is calculated. As the value of *x* increases, the value of *y* decreases. See if you can write a program that moves the ball in a diagonal path opposite from the program above. (Hint—use a FOR-NEXT loop for *x* with a STEP −1.)

Moving with DATA and ARRAY Tables

Developing algorithms which trace a diagonal line through screen memory can get pretty complicated. If the memory were consecutive it would be as easy as using HTAB and VTAB, but, since it's not, now is a good time to introduce another method of cruising through memory—a data table.

What you'll do is trace a path through memory using the screen memory grid from Chapter 1. Make a copy of the grid (or get some graph paper), and draw a series of dots representing the path you want to follow. Let's start with a short diagonal path and then see how to make it go somewhere.

Program 2-7.

```
 10 TEXT : HOME
 20 FOR XY = 1 TO 9
 30 READ D
 40 POKE D,32
 50 FOR PAUSE = 1 TO 100
 60 NEXT PAUSE
 70 POKE D,160
 80 NEXT XY
100 REM ************
```

Table 2-1. Address Data Table

1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043
1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171
1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299
1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427
1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551 1552 1553 1554 1555
1664 1665 1666 1667 1668 1669 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679 1680 1681 1682 1683
1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811
1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939
1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083
1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211
1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339
1448 1449 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467
1576 1577 1578 1579 1580 1581 1582 1583 1584 1585 1586 1587 1588 1589 1590 1591 1592 1593 1594 1595
1704 1705 1706 1707 1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723
1832 1833 1834 1835 1836 1837 1838 1839 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851
1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979
1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123
1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251
1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379
1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507
1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631 1632 1633 1634 1635
1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 1762 1763
1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 1890 1891
2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019

```
110 REM DIAGONAL DATA
120 REM *************
130 DATA 1024,1153,1283
140 DATA 1411,1540,1669
150 DATA 1798,1071,1200
```

Table 2-1 arranges the memory addresses sequentially from left to right, top to bottom. You can pick any sequence you want and put it in DATA statements. You can then use the screen memory without having to calculate the positions.

Putting all of the data you need in DATA statements will give you a bad case of "hacker's cramp," even though it provides a nice roadmap of what you need. Instead of using DATA statements to animate characters, make an array—it involves a lot less work. Once the data are placed sequentially in the array, you can use the array as your table. Since the addresses are now in sequential order, it makes it easier to program your movement.

The following program first places the screen addresses in a sequence from left to right and top to bottom (lines 10–130). It's a useful subroutine, and you might want to save it separately. The second part of the program moves the ball along a diagonal trail.

1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063
1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191
1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319
1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441 1442 1443 1444 1445 1446 1447
1556 1557 1558 1559 1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 1570 1571 1572 1573 1574 1575
1684 1685 1686 1687 1688 1689 1690 1691 1692 1693 1694 1695 1696 1697 1698 1699 1700 1701 1702 1703
1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823 1824 1825 1826 1827 1828 1829 1830 1831
1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959
1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103
1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231
1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359
1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487
1596 1597 1598 1599 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615
1724 1725 1726 1727 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743
1852 1853 1854 1855 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871
1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999
1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143
1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271
1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399
1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527
1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649 1650 1651 1652 1653 1654 1655
1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781 1782 1783
1892 1893 1894 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911
2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039

The Y loop establishes the vertical position and the X loop increments the horizontal position. Now that the addresses are sequential, each row is a jump of 40, and, each column, a jump of 1. Remember, though, you're going through the values of the array, not the addresses directly. (By the way, when you run this program, there will be a noticeable pause while the addresses are loaded into the array.)

Program 2-8.

```
 10 REM ***************
 20 REM PLACE SEQUENTIAL
 30 REM TABLE IN ARRAY
 40 REM ***************
 50 TEXT : HOME
 60 DIM V(960)
 70 FOR PA = 1024 TO 1104 STEP 40
 80 FOR X = PA TO 2039 STEP 128
 90 FOR SCREEN = 0 TO 39
100 COUNT = COUNT + 1
110 V(COUNT) = X + SCREEN
```

```
120 NEXT SCREEN
130 NEXT X: NEXT PA
140 REM **********
150 REM PLACE ON SCREEN
160 REM USING ARRAY DATA
170 REM **********
180 X = 0
190 FOR Y = 1 TO 940 STEP 40
200 X = X + 1
210 POKE V(Y + X),32
220 FOR PAUSE = 1 TO 100
230 NEXT PAUSE
240 POKE V(Y + X),160
250 NEXT Y
```

Now that you have mastered some of the basics, let's take a short detour to the 80-column screen before returning to animation.

80 Columns—Two Memory Banks

Your Apple IIGS's 80-column screen works exactly like the 40-column screen. For example, the following program runs a character across your 80-column screen. (If you're not currently in 80-column mode, just press the Esc and 8 keys at the same time.)

Program 2-9.

```
10 TEXT : HOME
20 FOR X = 1 TO 80
30 HTAB X: VTAB 20
40 PRINT ">"
50 FOR PAUSE = 1 TO 50
60 NEXT PAUSE
70 HTAB X: VTAB 20
80 PRINT CHR$ (32)
90 NEXT X
```

Earlier, you learned how to rearrange screen memory and move through it sequentially. Even though 80-column mode has the same screen addresses as 40-column mode, there's a catch to

using 80 columns. *Only every other column can be reached from BASIC.* This next program will demonstrate this. Run the program in both 40- and 80-column mode.

Program 2-10

```
10 TEXT : HOME
20 FOR X=1 TO 40
30 POKE 1023+X,65+128
40 NEXT X
```

In 40-column and 80-columns, the letter *A* was placed across the top of the screen, but in the 80-column mode only every other column was used. How do you fill the alternate columns?

Your Apple IIGS has another *bank* of memory which uses the same addresses for the alternating rows. When programming in BASIC, you're automatically in bank 0. The problems are getting to bank 1 and getting the memory for the alternate addresses. In machine language programming, this isn't a problem, but from BASIC, it's tough. You have to POKE in a short machine language routine to access the other memory bank.

Bank-Switching in Machine Language

Let's examine a short machine language program to see what's going on and then see how to access the *long jump* routine from BASIC. (*Long jump* refers to jumping from one bank to another.)

```
00/8000: 20 58 FC     JSR  FC58
00/8003: A2 28        LDX  #28
00/8005: A9 C1        LDA  #C1
00/8007: 9D FF 03     STA  03FF,X
00/800A: 9F FF 03 01  STA  0103FF,X
00/800E: CA           DEX
00/800F: D0 F6        BNE  8007 {-0A}
00/8011: 60           RTS
```

This routine fills the first row of an 80-column screen with the letter *A*. It places the value for the letter *A* ($C1) in each address of the first row using a loop. The loop first puts $C1 in address $0427 (1063, the address of the upper right corner of the 80-column screen), then in each address to the left.

Notice the STA instructions in the fourth and fifth lines. (*STA* stands for *STore the Accumulator.*) The first STA instruction has a machine language opcode of $9D (the first value after the *00/8007:*), and it works much like a POKE statement, except it stores a value in an address *indexed* by *x*. The first time through the loop, the value goes into address $03FF + $28, or $0427 (1063). The last time through the loop, it goes into address $03FF + $01, or $0400 (1024).

However, there's no BASIC equivalent to the second STA, which has an opcode of $9F. This version of STA stores the same value in the same address, except this time the address is in bank 1 instead of bank 0. This is the *long jump* mentioned earlier. There's no *long jump* POKE equivalent in BASIC. That means you must figure out how to get a value which can be used from within a BASIC program. Let's create a machine language routine to make the jump from bank 0 to bank 1. This routine will allow us to place characters in consecutive locations on the screen in 80-column mode.

First, translate the *long jump* STA machine language opcode into decimal. The STA which places values in addresses directly, not in addresses indexed from the X register, has an opcode of $8F (143). The code for storing a character to the screen will then look like this:

8F = Long jump STA - Works like POKE to other banks
FF = Low byte of $3FF (address to POKE bank number)
03 = High byte of $3FF (address to POKE bank number)
01 = Bank number

In decimal, then, the values to POKE are

143
255
3
1

There's some empty memory beginning at address $300 (768); let's use it for writing the routine.

Fill the top line of the screen with inverse spaces. To do this in the machine language routine, you must load into the *accumulator* the value for an inverse space. The value for an inverse space is 32,

and the opcode for loading the accumulator (the instruction is LDA, for *LoaD the Accumulator*) is

LDA = 169

In hexadecimal, 169 is $A9.

Since the same range of addresses are used in Bank 00 and Bank 01, you can use the same array table as for the 40-column mode. The following program is the result.

Program 2-11.

```
 10 REM ***************
 20 REM PLACE SEQUENTIAL
 30 REM TABLE IN ARRAY
 40 REM ***************
 50 TEXT : HOME
 60 DIM V(960)
 70 FOR PA = 1024 TO 1104 STEP 40
 80 FOR X = PA TO 2039 STEP 128
 90 FOR SCREEN = 0 TO 39
100 COUNT = COUNT + 1
110 V(COUNT) = X + SCREEN
120 NEXT SCREEN
130 NEXT X: NEXT PA
140 REM ***************
150 REM PLACE ON SCREEN
160 REM USING ARRAY DATA
170 REM ***************
180 FOR XY = 1 TO 40
190 POKE V(XY),32
200 N = V(XY)
210 GOSUB 270
220 NEXT XY
230 END
240 REM ******************
250 REM CONVERT TO 2 BYTE #
260 REM ******************
270 LB = N - INT (N / 256) * 256
280 HB = INT (N / 256)
```

```
290 REM ***************
300 REM MACHINE LANGUAGE
310 REM ***************
320 POKE 768,169: REM LDA
330 POKE 769,32: REM INVERSE SPACE
340 POKE 770,143: REM LONG STA
350 POKE 771,LB: REM LOWBYTE
360 POKE 772,HB: REM HIGHBYTE
370 POKE 773,1: REM BANK01
380 POKE 774,96: REM RTS
390 CALL 768: REM EXECUTE ROUTINE
400 RETURN
```

The program actually rewrites and then executes the short machine language routine every time it goes through the loop.

Now that you can draw across banks, let's change the program to animate across banks.

Alter the 40-column routine for diagonal movement through memory so that it traces the same route in 80 columns.

Program 2-12.

```
 10 REM ***************
 20 REM PLACE SEQUENTIAL
 30 REM TABLE IN ARRAY
 40 REM ***************
 50 TEXT : HOME
 60 DIM V(960)
 70 FOR PA = 1024 TO 1104 STEP 40
 80 FOR X = PA TO 2039 STEP 128
 90 FOR SCREEN = 0 TO 39
100 COUNT = COUNT + 1
110 V(COUNT) = X + SCREEN
120 NEXT SCREEN
130 NEXT X: NEXT PA
200 REM ***************
210 REM PLACE ON SCREEN
220 REM USING ARRAY DATA
230 REM ***************
240 X = 0
```



```
250 FOR Y = 1 TO 940 STEP 40
260 X = X + 1
270 POKE V(Y + X),32
280 FOR PAUSE = 1 TO 100
290 NEXT PAUSE
300 POKE V(Y + X),160
310 N = V(Y + X)
320 GOSUB 430
330 FOR PAUSE = 1 TO 10
340 NEXT PAUSE
350 GOSUB 730
360 NEXT Y
370 END
400 REM ******************
410 REM CONVERT TO 2 BYTE #
420 REM ******************
430 LB = N - INT (N / 256) * 256
440 HB = INT (N / 256)
500 REM ***************
510 REM MACHINE LANGUAGE
520 REM ***************
530 POKE 768,169: REM LDA
540 POKE 769,32: REM INVERSE SPACE
550 POKE 770,143: REM LONG STA
560 POKE 771,LB: REM LOWBYTE
570 POKE 772,HB: REM HIGHBYTE
580 POKE 773,1: REM BANK01
590 POKE 774,96: REM RTS
600 CALL 768: REM EXECUTE ROUTINE
610 RETURN
700 REM ************
710 REM NORMAL SPACE
720 REM ************
730 POKE 768,169: REM LDA
740 POKE 769,160: REM NORMAL SPACE
750 POKE 770,143: REM LONG STA
760 POKE 771,LB: REM LOWBYTE
770 POKE 772,HB: REM HIGHBYTE
```

```
780 POKE 773,1: REM BANK01
790 POKE 774,96: REM RTS
800 CALL 768: REM EXECUTE ROUTINE
810 RETURN
```

Animation Applications

One application of this animation is in games. In some computer games, you need a way for the machine to animate characters to a target—usually the player's character. There are many sophisticated algorithms for doing this, but let's start with the basics. To get started, make a guided missile on the 40-column screen. This little program will have two moving characters:

- A target represented by a right arrow (>).
- A missile represented by a carat (^).

The missile will track the target with a radar that scans the path of the target. It will always move toward the target based on the information it gets from scanning the addresses in the first row (1024–1063) along which the target is moving. Then it compares the position of the target with the position of the missile. It does this by subtracting the variable F, which is the position of the missile + 2. (It 'leads' the target with the +2.) If the difference between the missile's horizontal positon and the target is positive, the program knows the target is to the left and subtracts from its horizontal value (MH). If the difference is positive, the program adds to the horizontal position of the missile. This is a good example of how to use the SGN function in Applesoft BASIC. The vertical movement is constantly decreased for the missile.

Program 2-13.

```
10 GOSUB 270
20 TEXT : HOME
30 FOR T = 1 TO 40: HTAB T: VTAB 1: PRINT T$;
40 HTAB MH: REM MISSILE HORIZONTAL POS
50 VTAB MV: REM MISSILE VERTICAL POS
60 PRINT M$;: REM PUT ON SCREEN
70 REM *****
```



```
 80 REM RADAR
 90 REM *****
100 FOR X = 0 TO 39
110 P = PEEK (1024 + X): REM SEEK TARGET
120 IF P < > 160 THEN F = X + 2
130 NEXT X
140 REM *************
150 REM EVALUATE DATA
160 REM *************
170 DH = MH - F:DV = MV - VT
180 IF SGN (DH) = 1 THEN MH = MH - 1
190 IF SGN (DH) = - 1 THEN MH = MH + 1
200 MV = MV - 1: IF MV < 1 THEN END
210 HTAB T: VTAB 1: PRINT S$
220 NEXT T
230 END
240 REM ******
250 REM SET UP
260 REM ******
270 MH = 40:MV = 24:VT = 1
280 T$ = ">": REM TARGET
290 S$ = " ": REM SPACE
300 M$ = "^": REM MISSILE
310 RETURN
```

That program was fairly simple, and the target was moving in a single direction. To provide a more interesting example of the same principle, let's provide random horizontal movement for the target. This time, change it just a bit.

This next program puts all of the horizontal addresses in an integer array and then scans it. It also includes a more interesting-looking target and missile. The missile's radar is far more deadly. It never misses, no matter how erratic a path the random number generator devises. (And in the 80-column mode, it never hits.)

Program 2-14.

```
10 DIM C%(40)
20 FOR X = 0 TO 39:C%(X) = 1024 + X: NEXT
30 TEXT : HOME
```

```
 40 AV = 1:AH = 10:VM = 20:HM = 20
 50 FH = INT ( RND (1) * (2) + 1)
 60 IF FH = 1 THEN AH = AH + 3
 70 IF FH = 2 THEN AH = AH - 3
 80 IF AH < 1 THEN AH = 1
 90 IF AH > 38 THEN AH = 38
100 VTAB AV: HTAB AH: PRINT "=0="
110 FOR X = 0 TO 39: IF PEEK (C%(X)) < > 160 THEN F = X
120 NEXT X
130 VTAB VM + 1: HTAB FF: PRINT CHR$ (32)
140 VTAB VM: HTAB F: PRINT "#":FF = F
150 VM = VM - 1: IF VM < 1 THEN END
160 HTAB AH: VTAB AV: PRINT SPC( 3)
170 GOTO 50
```

Now that you can bounce a target around and program the computer to track it, let's create an animated shoot-'em-up game where an "alien" moves around the screen. Let's spice it up by adding a time element and giving the target both downward and random horizontal movement. The object of the game is to score as many hits as possible either before time runs out or before the alien lands. This simple game is written for 80 columns. (If you want to display it in 40 columns, make the indicated changes, and change the placement of the score box in line 360.)

Other Game Features

Pay close attention to the following features in this program that haven't yet been examined in detail:

- The read-keyboard subroutine
- How the variable P is used in moving and firing
- The hit and fire subroutines

Basically, the program reads the keyboard and stores the value of the last key pressed in the variable P. That value is then used in the move/fire subroutine. The movement values in P are evaluated in terms of the ASCII values of the left and right arrow keys on your keyboard. When using a joystick, paddles, or mouse, use the same concept.

Each time the player fires, the alien moves down. That's why the AV (Alien Vertical) variable is incremented in the FIRE! subroutine. If a hit is scored, a new alien is placed at the top of the screen. That's why the vertical movement of the alien is also in the HIT subroutine.

Program 2-15.

```
 10 TIME = 250:PTS = 0
 20 TEXT : HOME :FLAG = 0
 30 H = 15:V = 22:AV = 1:AH = 40
 40 REM ************
 50 REM READ KEYBOARD
 60 REM ************
 70 P = 0: IF PEEK ( - 16384) > 127 THEN P = PEEK ( - 16384) - 128:
    POKE - 16368,0
 80 REM ****************************
 90 REM RANDOM ALIEN HORIZONTAL MOVE
100 REM ****************************
110 FH = INT ( RND (1) * (2) + 1)
120 IF FH = 1 THEN AH = AH + 3
130 IF FH = 2 THEN AH = AH - 3
140 IF AH < 2 THEN AH = 2
150 IF AH > 78 THEN AH = 78: REM CHANGE TO 38 FOR 40 COL
160 VTAB AV: HTAB AH: PRINT "=0="
170 REM ******************
180 REM MOVE OR FIRE PLAYER
190 REM ******************
200 IF P = 32 THEN GOSUB 400
210 IF P = 8 THEN H = H - 1
220 IF P = 21 THEN H = H + 1
230 IF H > 79 THEN H = 79: REM 39 FOR 40 COL
240 IF H < 1 THEN H = 1
250 IF V > 23 THEN V = 23
260 IF V < 1 THEN V = 1
270 HTAB H: VTAB V: PRINT "^"
280 FOR HOLD = 1 TO 100: NEXT HOLD
290 HTAB AH: VTAB AV: PRINT " "
300 HTAB H: VTAB V: PRINT " "
```

```
310 IF FLAG = 1 THEN 20
320 REM **********************
330 REM DISPLAY TIME AND SCORE
340 REM **********************
350 TIME = TIME - 1: IF TIME = 0 THEN HOME : PRINT "FINAL
    SCORE =";PTS: END
360 HTAB 1: VTAB 23: INVERSE : PRINT " TIME = ";TIME;
    SPC( 1): HTAB 70: VTAB 23: PRINT " SCORE = ";PTS: NORMAL
370 GOTO 70
400 REM *****
410 REM FIRE!
420 REM *****
430 FOR F = 22 TO 1 STEP - 1
440 HTAB H: VTAB F: PRINT "*"
450 HIT = (H = (AH + 1)) AND (F = AV)
460 IF HIT THEN FLAG = 1: GOTO 600
470 HTAB H: VTAB F: PRINT "  ": NEXT
480 HTAB AH: VTAB AV: PRINT "     "
490 AV = AV + 1: IF AV = 22 THEN PRINT "THEY GOT YOU!": END
500 RETURN
600 REM **********
610 REM *** HIT ***
620 REM **********
630 HTAB AH: VTAB AV: PRINT "BOOM!"
640 PRINT CHR$ (7)
650 FOR PAUSE = 1 TO 500: NEXT PAUSE
660 AV = 1: HOME
670 PTS = PTS + 1
680 GOTO 500
```

Summary

Using text characters, you've simulated graphics applications involving movement or animation. There are two ways to move things on the screen: One is to use BASIC VTAB and HTAB statements; the other is to POKE directly to the screen using sequentially arranged addresses. Since the screen address space in low resolution is almost identical to the text screen, it's possible to use many of the same routines used in text. Likewise, while not all of the BASIC statements are the same in graphics and text, you can use the same principles of screen placement with both.

Chapter 3
Low-Resolution Graphics

Low-resolution graphics are similar to text graphics, though the results are a bit different. Instead of seeing text characters on the screen, you'll see small blocks of colors.

The low-resolution graphics screen is divided into a matrix that's 40 × 48 blocks (instead of one that's 40 × 24 blocks, as in text). Except for its dimensions, the matrix is the same as that used by text.

In this chapter, you'll first look at the BASIC statements regulating low-resolution graphics, then see some programs which use such graphics. A lot of attention will be paid to colors, since they are what make low-resolution graphics special. And you'll see how low-resolution graphics work in memory.

Color

There are 16 low-resolution colors numbered from 0 to 15.

0 Black	8 Brown
1 Magenta	9 Orange
2 Dark blue	10 Gray
3 Purple	11 Pink
4 Dark green	12 Green
5 Gray	13 Yellow
6 Blue	14 Aqua
7 Light blue	15 White

An unusual thing happens when you go into low-resolution graphics. No matter what background color the screen had been before, it turns to black when it goes into the graphics mode.

First, let's take a look at the Applesoft statements you can use with low-resolution graphics.

Command	Function
GR	Turn on low-resolution graphics
COLOR=*x*	Set color (*x* = 0–15)
PLOT *x,y*	Place block at specified position. Horizontal (0–39), Y=Row/Vertical (0–47)
HLIN	Horizontal line from X1 to X2 at Y (X=0–39)
VLIN	Vertical line from Y1 to Y2 at X (Y=0–47)
SCRN(*x,y*)	Returns the color code of position *x,y* in color values 0–15.

A number of memory addresses can be POKEd with 0 to turn special features on or off.

Address	Switch Function
49232	Turn on graphics
49234	Full text or full graphics mode
49235	Mixed text and graphics mode
49236	Display page 1
49237	Display page 2
49238	Low-resolution graphics

WAIT −16384,128
Stops until key is pressed (no cursor is displayed)

Your first low-resolution BASIC program shows all 16 colors. Using the HLIN statement and a variable for the colors, the program scrolls through the colors.

Program 3-1.

```
 10 K = 5
 20 GR
 30 FOR X = 0 TO 15
 40 COLOR= X
 50 HLIN 0,39 AT K
 60 K = K + 2
 70 NEXT
 80 GET A$: PRINT A$
 90 TEXT : HOME
100 LIST
```

The first thing to notice is that the background turned black. If you had the default colors (blue background and border), you'll still see a blue border, but the background's changed. To get out of

the graphics mode, just type **TEXT**. (Program 3-1 does that for you and LISTs the program as well. Lines 80 and 90 are a good routine to stick on the end of your graphics programs while they're being developed.) On return to the TEXT mode, the background is restored to the default colors.

If the colors in the bars don't seem right, adjust your television or monitor so that the colors correspond to those in the list above. If they're never quite right, it might be a problem with the TV set or monitor, or even with your IIGS.

Low-Resolution Graphics in BASIC

Now let's see what you can do with low-resolution graphics from BASIC. Start off with HLIN and VLIN to see the limits of their parameters.

Program 3-2.

```
10 GR
20 COLOR= 15
30 HLIN 0,39 AT 20
40 VLIN 0,47 AT 20
```

The first thing to notice is that you still have four lines of text at the bottom of your screen, where the cursor is waiting. You can also see where the vertical line fell out of the graphics page and spilled onto the text page. The first two chapters of this book pointed out that there was a connection between the text page and the low-resolution graphics page. Now you can see it.

In many instances, you'll want to have the full graphics page available. To use the entire graphics page, you need to "flip" the full-page graphics *soft switch* with a POKE statement. From the short list above, you can see that the soft switch for a full graphics page is address 49234. POKEing 0 into that location flips the switch, so to speak.

Add a POKE 49234,0 statement to the program, as well as a WAIT statement that freezes the graphics without the cursor or a prompt disturbing the display. Hit any key and the screen clears; you're back in text for more programming.

Program 3-3.

```
 10 HOME
 20 GR
 30 REM *************
 40 REM FULL GRAPHICS
 50 REM *************
 60 POKE 49234,0
 70 COLOR= 15
 80 HLIN 0,39 AT 20
 90 VLIN 0,47 AT 20
100 WAIT - 16384,128
110 TEXT : HOME
```

You'll notice there are now some gray lines where the text was. These are text spaces—ASCII value 160. In low-resolution graphics, 160 is the value for black over gray. We want black over black, which is ASCII value of 0. Put zeros in the addresses of the bottom four lines.

The addresses run from 1616 to 2000. A loop (lines 50–90 in the program below) can quickly POKE each address with black over black.

Program 3-4.

```
 10 GR
 20 REM ***********
 30 REM CLEAR LINES
 40 REM ***********
 50 FOR X = 1616 TO 2000 STEP 128
 60 FOR L = X TO X + 39
 70 POKE L,0
 80 NEXT L
 90 NEXT X
100 POKE 49234,0
110 COLOR= 15
120 HLIN 0,39 AT 20
130 VLIN 0,47 AT 20
140 WAIT - 16384,128
150 TEXT : HOME
```


More on screen memory locations is discussed later in this chapter, but for now just remember that the low-resolution graphics screen is like the text screen, except that it generates stacked color blocks instead of text.

PLOT It

Let's look at the PLOT statement. This next programs starts by drawing an *X* on the screen with PLOT statements. (The program uses yellow, but you may change it to another color if you want.)

Program 3-5.

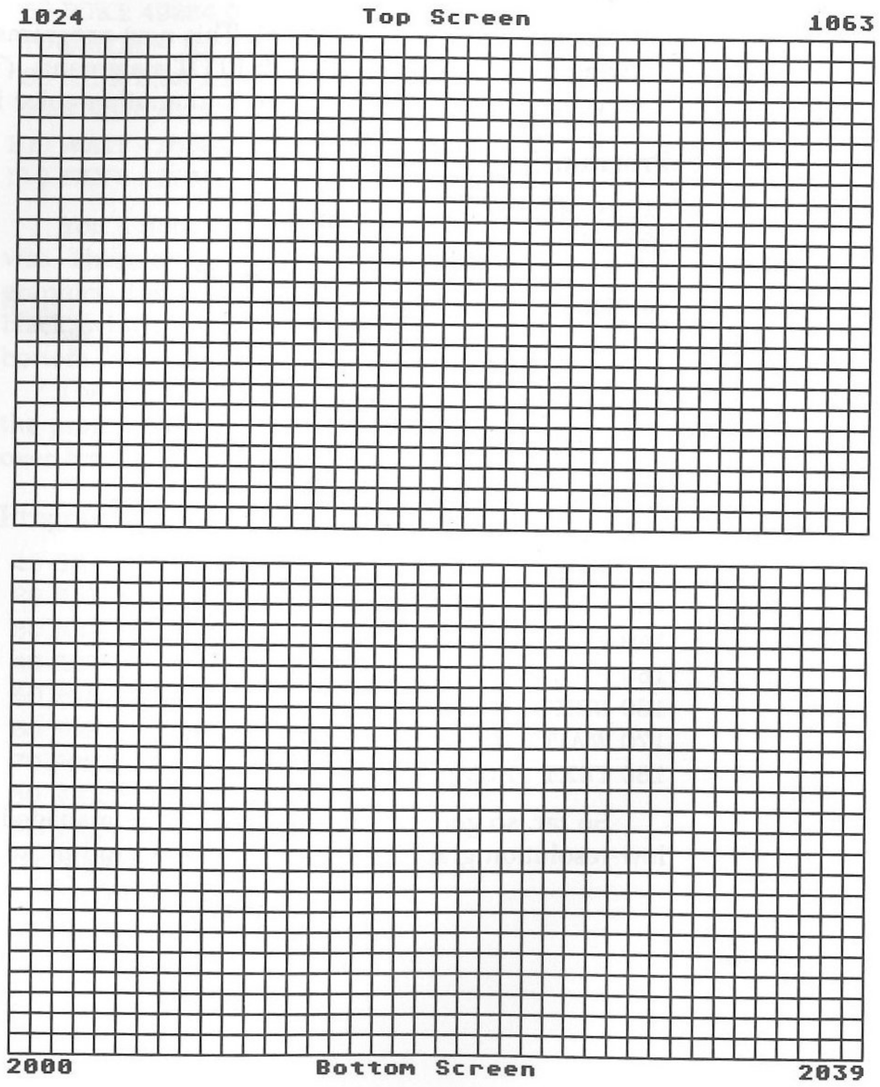
```
 10 HOME
 20 GR
 30 FOR X = 1616 TO 2000 STEP 128
 40 FOR L = X TO X + 39
 50 POKE L,0
 60 NEXT L
 70 NEXT X
 80 POKE 49234,0
 90 COLOR= 13
100 REM ***********
110 REM PLOT POINTS
120 REM ***********
130 FOR X = 0 TO 39
140 PLOT X,X
150 PLOT 39 - X,X
160 NEXT
170 WAIT - 16384,128
180 TEXT : HOME : LIST
```

So far, so good. In a short time, we've managed to use every low-resolution graphics BASIC statement, and most of the POKEs.

The Low-Resolution Graphics Screen

Think of the low-resolution graphics screen as two text screens stacked on top of one another. (See Figure 3-1.)

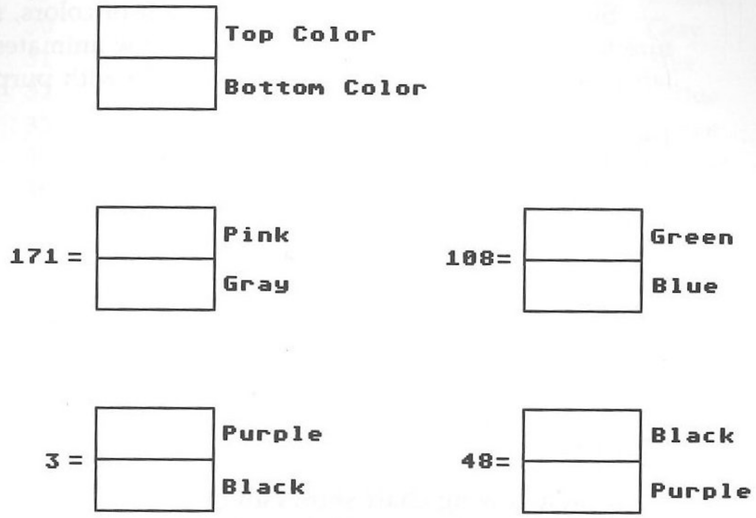
Figure 3-1. Lo-Res Screen



Even though there are *twice* as many vertical positions (48 rows instead of 24) as on the text screen, there are the *same* number of address spaces (1024 to 2039). What's going on?

Each block in the above grid is actually *one half of a vertical pair.* On the low-resolution screen, the upper left corner is address 1024, *as is the block directly below it.*

Figure 3-2. Top Color/Bottom Color



Every block is assigned a color based on the COLOR= statement from BASIC. When you plot a color block, you actually plot half of it black and the other half the assigned color.

Let's take a look at some examples to demonstrate the point. You'll first use pink and gray to plot a block in the upper left corner. Pink will be on top while gray will be on the bottom.

Program 3-6.

```
10 GR
20 COLOR= 11
30 PLOT 0,0
40 COLOR= 10
50 PLOT 0,1
60 P = PEEK (1024)
70 PRINT P
```

The way to check that each byte controls two blocks of color is to take the value the program returned, and POKE it into address 1024 with no PLOT statement at all.

```
10 GR
20 POKE 1024,171
```

Try the same thing with green and blue. First plot the two colors with the PLOT statement, then POKE them in with a single POKE statement.

Since each address has a combination of colors, use that feature to have a little fun. The program below animates the upper left corner by alternating black over purple with purple over black.

Program 3-7.

```
10 GR
20 FOR X = 1 TO 100
30 POKE 1024,3
40 FOR PAUSE = 1 TO 10
50 NEXT PAUSE
60 POKE 1024,48
70 FOR PAUSE = 1 TO 10
80 NEXT PAUSE
90 NEXT X
```

The following chart shows all of the color combinations in low-resolution graphics. Pick the combination you want, then POKE it into a memory address between 1024 and 2039.

Value	Top	Bottom	Value	Top	Bottom
0	Black	Black	11	Pink	Black
1	Magenta	Black	12	Green	Black
2	D Blue	Black	13	Yellow	Black
3	Purple	Black	14	Aqua	Black
4	D Green	Black	15	White	Black
5	Gray	Black	16	Black	Magenta
6	Blue	Black	17	Magenta	Magenta
7	L Blue	Black	18	D Blue	Magenta
8	Brown	Black	19	Purple	Magenta
9	Orange	Black	20	D Green	Magenta
10	Gray	Black	21	Gray	Magenta

Value	Top	Bottom	Value	Top	Bottom
22	Blue	Magenta	61	Yellow	Purple
23	L Blue	Magenta	62	Aqua	Purple
24	Brown	Magenta	63	White	Purple
25	Orange	Magenta	64	Black	D Green
26	Gray	Magenta	65	Magenta	D Green
27	Pink	Magenta	66	D Blue	D Green
28	Green	Magenta	67	Purple	D Green
29	Yellow	Magenta	68	D Green	D Green
30	Aqua	Magenta	69	Gray	D Green
31	White	Magenta	70	Blue	D Green
32	Black	D Blue	71	L Blue	D Green
33	Magenta	D Blue	72	Brown	D Green
34	D Blue	D Blue	73	Orange	D Green
35	Purple	D Blue	74	Gray	D Green
36	D Green	D Blue	75	Pink	D Green
37	Gray	D Blue	76	Green	D Green
38	Blue	D Blue	77	Yellow	D Green
39	L Blue	D Blue	78	Aqua	D Green
40	Brown	D Blue	79	White	D Green
41	Orange	D Blue	80	Black	Gray
42	Gray	D Blue	81	Magenta	Gray
43	Pink	D Blue	82	D Blue	Gray
44	Green	D Blue	83	Purple	Gray
45	Yellow	D Blue	84	D Green	Gray
46	Aqua	D Blue	85	Gray	Gray
47	White	D Blue	86	Blue	Gray
48	Black	Purple	87	L Blue	Gray
49	Magenta	Purple	88	Brown	Gray
50	D Blue	Purple	89	Orange	Gray
51	Purple	Purple	90	Gray	Gray
52	D Green	Purple	91	Pink	Gray
53	Gray	Purple	92	Green	Gray
54	Blue	Purple	93	Yellow	Gray
55	L Blue	Purple	94	Aqua	Gray
56	Brown	Purple	95	White	Gray
57	Orange	Purple	96	Black	Blue
58	Gray	Purple	97	Magenta	Blue
59	Pink	Purple	98	D Blue	Blue
60	Green	Purple	99	Purple	Blue

Value	Top	Bottom
100	D Green	Blue
101	Gray	Blue
102	Blue	Blue
103	L Blue	Blue
104	Brown	Blue
105	Orange	Blue
106	Gray	Blue
107	Pink	Blue
108	Green	Blue
109	Yellow	Blue
110	Aqua	Blue
111	White	Blue
112	Black	L Blue
113	Magenta	L Blue
114	D Blue	L Blue
115	Purple	L Blue
116	D Green	L Blue
117	Gray	L Blue
118	Blue	L Blue
119	L Blue	L Blue
120	Brown	L Blue
121	Orange	L Blue
122	Gray	L Blue
123	Pink	L Blue
124	Green	L Blue
125	Yellow	L Blue
126	Aqua	L Blue
127	White	L Blue
128	Black	Brown
129	Magenta	Brown
130	D Blue	Brown
131	Purple	Brown
132	D Green	Brown
133	Gray	Brown
134	Blue	Brown
135	L Blue	Brown
136	Brown	Brown
137	Orange	Brown
138	Gray	Brown
139	Pink	Brown
140	Green	Brown
141	Yellow	Brown
142	Aqua	Brown
143	White	Brown
144	Black	Orange
145	Magenta	Orange
146	D Blue	Orange
147	Purple	Orange
148	D Green	Orange
149	Gray	Orange
150	Blue	Orange
151	L Blue	Orange
152	Brown	Orange
153	Orange	Orange
154	Gray	Orange
155	Pink	Orange
156	Green	Orange
157	Yellow	Orange
158	Aqua	Orange
159	White	Orange
160	Black	Gray
161	Magenta	Gray
162	D Blue	Gray
163	Purple	Gray
164	D Green	Gray
165	Gray	Gray
166	Blue	Gray
167	L Blue	Gray
168	Brown	Gray
169	Orange	Gray
170	Gray	Gray
171	Pink	Gray
172	Green	Gray
173	Yellow	Gray
174	Aqua	Gray
175	White	Gray
176	Black	Pink
177	Magenta	Pink

Value	Top	Bottom
178	D Blue	Pink
179	Purple	Pink
180	D Green	Pink
181	Gray	Pink
182	Blue	Pink
183	L Blue	Pink
184	Brown	Pink
185	Orange	Pink
186	Gray	Pink
187	Pink	Pink
188	Green	Pink
189	Yellow	Pink
190	Aqua	Pink
191	White	Pink
192	Black	Green
193	Magenta	Green
194	D Blue	Green
195	Purple	Green
196	D Green	Green
197	Gray	Green
198	Blue	Green
199	L Blue	Green
200	Brown	Green
201	Orange	Green
202	Gray	Green
203	Pink	Green
204	Green	Green
205	Yellow	Green
206	Aqua	Green
207	White	Green
208	Black	Yellow
209	Magenta	Yellow
210	D Blue	Yellow
211	Purple	Yellow
212	D Green	Yellow
213	Gray	Yellow
214	Blue	Yellow
215	L Blue	Yellow
216	Brown	Yellow
217	Orange	Yellow
218	Gray	Yellow
219	Pink	Yellow
220	Green	Yellow
221	Yellow	Yellow
222	Aqua	Yellow
223	White	Yellow
224	Black	Aqua
225	Magenta	Aqua
226	D Blue	Aqua
227	Purple	Aqua
228	D Green	Aqua
229	Gray	Aqua
230	Blue	Aqua
231	L Blue	Aqua
232	Brown	Aqua
233	Orange	Aqua
234	Gray	Aqua
235	Pink	Aqua
236	Green	Aqua
237	Yellow	Aqua
238	Aqua	Aqua
239	White	Aqua
240	Black	White
241	Magenta	White
242	D Blue	White
243	Purple	White
244	D Green	White
245	Gray	White
246	Blue	White
247	L Blue	White
248	Brown	White
249	Orange	White
250	Gray	White
251	Pink	White
252	Green	White
253	Yellow	White
254	Aqua	White
255	White	White

These color combinations work just as well with double-low-resolution graphics.

Experiment with some short programs of your own to see what you can draw on the screen. Later, you'll see several low-resolution graphics programs which will give you ideas for some interesting projects.

Double-Low-Resolution Graphics

If you've tried any of the graphics programs up to now in the 80-column mode, you may have found that when you go into the low-resolution graphics mode, you're still in 40 columns. It's possible to double the resolution of the graphics with a simple POKE to the double-resolution soft switch located at address 49246. By writing programs from the 80-column mode, you can double the horizontal (but not the vertical) resolution of low-resolution graphics.

Enter the 80-column mode now by pressing ESC and the 8 key at the same time.

The following program sets the soft switch and draws a line on the screen—don't forget to type this in and run it while in 80-column mode.

Program 3-8.

```
 5 HOME
10 GR
20 COLOR= 15
30 POKE 49246,0
40 HLIN 0,79 AT 24
50 POKE 49247,0
```

Within BASIC, each of these 3 commands—PLOT, HLIN, and VLIN—expects 2 numbers. The first falls within the range 0–79 and and the second falls in the range 0–47. You can also use all the colors from BASIC in this increased resolution. To turn off double-low-resolution graphics, use the soft switch at location 49247.

Let's look at what's going on behind the scenes by POKEing in a line of low-resolution points.

Since screen addresses are the same for the low-resolution graphics page and text, you can POKE a line across the top of the

screen from 1024 to 1024 + 79. (Remember to switch to 80-column mode, if you're not in it already.)

Program 3-9.

```
10 HOME
20 GR
40 POKE 49246,0
50 FOR X = 1024 TO 1024 + 79
60 POKE X,15
70 NEXT
```

You should see a checkerboard on the screen, with some white dots across the top and just above the middle. The gray squares are a mystery. Let's turn off low-resolution graphics and try it again.

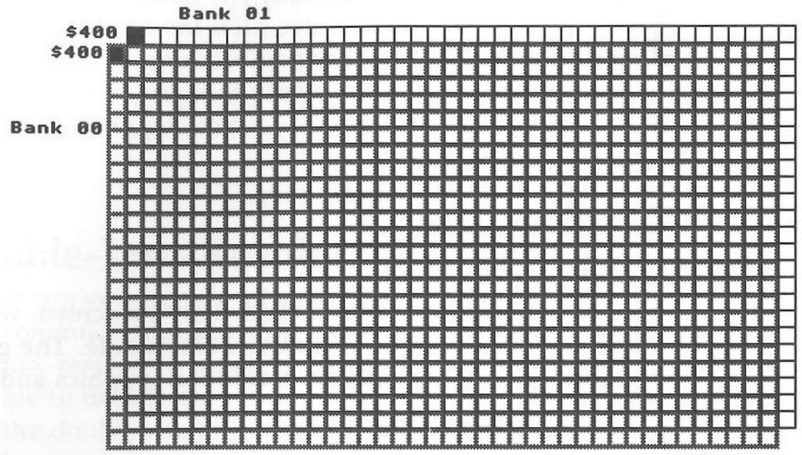
Program 3-10.

```
10 HOME
20 GR
40 POKE 49247,0: REM TURN OFF DOUBLE-RESOLUTION
50 FOR X = 1024 TO 1024 + 79
60 POKE X,15
70 NEXT
```

This time you should see two solid lines. Here's what happened: Both programs plotted 80 points, but the second could only fit 40 points in a line; that's why you saw two full lines. The first program, however, used double-resolution graphics, and, like the 80-column text, used the parallel screen in bank 1 of your IIGS. The color gray came from the value 160 ($A0), which is stored in those locations. (Remember that 160 is a blank space in text and a gray block in low-resolution graphics.) Figure 3-3 shows the relationship between the two banks in double-low-resolution graphics.

Earlier, you needed a short machine language routine to access text graphics in bank 1—with some modification, you can use that same program to POKE values into memory to create color instead of inverse spaces.

Figure 3-3. Bank 0–Bank 1



Program 3-11.

```
 10 REM ***************
 20 REM PLACE SEQUENTIAL
 30 REM TABLE IN ARRAY
 40 REM ***************
 50 GR
 60 DIM V(960)
 70 FOR PA = 1024 TO 1104 STEP 40
 80 FOR X = PA TO 2039 STEP 128
 90 FOR SCREEN = 0 TO 39
100 COUNT = COUNT + 1
110 V(COUNT) = X + SCREEN
120 NEXT SCREEN
130 NEXT X: NEXT PA
140 REM ***************
150 REM PLACE ON SCREEN
160 REM USING ARRAY DATA
170 REM ***************
180 FOR XY = 1 TO 40
```



```
190 POKE V(XY),15
200 N = V(XY)
210 GOSUB 240
220 NEXT XY
230 END
240 REM ***************
250 REM CONVERT TO 2 BYTE #
260 REM ***************
270 LB = N - INT (N / 256) * 256
280 HB = INT (N / 256)
290 REM ***************
300 REM MACHINE LANGUAGE
310 REM ***************
320 POKE 768,169: REM LDA
330 POKE 769,15: REM COLOR FOR WHITE
340 POKE 770,143: REM LONG STA
350 POKE 771,LB: REM LOWBYTE
360 POKE 772,HB: REM HIGHBYTE
370 POKE 773,1: REM BANK 1
380 POKE 774,96: REM RTS
390 CALL 768: REM EXECUTE ROUTINE
400 RETURN
```

If you change the value in line 190 to 151, you'll have a light-blue-over-orange line across the top of your screen.

Programs in Double-Low-Resolution

Since double-low-resolution graphics can more easily and just as effectively be accessed from BASIC, there are only special occasions where it's necessary to use machine language to get to them.

Take a look at the program below for an example of what you can do with double-low-resolution graphics in BASIC.

Program 3-12.

```
 10 HOME
 20 GR
 30 COLOR= CR
 40 POKE 49246,0
```

```
 50 REM ************
 60 REM READ KEYBOARD
 70 REM ************
 80 WAIT - 16384,128
 90 K = PEEK ( - 16384)
100 POKE - 16368,0
110 IF K = 136 THEN K = 1: REM LEFT
120 IF K = 149 THEN K = 2: REM RIGHT
130 IF K = 138 THEN K = 3: REM DOWN
140 IF K = 139 THEN K = 4: REM UP
150 IF K = 195 OR K = 227 THEN K = 5
160 IF K = 241 OR K=209 THEN VTAB 22 : END
170 ON K GOSUB 300,400,500,600,700
180 PLOT H,V
190 FOR PAUSE = 1 TO 50: NEXT PAUSE
200 GOTO 80
300 REM *****
310 REM LEFT
320 REM *****
330 H = H - 1
340 IF H < 0 THEN H = 0
350 RETURN
400 REM *****
410 REM RIGHT
420 REM *****
430 H = H + 1
440 IF H > 79 THEN H = 79
450 RETURN
500 REM ****
510 REM DOWN
520 REM ****
530 V = V + 1
540 IF V > 47 THEN V = 47
550 RETURN
600 REM **
610 REM UP
620 REM **
630 V = V - 1
```



```
640 IF V < 0 THEN V = 0
650 RETURN
700 REM ************
710 REM CHANGE COLOR
720 REM ************
730 HOME
740 VTAB 22: HTAB 1
750 INPUT "Color Code 0-15 ";CL
760 COLOR= CL
770 RETURN
```

The program will give you the option to change colors when you press C. To quit the program, press Q.

Animating Low-Resolution Graphics

Animating low- or double-low-resolution graphics involves the same steps. Let's use double-low-resolution since it offers a larger screen area to work with.

Animation in low-resolution graphics works much the same as it does in text, except that instead of erasing with a blank space, you erase with a black dot.

A feature of animation not covered in chapter 2 is using *drawn objects*. In that chapter, you used single text characters instead of several different characters to make a new character. In low-resolution graphics, you probably want to move something other than a single block of color.

The best way to deal with more complicated objects is with subroutines that move a single unit at a time. First, draw your object in a subroutine using variables for the *x* and *y* positions of the object. Next, run a loop that repeatedly jumps to the subroutine to animate the object.

To make it clear how this works, start with simple horizontal movement. A boat will serve as the animated object. The boat will be gray with a yellow smokestack. To make it more interesting, puffs of white smoke will come out of the stack. The following program shows all of the elements you need. The boat's superstructure is drawn with HLIN, and the rest with PLOT. The puffs of smoke are alternated to leave a smoke trail behind the boat.

Program 3-13.

```
 10 TEXT : HOME
 20 GR
 30 POKE 49246,0
 40 REM *****
 50 REM WATER
 60 REM *****
 70 COLOR= 2: REM DARK BLUE
 80 FOR X = 36 TO 39
 90 HLIN 0,79 AT X
100 NEXT X 110 X = 35
120 FOR HP = 1 TO 73
130 GOSUB 160
140 NEXT HP
150 GOTO 430
160 REM *****
170 REM ERASE
180 REM *****
190 COLOR= 0: REM BLACK
200 PLOT HP - 1,X
210 PLOT HP,X - 1
220 PLOT HP + 2,X - 2
230 PLOT HP + 2,X - 3
240 REM *****
250 REM BOAT
260 REM ****
270 COLOR= 10
280 HLIN HP,HP + 6 AT X
290 HLIN HP + 1,HP + 4 AT X - 1
300 REM **********
310 REM SMOKESTACK
320 REM **********
330 COLOR= 13: REM YELLOW
340 PLOT HP + 3,X - 2
350 PLOT HP + 3,X - 3
360 REM **********
370 REM SMOKE PUFFS
380 REM **********
```



```
390 IF FLAG = 0 THEN COLOR= 15:FLAG = 1: GOTO 410
400 FLAG = 0: COLOR= FLAG
410 PLOT HP + 3,X - 4
420 RETURN
430 WAIT - 16384,128
440 TEXT : HOME : LIST
```

With a little planning, it's not too difficult to move block figures as easily as it is to move text. The only difference is that your program jumps to an entire subroutine to make the move rather than to a couple of PRINT statements.

Summary

Low-resolution graphics offer ease of use with lots of color. This graphics mode is useful for making colorful drawings, action and educational games, and, as you'll see in a later chapter, charts and graphs.

Its major advantage lies in the powerful BASIC statements that can be used for either low-resolution or double-low-resolution graphics. It's possible to enhance the horizontal resolution without losing any of the color and still enable full use of all Applesoft statments for standard low-resolution.

Chapter 4

High-Resolution Graphics

The major distinguishing characteristic of high-resolution graphics, besides its higher-resolution, is its location. Both color and text reside in screen memory beginning at 1024 ($400) and ending at 2039 ($7F7). Now, however, let's explore two different areas of memory.

Page	Start	End
Primary	8192 ($2000)	16383 ($3FFF)
Secondary	16384 ($4000)	24575 ($5FFF)

Though not mentioned in previous chapters, text and low-resoluton graphics also have a secondary page. But since the secondary page conflicts with BASIC, and since you used BASIC extensively in Chapters 1–3, that page wasn't investigated. In this chapter, however, you'll discover how the primary and secondary page of graphics can be used together.

To get started, let's summarize the BASIC statements and some POKEs and CALLs used in this chapter. Note that some of the statements are identical to those you used in the last chapter.

Command	Function
HGR	Clears primary high-resolution screen and goes to high-resolution mode.
HGR2	Clears secondary high-resolution screen and goes to high-resolution mode.
HCOLOR=	Set high-resolution color (0–7) 0=black 4=black 1=green *5=check 2=blue *6=check 3=white 7=white

*These colors vary depending on type of TV or color monitor. Use the "Color Check" program to test.

HPLOT X,Y	Places a dot of light (pixel) at specified X,Y position. X = 0–279, Y = 0–191

Command	Function
HPLOT X,Y	Places a dot of light (pixel) at specified X,Y position. X = 0–279, Y = 0–191
HPLOT X,Y TO X1,Y1 TO X2,Y2...	To draw lines in high-resolution graphics HPLOT from X,Y to X1,Y1 (and so on) using single or multiple HPLOT statements. HPLOT 10,10 TO 20,20 TO 0,55 HPLOT TO 99,111
CALL 62454	From 40-column mode, clears the screen to last plotted color. (Does not work in 80-column mode even though resolution is the same.)

The following memory addresses act as soft switches. POKE with 0 to turn on or off each special feature.

Address	Switch Function
49232	Graphics mode
49233	Text mode
49234	All text or all graphics
49235	Mix text and graphics (4 lines of text only)
49236	Display page 1
49237	Display page 2
49238	Lo-res graphics
49239	Hi-res graphics

Simple Things in Hi-Res

To start, let's do something really simple—draw a box on the screen.

Program 4-1.

```
 10 HGR
 20 FOR X = 3 TO 7 STEP 4
 30 HCOLOR= X
 40 HPLOT 0,0 TO 279,0 TO 279,159
 50 HPLOT TO 0,159 TO 0,0
 60 HOME : VTAB 21
 70 PRINT "HIT ANY KEY ";
 80 WAIT - 16384,128
 90 NEXT X
100 VTAB 22
```


In that program, two different whites were used. The white with HCOLOR value of 7 and that with a value of 3 are different—press a key and change the HCOLOR from 3 to 7 to see how the vertical lines change. Later, when you see how high-resolution color is placed in memory, you'll see how this works. For now, though, just be aware of it.

Let's check out all of the colors on your TV set or monitor.

Program 4-2.

```
 10 ROW = 10
 20 HOME
 30 HGR
 40 FOR X = 1 TO 7
 50 HCOLOR= X
 60 FOR K = 1 TO 4
 70 ROW = ROW + 1
 80 HPLOT 0,ROW TO 279,ROW
 90 NEXT K
100 NEXT X
110 VTAB 21
120 PRINT "HIT A KEY"
130 WAIT - 16384,128
140 TEXT
150 HOME
160 LIST
```

Horizontal lines will give you a good idea of how your TV or monitor shows color in high-resolution. However, vertical lines will show you how different vertical columns give different colored lines.

A slight change to the program above puts vertical lines on the screen instead.

Program 4-3.

```
 10 COL = 10
 20 HOME
 30 HGR
 40 FOR X = 1 TO 7
 50 HCOLOR= X
```

```
 60 FOR K = 1 TO 4
 70 COL = COL + 1
 80 HPLOT COL,0 TO COL,159
 90 NEXT K
100 COL = COL + 5
110 NEXT X
120 VTAB 21
130 PRINT "HIT A KEY"
140 WAIT - 16384,128
150 TEXT
160 HOME
170 LIST
```

Make any ajustments with the color on your TV or monitor using these two programs.

Before moving on to more complicated high-resolution graphics, let's have a little fun with some loops and the random number generator.

These next programs create different colored triangles and draw lines on the screen in different colors.

Program 4-4.

```
10 HGR
20 FOR C = 1 TO 7
30 FOR X = 1 TO 279
40 HCOLOR= C
50 HPLOT 139,0 TO X,100
60 NEXT X
70 NEXT C
```

Program 4-5.

```
 10 HGR
 20 FOR X = 1 TO 30
 30 REM ************
 40 REM RANDOM VALUES
 50 REM ************
 60 C = INT ( RND (1) * (7) + 1)
 70 V = INT ( RND (1) * (159) + 1)
 80 H = INT ( RND (1) * (279) + 1)
```



```
 90 V1 = INT ( RND (1) * (159) + 1)
100 H1 = INT ( RND (1) * (279) + 1)
110 HCOLOR= C
120 HPLOT H,V TO H1,V1
130 NEXT
```

Notice how the RND statement was used to generate random lines that would stay within the screen boundaries.

Saving Graphic Drawings

There'll be times when you'll want to save something you've created on either page of your high-resolution graphics screen. The random drawing program listed above may have created a terrific design you want to look at later. Since it was created randomly, it would be difficult to recreate.

In Applesoft BASIC, the BSAVE command saves binary files to your disk as binary (BIN or B) files. You already know that the primary page of high-resolution resides at addresses $2000–$3FFF, and the secondary page is at $4000–$5FFF. That means each screen is $2000 (8192) bytes long. All you have to do to save a screen is type:

BSAVE *filename*,**A$2000,L$2000** (primary page)
BSAVE *filename*,**A$4000,L$2000** (secondary page)

Let's start with a simple example that will create a pattern for you to save.

Program 4-6.

```
10 HGR
20 FOR X = 1 TO 100
30 C = C + 1
40 IF C = 8 THEN C = 1
50 HCOLOR= C
60 S = ABS( INT( SIN (X) * 100 ))
70 HPLOT 139,0 TO S,139
80 NEXT
```

That's on the primary page, so you should use $2000 as the starting address. Call it *COLORSIN*.

```
BSAVE COLORSIN,A$2000,L$2000
```

Since it's a large file, it will take a bit to save to disk. To reload the file, do this:

```
HGR (press Return)
VTAB 21 (press Return)
BLOAD COLORSIN (press Return)
```

One problem with loading graphics saved in binary files is that they're erased with an HGR or HGR2 command. For the time being, use the above method to help you get by. Later on you'll see some other tricks you can use with loading and saving graphics.

Switching Screens

Two of the important elements of high-resolution graphics are the *primary* and *secondary* pages. You can switch pages without erasing the contents of one screen to show the other. Not only can you switch between primary and secondary graphics pages, but you can switch between low- and high-resolution graphics without losing anything.

The following program shows how to do this with POKEs to the screen soft switches. Notice also that the lines are only drawn once, and after switching to the opposite resolution, they are reclaimed without being redrawn.

Program 4-7.

```
 10 HOME
 20 REM ************
 30 REM LO-RES LINE
 40 REM ************
 50 GR
 60 COLOR= 15
 70 HLIN 0,39 AT 10
 80 VTAB 21
 90 PRINT "LOW-RESOLUTION LINE"
100 WAIT - 16384,128: POKE 49168,0
110 REM ************
120 REM HI-RES LINE
```



```
130 REM ************
140 HGR
150 HCOLOR= 7
160 HPLOT 0,50 TO 279,50
170 VTAB 21
180 PRINT "HIGH-RESOLUTION LINE"
190 WAIT - 16384,128: POKE 49168,0
200 REM ***************
210 REM SWITCH TO LO RES
220 REM ***************
230 POKE 49238,0
240 HOME
250 VTAB 21
260 PRINT "LOW-RESOLUTION LINE"
270 WAIT - 16384,128: POKE 49168,0
280 REM ***************
290 REM SWITCH TO HI RES
300 REM ***************
310 POKE 49239,0
320 VTAB 21
330 PRINT "HIGH-RESOLUTION LINE"
340 WAIT - 16384,128
350 TEXT : HOME : LIST
```

Animation with Page Switching

In case you're wondering what kind of applications use screen switching, take a look at animation. Animation which uses screen switching works much like other forms of animation.

One common use of screen switching in animation is to display movement from side to side or up and down. In the next program, for instance, a can of paint is held by a rod between two parts of a machine and is shaken back and forth. The results may be crude, but they illustrate how good graphics animation is made possible by drawing similar pictures on different screens.

Actually switching the screen is accomplished with a single POKE. Returning to the original screen takes another POKE. If HGR and HGR2 are used instead, the animation isn't as smooth—remember that HGR and HGR2 clear the screen each time they're used.

Try changing the value in the PAUSE loop to see the effect of different speeds in screen switching.

Program 4-8.

```
 10 HOME
 20 HGR
 30 GOSUB 200
 40 HGR2
 50 GOSUB 400
 60 FOR X = 1 TO 20
 70 POKE 49236,0
 80 FOR PAUSE = 1 TO 100
 90 NEXT PAUSE
100 POKE 49237,0
110 FOR PAUSE = 1 TO 100
120 NEXT PAUSE
130 NEXT X
140 WAIT - 16384,128
150 TEXT : HOME
160 LIST
170 END
200 REM ******
210 REM PAGE 1
220 REM ******
230 HCOLOR= 3
240 HPLOT 10,50 TO 100,50
250 HPLOT TO 100,100 TO 10,100
260 HPLOT TO 10,50
270 HPLOT 100,75 TO 120,75
280 HPLOT TO 120,60 TO 140,60
290 HPLOT TO 140,80 TO 120,80
300 HPLOT TO 120,75
310 HPLOT 140,75 TO 150,75
320 HPLOT 150,50 TO 190,50
330 HPLOT TO 190,100 TO 150,100
340 HPLOT TO 150,50
350 RETURN
400 REM ******
```



```
410 REM PAGE 2
420 REM ******
430 HPLOT 10,50 TO 100,50
440 HPLOT TO 100,100 TO 10,100
450 HPLOT TO 10,50
460 HPLOT 100,75 TO 110,75
470 HPLOT TO 110,60 TO 130,60
480 HPLOT TO 130,80 TO 110,80
490 HPLOT TO 110,75
500 HPLOT 130,75 TO 150,75
510 HPLOT 150,50 TO 190,50
520 HPLOT TO 190,100 TO 150,100
530 HPLOT TO 150,50
540 RETURN
```

Slide Show

Two graphics screens can also present a "slide show" effect. Suppose, for example, that you have a number of pieces of computer art to show an audience. While you're showing one, you can load the other on the other screen. When you switch to the next picture, it's there waiting—no delay.

Let's take a look at an example. First, however, you'll need a simple high-resolution drawing program to create your "slides."

This program incorporates many of the features of the double-low-resolution drawing program from chapter 3 and also includes a BSAVE option to save the creations to disk.

Program 4-9.

```
 10 HOME
 20 HGR
 30 HCOLOR= 7: REM START WITH WHITE
 40 REM ************
 50 REM READ KEYBOARD
 60 REM ************
 70 WAIT - 16384,128
 80 K = PEEK ( - 16384)
 90 POKE - 16368,0
```

```
100 IF K = 136 THEN K = 1: REM LEFT
110 IF K = 149 THEN K = 2: REM RIGHT
120 IF K = 138 THEN K = 3: REM DOWN
130 IF K = 139 THEN K = 4: REM UP
140 IF K = 195 OR K = 227 THEN K = 5
150 IF K = 241 OR K = 209 THEN VTAB 24: PRINT CHR$(4);
    "BSAVE GRAPHIC1,A$2000,L$2000";: END
160 ON K GOSUB 200,260,320,380,500
170 HPLOT H,V
180 FOR PAUSE = 1 TO 50: NEXT PAUSE
190 GOTO 70
200 REM *****
210 REM LEFT
220 REM *****
230 H = H - 1
240 IF H < 0 THEN H = 0
250 RETURN
260 REM *****
270 REM RIGHT
280 REM *****
290 H = H + 1
300 IF H > 279 THEN H = 279
310 RETURN
320 REM ****
330 REM DOWN
340 REM ****
350 V = V + 1
360 IF V > 151 THEN V = 151
370 RETURN
380 REM **
390 REM UP
400 REM **
410 V = V - 1
420 IF V < 0 THEN V = 0
430 RETURN
500 REM ************
510 REM CHANGE COLOR
520 REM ************
530 HOME
```



```
540 VTAB 22: HTAB 1
550 INPUT "Color Code 0-7 ";CL
560 HCOLOR= CL
570 RETURN
```

Next, draw a graphics screen. Save it to disk by pressing the S key. In line 150, change GRAPHIC1 to GRAPHIC2, then create and save a second graphics screen. Change line 150 once more so that it uses GRAPHIC3, then draw and save.

It really doesn't matter what you draw—just make sure the three are different so you can tell when one is switched with another.

Use this next program to retrieve and show your graphics. Look at lines 60 and 70, which BLOAD a program to the primary and secondary graphics pages, respectively. You BLOAD a graphic to the primary page by specifying A$2000; using A$4000 sends the graphic to the secondary page.

Program 4-10.

```
10 TEXT : HOME
20 REM *************
30 REM LOAD GRAPHICS
40 REM *************
50 D$ = CHR$ (4)
60 PRINT D$"BLOAD GRAPHIC1,A$2000"
70 PRINT D$"BLOAD GRAPHIC2,A$4000"
100 REM ****************
110 REM TURN ON GRAPHICS
120 REM ****************
130 POKE 49232,0
140 POKE 49234,0
150 POKE 49236,0
160 POKE 49239,0
170 WAIT - 16384,128
180 POKE - 16368,0
200 REM ****************
210 REM SWITCH TO PAGE 2
220 REM ****************
230 POKE 49237,0
```

```
240 REM ****************
250 REM LOAD NEXT GRAPHIC
260 REM ****************
270 PRINT D$"BLOAD GRAPHIC3,A$2000"
280 WAIT - 16384,128
290 POKE - 16368,0
300 REM ***************
310 REM SWITCH TO PAGE 1
320 REM ***************
330 POKE 49236,0
340 WAIT - 16384,128
350 POKE - 16368,0
360 TEXT : HOME
370 LIST
```

Color in Memory

Color memory is a lot more complex in high-resolution than in low-resolution graphics.

Each pixel is part of an eight-bit pattern. Seven of those bits correspond to pixels visible on the screen, while the eighth bit is used as a *control bit.*

The following program gives you a quick look at the 255 different color value combinations—each is seven pixels long.

Program 4-11.

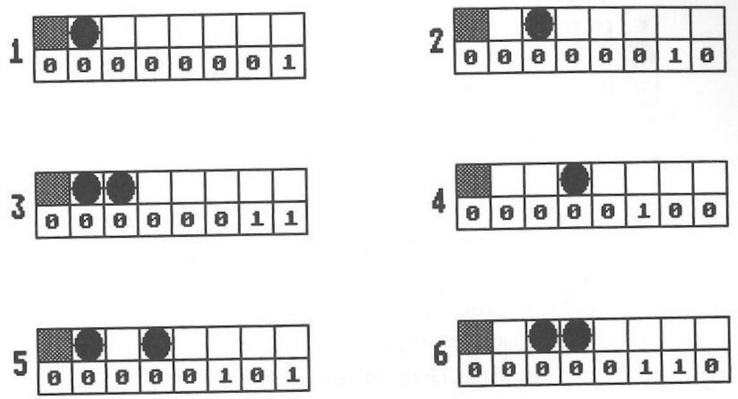
```
 5 C = 1
10 HGR
20 FOR X = 8192 TO 18382 STEP 40
30 POKE X,C
40 C = C + 1
50 NEXT X
60 WAIT - 16384,128
70 TEXT : HOME
80 LIST
```

To see what's going on in this program, it's necessary to take a closer look at how the pixels get to the screen. The bytes used to display pixels and color contain a binary number made up of 0s

and 1s. Depending on the bit or bits that are turned *on* (have a value of 1), a pixel will be lit or not lit.

High-resolution color values 1–8 and their binary equivalents are shown in Figure 4-1. The lit pixel is shown as a black dot. (The shaded box indicates the control bit.)

Figure 4-1. Values 1–8



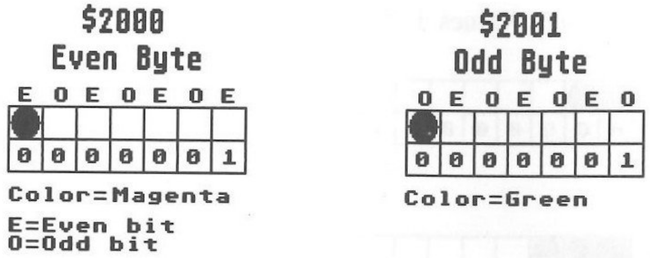
Notice that the pixels are in *complementary* positions relative to the bit or bits turned on. In other words, when the rightmost bit is on (has a value of 1), then the pixel at the far left is lit. In the color value 8, the center pixel is atop the turned-on bit since the bit is in the middle of the byte.

Depending on which address you use to store a value, the first bit of a screen color byte will be considered either *even* or *odd.* The first screen address, $2000 (8192) is considered *even.* The second screen address, $2001 (8193) is *odd.* The addresses alternate in this pattern of even-odd.

Let's say you store a color byte at location $2032 (8242). That's an even address, so the first bit of the color byte is considered *even* as well. Think of the addresses arranged as pairs, one even and an-

other odd, and things may be clearer. Here's two color bytes, one stored at $2000 and the other at $2001. (Both bytes show only seven bits in Figure 4-2; consider the eighth bit, the control bit, as off. More on the control bit and its effect shortly.)

Figure 4-2. Even and Odd Bytes and Bits



Notice that both bytes have the same value—the difference is that the byte on the left produces a magenta pixel and the one on the right, a green pixel. If a lit pixel is in an *even* bit column, it's magenta. Look at Figure 4-2 again and check that the bit which creates magenta is in an even column (it is). If a pixel is set in an odd bit column, it's green.

Thus, if you wanted to put two magenta-colored pixels in adjacent addresses, you'll need to set an even bit in the even byte and an even bit in the odd byte. The odd byte's value could be 2 (10 in binary).

You've already seen how to create magenta and green—what if two adjacent bits are set, one in an even bit column, the other in an odd bit column? White is the result.

Any other combination yields black.

The first four color values show all these combinations (Figure 4-3). Keep in mind, however, that these colors apply *only* when the control bit is off.

When the eighth bit is turned on, 128 is added to the value. Blue (cyan) replaces magenta, and orange replaces green. Figure 4-4 shows what the color bytes would look like.

Figure 4-3. Black, Magenta, Green, and White

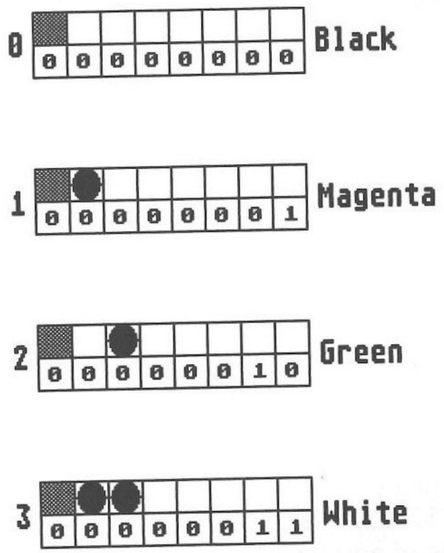
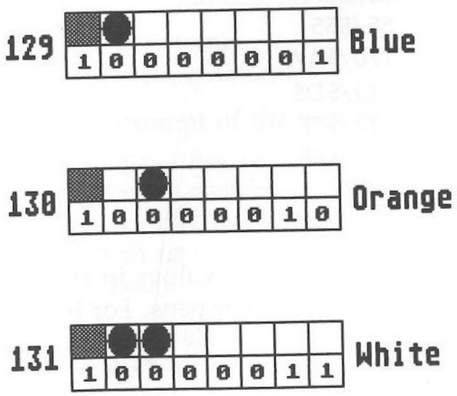


Figure 4-4. Blue and Orange



Storing Colors in Memory

To get an idea of how colors stored in memory look, type one of the following from the immediate mode.

In BASIC, type

HGR (press Return)
POKE 8192,1 : POKE 8193,2 (press Return)

In machine language, type

HGR (press Return)
CALL-151 (press Return)

When you see the * prompt, type

2000: 01 02 (press Return)

Either gives you two magenta dots near the top of your screen. Notice that the dots are not adjacent. What you've done is to place the value 1 in the first even byte ($2000/8192) and the value 2 in the first odd byte ($2001/8193).

To create a line of adjacent dots, you'll need combinations of pixels which line up on the even or odd columns. This chart provides the proper values.

	Even	Odd
Magenta	85/$55	42/$2A
Green	42/$2A	85/$55
Blue	213/$D5	170/$AA
Orange	170/$AA	213/$D5
White	127/$7F	127/$7F
	255/$FF	255/$FF
Black	0/$0	0/$0
	128/$80	128/$80

Try POKEing in some of these values in consecutive high-resolution addresses to see what happens. For instance, enter this program to draw an orange line across the top of the screen.

Program 4-12.

```
10 HGR
20 FOR EVEN = 8192 TO 8230 STEP 2
30 POKE EVEN,170
40 POKE EVEN+1,213
```



```
50 NEXT EVEN
60 WAIT -16284,128
70 TEXT: HOME
```

Replace the 8230 in line 20 with 16382 to fill the entire screen with orange with lines. It's easy to see the three divisions of screen memory when you do.

Double-High-Resolution Graphics

On your Apple IIGS, double-high-resolution graphics is a little like double-low-resolution graphics. You flip the same soft switch at address 49246 ($C05E) to turn it on. After that, however, it's far more difficult to control.

First, BASIC statements are not recognized on the bank 1 screen. That means all values have to be sent there by machine language routines. As you've seen, that means a rather trying bout with pixels. Another problem is putting graphics where you want them on both parts of the double high-resolution screen without running out of memory. (Assume you're doing this without using more than 48K of memory, and employing BASIC as much as you can.)

To get a sense of what's going on here, put your IIGS into 80-column mode, and enter the following:

HGR (press Return)
CALL-151 (press Return)

At the * prompt of the monitor, type

*C054: 00 (press Return)	Double-resolution graphics primary page
*00<01/2000.01/3FFFZ (press Return)	Clear bank 1 screen
*01/2000: 01 02 (press Return)	Bank 1 addresses
*00/2000: 01 02 (press Return)	Bank 0 addresses
*Q (press Return)	Return to BASIC

You should be back in BASIC.

That short routine gave you the magenta dots, but this time they're closer together than before. With double the resolution, you can get 560 dots across the screen.

Let's draw a white line across the screen. Addresses from $2000 to $2027 must be filled with a value on both screens. A technique to quickly do this in the monitor (the place you go after

typing CALL-151) is to use the same method you just used to clear the double-resolution graphics screen. The Z (for Zap) command in the monitor fills the range of given addresses with the specified value. The less than symbol (<) is used. You want to create a line in white, and from the chart listed earlier, you know that either $7F or $FF will do the trick (if you're not in 80-column mode, enter it now—don't forget to set the soft switch at location 49246 also).

HGR (press Return)
CALL-151 (press Return)

Again, at the * monitor prompt, type

*C054: 00 (press Return)
*7F<01/2000.01/2027Z (press Return)
*7F<00/2000.00/2027Z (press Return)
*Q (press Return)

That white line at the top of your screen is 560 dots across, not 280 as in single-resolution graphics.

Using an Array

The next program loads up all of the addresses into a sequential array. Notice that other than the HGR statement, there are no graphics statements in the program. The screen begins to fill with graphics, however.

Program 4-13.

```
 10 POKE 49246,0
 20 REM ***************
 30 REM PLACE SEQUENTIAL
 40 REM TABLE IN ARRAY
 50 REM ***************
 60 HGR
 70 INVERSE
 80 VTAB 21
 90 PRINT " BE PATIENT AND WATCH THE SCREEN "
100 DIM X%(8192)
110 FOR A = 8192 TO 8272 STEP 40
120 FOR B = 0 TO 896 STEP 128
130 FOR C = 0 TO 7168 STEP 1024
```



```
140 FOR D = 0 TO 38 STEP 2
150 N = A + B + C + D
160 X%(X) = N
170 X = X + 1
180 N = A + B + C + D + 1
190 X%(X) = N
200 X = X + 1
210 NEXT D
220 NEXT C
230 NEXT B
240 NEXT A
250 NORMAL : HOME
260 PRINT CHR$ (7)
270 VTAB 21
280 PRINT "ALL DONE"
290 END
```

What's happened is that the array is so large that it starts filling up the part of memory used by high-resolution graphics. You may also notice the black columns between the graphics. These are the bank 1 addresses not affected by the array.

You'll need a rearranged array to do graphics correctly, so let's use the LOMEM statment. LOMEM sets the lower limits for storing variable and array data. Such data is prevented from entering memory addresses below this limit.

To make sure you've got enough room, set LOMEM to 16384 ($4000), just above the primary page of high-resolution graphics. This will give you enough room for the large array and still preserve the space needed for graphics.

The following program does this, drawing a sequential line with the large array. It only uses 600 bytes in both banks (300 in each), but it illustrates how to produce drawings in double-high-resolution graphics.

Program 4-14.

```
 10 LOMEM: 16384
 20 POKE 49246,0
 30 REM ***************
 40 REM PLACE SEQUENTIAL
 50 REM TABLE IN ARRAY
```

```
 60 REM ***************
 70 HGR
 80 INVERSE
 90 VTAB 21
100 PRINT " LOADING ARRAY "
110 DIM X%(8192)
120 FOR A = 8192 TO 8272 STEP 40
130 FOR B = 0 TO 896 STEP 128
140 FOR C = 0 TO 7168 STEP 1024
150 FOR D = 0 TO 38 STEP 2
160 N = A + B + C + D
170 X%(X) = N
180 VTAB 21: HTAB 19: PRINT " *"
190 X = X + 1
200 N = A + B + C + D + 1
210 X%(X) = N
220 X = X + 1
230 NEXT D
240 VTAB 21: HTAB 19: PRINT "* "
250 NEXT C
260 NEXT B
270 NEXT A
280 NORMAL
290 HOME : VTAB 22
300 REM ************************
310 REM POKE SEQUENTIAL ADDRESSES
320 REM ************************
330 FOR X = 0 TO 300 STEP 2
340 N = X%(X)
350 V = 170: REM EVEN ORANGE
360 POKE N,V
370 GOSUB 500
380 N = N + 1
390 V = 213: REM ODD ORANGE
400 POKE N,V
410 GOSUB 500
420 NEXT X
430 VTAB 22
440 END
```



```
500 REM ******************
510 REM CONVERT TO 2 BYTE #
520 REM ******************
530 LB = N - INT (N / 256) * 256
540 HB = INT (N / 256)
600 REM ***************
610 REM MACHINE LANGUAGE
620 REM ***************
630 POKE 768,169: REM LDA
640 POKE 769,V: REM VARIABLE COLOR
650 POKE 770,143: REM LONG STA
660 POKE 771,LB: REM LOWBYTE
670 POKE 772,HB: REM HIGHBYTE
680 POKE 773,1: REM BANK 1
690 POKE 774,96: REM RTS
700 CALL 768: REM EXECUTE ROUTINE
710 RETURN
720 VTAB 21: HTAB 19: PRINT "* "
```

Chapter 5

Shapes and Bitmapped Graphics

Dealing with graphics shapes on the Apple IIGS takes some careful organization and planning. Though the details may be a bit intimidating at first, once you create a few shapes, it's really quite easy.

It's easier to buy a commercial shape editor, or to get a public domain editor from an Apple user group or one of the information services—but then you may not fully understand how to use the shapes. Creating shapes yourself will give you much more in-depth experience in how they work. If you do decide to use a shape editor, skip the first section of this chapter and go directly to the part which concerns manipulating shapes.

Getting into Shapes

As with most programming, you'll deal with 8-bit bytes when generating shapes and shape tables.

It's easier, however, to think of the eight bits in three packets—one of two bits, and two of three bits each: 2, 3, 3. The first packet of two bits is generally unused, so you'll work with two three-bit sets for the most part.

The shape table you'll build is based on values entered into a series of bytes using binary and hexadecimal codes. All of these, of course, must be further translated into decimal numbers which can be used from a BASIC program. With planning and clear explanation, you'll quickly grasp shape creation, and see what they can do for your own programs.

Drawing a shape in memory is much like drawing with paper and pencil. In fact, you can draw the shape on graph paper first, then plot it in your computer's memory.

Imagine a shape as something you draw in memory by specifying directions and draw/not draw instructions with three-bit packets; occasionally you can use a two-bit packet (see the chart listed in Figure 5-1).

Draw in Memory

As you plot the shape in memory, you can move the drawing tool (without actually drawing) up, down, left, or right. That's four choices. You can also move and plot in those same four directions. That's four more choices, for a total of eight.

It's easiest to understand drawing shapes if you use a combination of binary and hexadecimal numbers to start. Once you have the hexadecimal values, you can translate them into decimal numbers for a BASIC program.

Here are the available moves and the values they carry:

Figure 5-1. Moves and Values

Move	Plot	Binary	Hex
Up	No	000	0
Right*	No	001 or 01	1
Down*	No	010 or 10	2
Left*	No	011 or 11	3
Up	Yes	100	4
Right	Yes	101	5
Down	Yes	110	6
Left	Yes	111	7

Each move or move/plot is recorded in a three-bit segment of a byte, or a two-bit segment if appropriate. Notice the moves with an asterisk and their binary value. If, in the sequence of plotting and moving, one of those moves is to be recorded and the next available segment is two bits long, then it can be recorded as a two-bit value. Let's look at an example.

Shape A = Move/plot to the left, move/plot up, and move to the left.
1. Move/plot left = 111
2. Move/plot up = 100
3. Move left = 011 or 11

The three segments of the byte will be numbered from 1, 2, and 3, respectively, so that you can keep the sequence in order.

1. Segment 1 = 111
2. Segment 2 = 100
3. Segment 3 = 11

The byte which holds the values for these moves and plots looks like Figure 5-2.

Figure 5-2. Three Moves, One Byte

```
          Segment 3          Segment 2          Segment 1
Bit#   7     6     |  5     4     3     |  2     1     0
       1     1     |  1     0     0     |  1     1     1
```

That was pretty simple.

Consider what would be required if you had the following move, however. Note that it's just a slightly different sequence of moves and plots.

Shape B = Move/plot to the left, move/plot up, move/plot up, and move to the left.
1. Move/plot left = 111
2. Move/plot up = 100
3. Move/plot up = 100
4. Move left = 011 or 11

Since the third action involves a three-bit operation, you can't use Segment 3. You have to go to the next byte.

Figure 5-3 shows what Shape B will look like when mapped.

Figure 5-3. Four Moves, Two Bytes

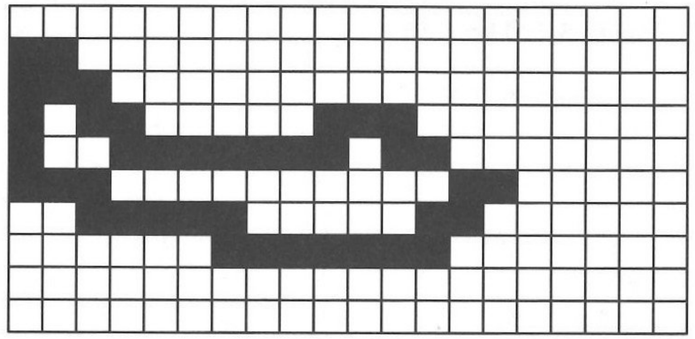
```
          Segment 3          Segment 2          Segment 1
Bit#   7     6     |  5     4     3     |  2     1     0   Byte #1
       0     0     |  1     0     0     |  1     1     1
             (*)                (2)                (1)

Bit#   7     6     |  5     4     3     |  2     1     0   Byte #2
                   |  0     1     1     |  1     0     0
                               (4)                (3)
```

The numbers in parentheses show the sequence of placing the values. You can draw any shape in memory with the same procedure, although most shapes take many more steps than the short examples you've seen so far.

You're ready to draw something you can see on your computer. Figure 5-4 shows a jet airplane drawn on graph paper.

Figure 5-4. Jet on Graph Paper

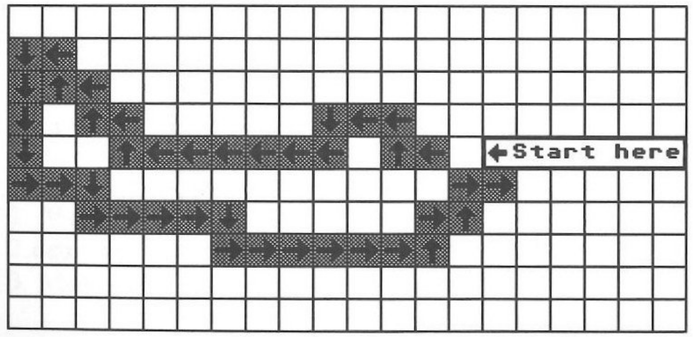


Beginning with a point above the nose of the jet, you can draw a continuous line which ends at the tip of the nose. All entries will be move and plot, so you'll be using these values:

← 111
↑ 100
→ 101
↓ 110

And Figure 5-5 shows shows how to plot each point.

Figure 5-5. Plotting the Jet



Following the plotting arrows around the jet, you'd get the this binary pattern:

1. 111
2. 100
3. 111
4. 111
5. 110
6. 111
7. 111
8. 111
9. 111
10. 111
11. 111
12. 100
13. 111
14. 100
15. 111
16. 100
17. 111
18. 110
19. 110
20. 110
21. 110
22. 101
23. 101
24. 110
25. 101
26. 101
27. 101
28. 101
29. 110
30. 101
31. 101
32. 101
33. 101
34. 101
35. 101
36. 100
37. 101
38. 100
39. 101
40. 101

The final step is to translate the binary into decimal and/or hexadecimal. Consider two different methods for accomplishing that task.

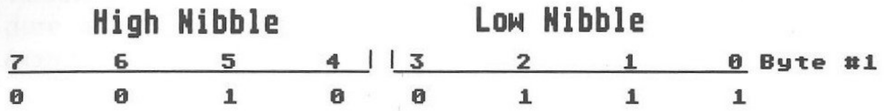
Translate by Hand

The first method is more time consuming, but it will give you a better understanding of the translation process.

First, rearrange the byte breakdown. Instead of treating each byte as three sets of bits (one two-bit packet and two three-bit packets), think of a byte now as composed of two four-bit segments. Let's look at what will be the first byte of the shape table.

Instead of Segments, you now have a *high* and a *low nibble.* The binary values in the bits are the same, but to translate the byte into a hexadecimal value, it's simply easier to break it into nibbles.

Figure 5-6. High Nibble/Low Nibble



Once you've broken it into nibbles, translating the byte into hexadecimal is easy—just substitute the four-bit nibble for a single-digit hex value. Notice that in the following chart the full range of four binary digits (0000–1111) exhausts the single-digit range of hexadecimal numbers (0–F). That's another clue as to why hexadecimal values are used with computers.

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

To translate the byte shown in Figure 5-6, for instance, you'd first break it into its two nibbles:

High nibble: 0010 = $2
Low nibble: 0111 = $7

Thus the byte's value in hex is $27.

Turn to appendix C, which offers a hexadecimal-to-decimal translation table, and you can see that $27 in hex is 39 in decimal.

You're done—with one byte. That's a lot of work. Once you

get everything organized, though, it's relatively simple—and, as you'll see, well worth the effort.

Here's a complete shape table depicting the jet from Figure 5-4.

Byte	Segment 3	Segment 2	Segment 1	Hex	Decimal
0	00	100	111	27	39
1	00	111	111	3F	63
2	00	111	110	3E	62
3	00	111	111	3F	63
4	00	111	111	3F	63
5	00	100	111	27	39
6	00	100	111	27	39
7	00	100	111	27	39
8	00	110	111	37	55
9	00	110	110	39	54
10	00	101	110	2D	45
11	00	110	101	35	53
12	00	101	101	2D	45
13	00	101	101	2D	45
14	00	101	110	2E	46
15	00	101	101	2D	45
16	00	101	101	2D	45
17	00	100	101	25	37
18	00	100	101	25	37
19	00	101	101	2D	45

Software Translation

A second method to get that same table would be to write a program which translates binary values into decimal and hexadecimal. This is a lot easier when dealing with a large table of values.

The next program does this for you.

Program 5-1.

```
 10 TEXT : HOME : GOSUB 440
 20 INPUT "How many bytes in your shape ";BN
 30 FOR B = 1 TO BN
100 REM ****************
110 REM BINARY CONVERSION
120 REM ****************
```

```
130 INPUT "Binary number ";B$
140 IF LEN (B$) < > 8 THEN 130
150 FOR X = 0 TO 7
160 V$ = MID$ (B$,X + 1,1)
170 V = VAL (V$): IF V > 1 THEN X = 7: PRINT "All digits must be
    '0' or '1'" CHR$ (7): NEXT : GOTO 130
180 P = 7 - X
190 IF V = 1 THEN BV = 2 ^ P
200 TD = TD + BV
210 BV = 0
220 NEXT X
230 PRINT "Your decimal value is ";TD
300 REM **************
310 REM CONVERT TO HEX
320 REM **************
330 POKE 780,TD
340 PRINT "Your hex value is ";
350 CALL 768
360 PRINT
370 TD = 0
380 NEXT B
390 END
400 REM ***************
410 REM MACHINE LANGUAGE
420 REM HEX CONVERSION
430 REM ***************
440 FOR A = 1 TO 12
450 READ MB
460 POKE 767 + A,MB
470 NEXT
480 RETURN
490 DATA 169,164,32,237,253,173,12,3,32,218,253,96
```

Once you've translated the shape into decimal and hexadecimal values, let's see how to use them in shapes.

Entering the Shape Table

You can either POKE in the decimal numbers from BASIC or, using the monitor, enter the hexadecimal values directly into a memory location. In either case the procedure involves two things:

- Placing the shape somewhere in memory where it won't clash with other information.
- Telling your Apple where to find the shape information.

First of all, store the starting address of your shape in a special register (which draws shapes) at locations $E8 and $E9 (232 and 233). This register holds a *pointer* to shape definitions. For this demonstration, store the shape in memory starting with address $300 (decimal 768)—that's a location with some free memory. The microprocessor of the Apple needs this information in a low byte, high byte pattern. This is backwards to us humans, but it suits the machine just fine.

Thus, the starting address will be stored in locations $E8 and $E9 as

$E8: 00
$E9: 03

Do It from BASIC

To store this information from BASIC, you must break up the $300 into two parts and POKE the decimal equivalents into those locations. Fortunately, you can make a direct translation from hex to decimal.

Hex	Decimal
00	00
03	03

Since the low byte is stored in the first address and the high byte in the second, enter

```
POKE 232,0
POKE 233,3
```

Finally, you must provide more information at the beginning and end of the shape data.

First byte:	Total number of shape definitions
Second byte:	Unused
Third and fourth bytes:	*Relative offset* for beginning of shape

The *relative offset* indicates where the shape data actually begins. It's not an address, but the number of bytes *after* the starting address. The relative offset makes the shape table *relocatable*—it can be placed in any area of free memory.

In this example, there's only one shape, and the shape data begins immediately after the relative offset information. That will be in the fifth byte, but since the first byte is considered zero, the value will be four (4).

Byte 0 = 01 (Number of shapes)
Byte 1 = 00 (Unused)
Byte 2 = 04 (Low byte of offset)
Byte 3 = 00 (High byte of offset)
Byte 4 = 39 (First value of shape)
Bytes 5–N (N=last value of shape table)
Byte N+1 = 00 (Indicates end of shape table)

After you've managed all this, you're ready to write a program which will place the shape information in memory and use the shape.

The table for the jet shape includes 20 bytes. There are 25 bytes total, however, since you have to use 4 bytes at the beginning of the table and 1 byte at the end. That means you'll have to POKE 25 shape values into memory, beginning at $300 (768).

Let's use the following loop.

```
FOR X = 768 TO (768 + 24)
```

To make it easy, put all the information in DATA statements, then have the loop read the data and sequentially place it in the assigned addresses. Before that, though, remember to indicate where the shape information is stored. Thus, $E8 and $E9 (232 and 233) will be POKEd with

```
POKE 232,0 : POKE 233,3
```

This program does everything for you.

Program 5-2.

```
 10 REM *****
 20 REM SETUP
 30 REM *****
 40 POKE 232,0: POKE 233,3
 50 TEXT : HOME
100 REM ************
110 REM READ IN DATA
120 REM ************
130 FOR X = 768 TO 768 + 24
140 READ S
150 POKE X,S
160 NEXT
170 DATA 1,0,4,0
180 DATA 39,63,62,63,63,39,39,39,55,54
190 DATA 45,53,45,45,46,45,45,37,37,45,0
```

Run this program. Nothing happens. The shape is in memory, but you haven't drawn it yet. You need more statements and commands to *see* the shape. You'll learn how that's done in a moment. For now, though, let's see how to enter the shape table data in memory with the monitor.

Do It from the Monitor

First of all, you need to store the starting address information in memory locations $E8 and $E9.

Here's how.

```
CALL -151 (press Return)
```

When you see the monitor's asterisk (*) prompt, type

```
*E8: 00 03 (press Return)
*Q (press Return)
```

You're now back in BASIC.

Now you're ready to enter the same data as you placed in memory with BASIC—this time with the monitor. You'll enter hexadecimal values.

```
CALL-151 (press Return)
```

Wait for the * prompt to appear; then type in the starting address ($300), a colon (:), and the 25 byte values.

```
*300: 01 00 04 00 27 3F 3E 3F 3F 27 27 27 37 39 2D 35 2D 2D 2E
 2D 2D 25 25 2D 00 (press Return)
*Q (press Return)
```

You may have one question—*why bother with the monitor when a program automatically does everything for you from BASIC?*

Two reasons. First, if you want to quickly edit the shape, you can enter the monitor and make changes quickly and easily byte-by-byte. Secondly, you can save your shape as a binary (BIN) file, which can be loaded into memory from disk at any time.

To save the above shape as a binary file, for instance, type this after you've exited the monitor and returned to BASIC:

```
BSAVE JET,A$300,L$19 (press Return)
```

or

```
BSAVE JET,A768,L25 (press Return)
```

This command saves the 25 (L$19) values stored in memory starting with 768 ($300) as the binary file JET (BSAVE JET).

You could have done the same thing from BASIC. By entering it in the monitor, however, you can get a better idea how shapes are stored in memory.

To use the shape table in another program, use this line:

```
100 PRINT CHR$(4);"BLOAD JET,A$300"
```

That single line can replace lines 100–190 in the last BASIC program.

Shape Manipulation

You've done a lot of work in creating a shape; now it's time to use it. Let's first take a look at the special shape statements you can use.

- **SCALE.** The SCALE statement sets the size of your shape. A SCALE value of 1 uses the single-pixel plot resolution you created your shape with. Higher value scales create larger shapes with lower resolution.

- **ROT.** The ROT value ROTates your shape in one of eight angles. ROT recognizes values 0, 8, 16, 24, 32, 40, 48, and 56. Other ROT values are dropped to the next lower value (for example, 12 will be treated as 8). At angles other than 0, 90, 180, or 360, the shapes are distorted.
- **DRAW.** Using the high-resolution pixel matrix, you plot the X and Y coordinates using the following format: *DRAW N% AT X,Y* where N% is the shape number and X and Y are the horizontal and vertical coordinates on your high-resolution screen.
- **XDRAW.** XDRAW has the same format as DRAW except it draws the complement of the color existing on the screen. When using animation, XDRAW is preferable to DRAW.

Add these lines to Program 5-2.

Place shape on screen.

```
200 REM ******************
210 REM PUT SHAPE ON SCREEN
220 REM ******************
230 HGR
240 HCOLOR= 3
250 SCALE= 1
260 ROT= 1
270 DRAW 1 AT 100,100
```

(You can use either XDRAW or DRAW to place a figure on the screen.)

Move shape.

```
200 REM **********
210 REM MOVE SHAPE
220 REM **********
230 HGR
240 HCOLOR= 3
250 SCALE= 1
260 ROT= 1
270 FOR X = 0 TO 279
280 DRAW 1 AT X,70
290 FOR PAUSE = 1 TO 2
300 NEXT PAUSE
310 XDRAW 1 AT X,70
320 NEXT X
```

For movement, it's best to alternate DRAW and XDRAW. It simply helps to think of it as a *draw* and *erase* sequence. It would be possible to use XDRAW for both, but that makes it a bit more confusing.

Notice the short pause loop in lines 290–300. That prevents the screen from clouding over the shape as it moves. Remove the lines to see what happens without that short delay. On some monitors, it may not make a difference, but if your screen seems to drop shadows over moving shapes, put in a short delay loop.

Change the program to see if you can make the jet fly from the upper left corner to the lower right.

Rotate shape.

```
200 REM ************
210 REM ROTATE SHAPE
220 REM ************
230 HGR
240 HCOLOR= 3
250 SCALE= 1
260 FOR R = 0 TO 56 STEP 8
270 ROT= R
280 XDRAW 1 AT 30 + R,30 + R
290 FOR PAUSE = 1 TO 400
300 NEXT PAUSE
310 NEXT R
```

Rotation can give spectacular animated effects. In the example, it looks like the jet is spinning out of control or performing a loop. Add and change the following lines for another view of the flip.

```
290 FOR PAUSE = 1 TO 100
305 XDRAW 1 AT 30 + R,30 + R
320 R=0
330 XDRAW 1 AT 30 + R,30 + R
```

That second view really lets you see the animation possibilities of using shape tables.

Change scale.

```
200 REM ************
210 REM CHANGE SCALE
220 REM ************
```



```
230 HGR
240 HCOLOR= 3
250 ROT= 0
260 FOR S = 1 TO 4
270 SCALE= S
280 DRAW 1 AT 50 + S,20 * S
290 NEXT S
```

Changing the scale lowers the resolution of your shape, but you can see the vectors better. In the jet you created, for instance, it's possible to see a gap between the nose and the cockpit area.

Some shapes may actually look better if they're scaled upwards. For the most part, though, larger scaling is most often used for editing shapes. See if you can figure out what would have to be added to the jet shape to fill in the gap near the nose. (Hint—move backward without plotting from the tip of the nose, then plot the gap.)

Try experimenting with the color and your shapes. Change the white value from 3 to 7 in HCOLOR. Even that will give you a different appearing shape. Other colors may break up the shape, and by filling in a more or less blank area of a shape, you can get interesting color results. The key is to experiment, then judge the results yourself.

Bitmapped Graphics

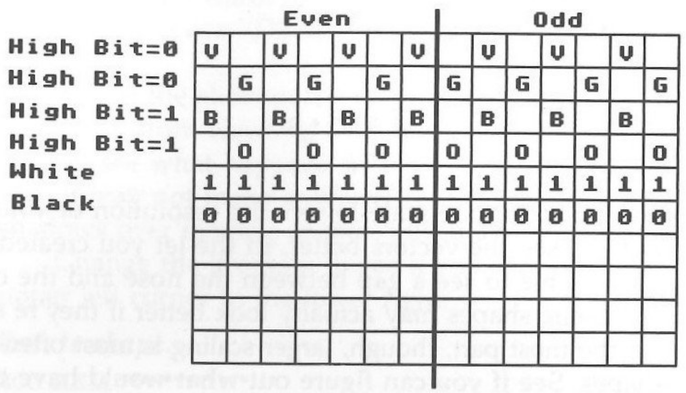
Another type of graphics character generation is *bitmapped graphics.* This process involves drawing a graphics figure with a bit configuration instead of plotting vectors.

Graphics blocks are built using seven bits of each byte in a character. The bytes are then stored in the high-resolution memory. By changing the addresses of the bytes, it's possible to program animation. You can use as many bytes as you have room for, in memory and on the screen. And since color is determined by the bit pattern, it's possible to create multicolored characters.

Color and High Bits

The reason that bitmapped graphics use seven bits in creating a figure is that the eighth bit controls the color. Figure 5-7 shows which on bits create which colors. Note that color not only depends on which bits are set on, but also whether the eighth bit (also called the *high bit*) is on or off.

Figure 5-7. Bitmapped Graphics Colors



V = bit is on, color is violet
G = bit is on, color is green
B = bit is on, color is blue
O = bit is on, color is orange

Within a single byte, it's possible to have up to four colors (black, white, and green/violet or blue/orange).

To see how bitmapped graphics and color work, let's look at some examples using the monitor. You'll start with the color violet. According to Figure 5-7, you need the following configuration.

Violet in binary

Low bit --> **1 0 1 0 1 0 1 0** <-- High bit

Important note: Remember that what shows above is the *reverse* of what you'd normally find in examining a binary number. It's *reversed,* meaning that the high bit is on the right and the low bit is on the left. Using the binary-to-decimal/hexadecimal conversion program which you used to convert binary values in making shape tables, start with the rightmost bit and enter the binary number from right to left. Running that through the conversion program should give you the hexadecimal value of $55.

Now type the following.

```
TEXT:HGR (press Return)
CALL-151 (press Return)
```

You're now in the monitor, and should see the asterisk prompt. Type

```
*2C00: 55 (press Return)
```


There's a violet line in the top left corner, isn't there? Now, while you're still in the monitor, enter

```
*2C01: 55 (press Return)
```

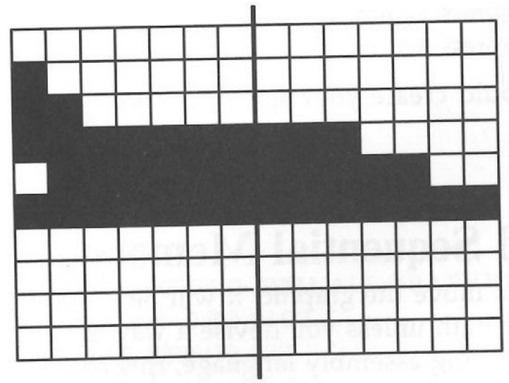
This time the line is green. The zero byte ($2C00) is even, and the first byte ($2001) is odd. That's why, although both bytes contain the value $55, one is violet, the other green.

You have to use $2A every other btye to get two violet lines in a row. That's because on odd bytes, the color for green is $55; for violet, it's $2A.

A Space Shuttle

To start with, let's try something simple. The next shape uses only white, so it doesn't matter whether bytes are even or odd.

Figure 5-8. The Shuttle



Remember that the rightmost bit controls the color in bitmapped shapes—since the shape includes white only, the high bit (rightmost in this case) is always zero. It's set off from the rest of the data by a vertical line.

Left Data	Right Data
1. 1 0 0 0 0 0 0 0 | 0	0 0 0 0 0 0 0 0 | 0
2. 1 1 0 0 0 0 0 0 | 0	0 0 0 0 0 0 0 0 | 0
3. 1 1 1 1 1 1 1 1 | 0	1 1 1 0 0 0 0 0 | 0
4. 0 1 1 1 1 1 1 1 | 0	1 1 1 1 1 0 0 0 | 0
5. 1 1 1 1 1 1 1 1 | 0	1 1 1 1 1 1 1 1 | 0

These values are translated to

1. $1/1	$0/0
2. $3/3	$0/0
3. $7F/127	$7/7
4. $7E/126	$1F/31
5. $7F/127	$7F/127

Plug the graphic into memory from the monitor to see it on the screen. To enter it in the correct order, it's necessary to place each pair of bytes in consecutive rows. Remember that your Apple's high-resolution memory is not consecutive or linear—each row begins $400 above the previous row.

```
TEXT : HGR (press Return)
CALL-151 (press Return)
*2C00:01 00 (press Return)
*3000:03 00 (press Return)
*3400:7F 07 (press Return)
*3800:7E 1F (press Return)
*3C00:7F 7F (press Return)
```

That should create your space shuttle. Crude, but you get the idea.

Moving and Sequential Memory

If you want to move the graphic, it will be a complex task trying to figure out the path unless you devise a way to sequentially line up the memory. Using assembly language, this could be done very efficiently. However, it *is* possible from BASIC. Here's how.

The trick is to store the addresses of high-resolution memory in an array, just like you did for screen memory. That is, the addresses—beginning in the upper left corner and proceeding from left to right to the lower right corner—will be put into an integer array. Then, using the array data, you can POKE the data for the graphics character into sequential locations on the screen.

The following program does just that, and also provides a running record of the array space filling as it configures the array into a linear set. Finally, it fills the screen sequentially with a white, violet, and green pattern.

Program 5-3.

```
 10 HOME
 20 REM ********************
 30 REM REARRANGE HGR MEMORY
 40 REM ********************
 50 LOMEM: 16383
 60 DIM HR%(8192)
 70 FOR A = 8192 TO 8272 STEP 40: REM $2000-$2050
 80 FOR B = A TO 9168 STEP 128: REM $23D0 STEP $80
 90 FOR X = B TO 16383 STEP 1024: REM $3FFF STEP $400
100 FOR Y = X TO X + 39: REM $27
110 HR%(V%) = Y
120 V% = V% + 1
130 NEXT Y
140 K = V% / 8192
150 P = K * 100
160 P% = INT (P) + 7
170 HTAB 1: VTAB 22: PRINT STR$ (P%);"% "
180 NEXT X
190 NEXT B
300 NEXT A
310 REM *****************
320 REM FILL SEQUENTIALLY
330 REM *****************
340 HGR
350 POKE 49234,0: REM ALL GRAPHICS
360 FOR F = 0 TO 8192
370 POKE HR%(F),123
380 NEXT
```

There are a couple of things to note here. First, LOMEM was set to 16383. That means the lowest memory for storing the array was right at the beginning of HGR2. If the program had not done that, the array would have stored data on Page 1 of the high-resolution screen.

Just for fun, change line 50 to read

```
50 HGR
```

and you can see the high-resolution screen fill with color bits representing the data that's loaded into that part of memory.

Graphics on the Screen

The following program, which uses rearranged memory, shows how a bitmapped graphic can be placed on the screen. (Remember that each row is 40 bytes wide—to jump a row, add 40 since you can treat the high-resolution screen sequentially.)

Program 5-4.

```
 10 HOME
 20 REM ********************
 30 REM REARRANGE HGR MEMORY
 40 REM ********************
 50 LOMEM: 16383
 60 DIM HR%(8192)
 70 FOR A = 8192 TO 8272 STEP 40: REM $2000-$2050
 80 FOR B = A TO 9168 STEP 128: REM $23D0 STEP $80
 90 FOR X = B TO 16383 STEP 1024: REM $3FFF STEP $400
100 FOR Y = X TO X + 39: REM $27
110 HR%(V%) = Y
120 V% = V% + 1
130 NEXT Y
140 K = V% / 8192
150 P = K * 100
160 P% = INT (P) + 7
170 HTAB 1: VTAB 22: PRINT STR$ (P%);"% "
180 NEXT X
190 NEXT B
200 NEXT A
300 REM ***********************
310 REM MAKE BITMAPPED GRAPHIC
320 REM ***********************
330 HOME : TEXT
340 HGR
350 FOR J = 1 TO 5
360 READ A,B
370 POKE HR%(G%),A: POKE HR%(G% + 1),B
380 G% = G% + 40
390 NEXT J
400 REM ***************
```



```
410 REM DATA FOR GRAPHIC
420 REM ***************
430 DATA 1,0,3,0,127,7,126,31,127,127
```

Finally, you'll want the graphic to "ride the array." If you hadn't rearranged the screen, it would have been difficult to move the graphic diagonally while keeping it in one piece. However, since you *can* track it on the array instead of on the screen directly, it's simply a matter of using a step loop. This next program shows how.

Program 5-5.

```
 10 HOME
 20 REM ********************
 30 REM REARRANGE HGR MEMORY
 40 REM ********************
 50 LOMEM: 16383
 60 DIM HR%(8192)
 70 FOR A = 8192 TO 8272 STEP 40: REM $2000-$2050
 80 FOR B = A TO 9168 STEP 128: REM $23D0 STEP $80
 90 FOR X = B TO 16383 STEP 1024: REM $3FFF STEP $400
100 FOR Y = X TO X + 39: REM $27
110 HR%(V%) = Y
120 V% = V% + 1
130 NEXT Y
140 K = V% / 8192
150 P = K * 100
160 P% = INT (P) + 7
170 HTAB 1: VTAB 22: PRINT STR$ (P%);"% "
180 NEXT X
190 NEXT B
200 NEXT A
300 REM ***********************
310 REM MOVE BITMAPPED GRAPHIC
320 REM ***********************
330 HOME : TEXT
340 HGR
350 FOR F = 1 TO 1640 STEP 41
360 G% = F: RESTORE
```

```
370 FOR J = 1 TO 5
380 READ A,B
390 POKE HR%(G%),A: POKE HR%(G% + 1),B
400 G% = G% + 40
410 NEXT J
420 FOR PAUSE = 1 TO 30: NEXT PAUSE
430 G% = F
440 FOR E = 1 TO 5
450 POKE HR%(G%),0: POKE HR%(G% + 1),0 : REM Erase graphic
460 G% = G% + 40
470 NEXT E
480 NEXT F
500 REM ***************
510 REM DATA FOR GRAPHIC
520 REM ***************
530 DATA 1,0,3,0,127,7,126,31,127,127
```

That was a bit jerky, and you might want to experiment with the delay loop in line 420 to see if you can smooth it out. If you want to see a straight downward vertical move (not diagonal) use STEP 40 in the movement loop.

Summary

If you want to create arcade and graphics games, you'll want to further explore vector and bitmapped graphics. The power of your Apple IIGS is considerable. And as you've seen, even though BASIC has its limitations, there's a great deal of flexibility and control at your fingertips. Using shape tables and shape statements in BASIC, and creating and manipulating shapes is not as difficult as it may first seem. The trick is to get organized and take things a step at a time. The planning is well worth the results, and with more explorations into how your Apple IIGS works, you'll be able to create more and more spectacular effects.

Bitmapped graphics offer a real challenge to the programmer, but they also offer a lot of opportunities. What's here can only whet your appetite for this part of high-resolution graphics.

Chapter 6

Making Graphs and Circles

Now that you know something about both low- and high-resolution graphics, how about putting that knowledge to use? This chapter shows you how to create graphs, charts, and circles in both resolutions and how to combine them. You'll see how to construct bar graphs, scatter graphs, and line graphs.

Low-Resolution

When drawing on the low-resolution screen, you're limited by a 40 × 48 matrix. With text at the bottom of the screen, the matrix is further restricted to 40 × 40. Since you may want to label charts and graphs, we'll need those four lines of text at the bottom. Plan on using the smaller 40 × 40 matrix most of the time.

Since a chart or graph is a graphic representation of data, you need a way to translate that data into a visual graphic. To start, let's draw a couple of bars which have the same base, but are of different colors and heights.

Program 6-1.

```
10 GR
20 COLOR= 3
30 FOR A= 1 TO 3
40 VLIN 5,39 AT 5 + A
50 NEXT A
60 COLOR= 4
70 FOR B = 1 TO 3
80 VLIN 20,39 AT 10 + B
90 NEXT B
```

If that graph represented data of, say sales of two different types of software packages, you could see at a glance that the first bar (magenta) illustrates considerably higher sales than the second

bar (green). A range from 0 to 39, however, doesn't allow for sales above 40 units.

What you do is make the graphic data proportionately representative of the data. For example, each vertical step could represent 10, 20, 100, 1000 or even a million units instead of just one. In fact, it doesn't matter what unit a vertical bar represents as long as all the data are proportional. The problem is how to make that data proportional.

Proportional Data

To understand how to make the data equally proportional and still fit on the screen, let's start with a simple example. Suppose you want to chart two sets of data. One set has a maximum value of 10 and the other has a maximum value of 1000. Instead of using 40 (0–39) as the maximum, use 39.9—that provides the limit up to, but not including, 40. (You could be even more precise and use 39.9999, but that's unnecessary.)

With the first set of data with a maximum of 10, you'll want to use more than just 10 vertical positions. You want all 40. So you'll need to change the basic unit from 1 to something else. By dividing 39.9 by 10, the basic unit becomes 3.99. For each full data point, the bar should be incremented by 3.99. To represent 7 units, for instance, you'd calculate the following:

D= 7
D= 7 * 3.99
D= 27.93

Let's take another number—five—and do that same thing.

D= 5
D= 5 * 3.99
D= 19.95

In other words, 7 has the same relationship to 27.93 as 5 has to 19.95.

Now let's see if a simple bar chart will provide the correct proportions. This next program compares the raw figures and the proportional figures (the first set uses the raw figures, and the second set uses the proportional ones).

Program 6-2.

```
10 GR
20 COLOR= 7
30 VLIN 39-7,39 AT 10
40 VLIN 39-5,39 AT 12
50 COLOR= 9
60 VLIN 39-27.93,39 AT 15
70 VLIN 39-19.95,39 AT 17
```

Notice in lines 30 and 40, and in lines 60 and 70, how the value to plot was subtracted from 39. That's because lower values are *higher* on the screen. To reverse that, all you have to do is subtract the converted value from 39.

Now let's look at the data where 1000 is the maximum value. Again you divide 39.9 by the maximum value (1000), but instead of getting a whole number, the result is a fraction, .0399. Therefore, each unit will be less than 1. Using the values 500 and 700, let's see what happens:

500 * .0399 = 19.95
700 * .0399 = 27.93

You get the same values as you did when the maximum value was 10 and you used 5 and 7. As long as the proportions are consistent, you can chart any values you want. Thus, you can use this formula:

Ratio = 39.9/Maximum Value
N = N * Ratio

Now, any value, assuming it's equal to or less than the maximum value, will fit proportionately into the low-resolution matrix. The following simple program does that.

Program 6-3.

```
10 TEXT : HOME
20 INPUT "Maximum value ";MV
30 R = 39.9 / MV
40 FOR X = 1 TO 2
50 INPUT "Value ";N(X)
60 N(X) = N(X) * R
```

```
 70 NEXT
 80 GR
 90 FOR X = 1 TO 2
100 COLOR= X * 3 + 1
110 VLIN 39 - N(X),39 AT 3 * X
120 NEXT X
```

Draw a Proportional Chart

You're all set to create a program that will draw a proportional bar chart. To make it more interesting, it will toggle between showing the graphic and the raw numeric data, as well as automatically calculate the maximum value.

The program needs a variable to represent the maximum value entered. It uses the variable K, which is placed in the data entry loop to determine the maximum value. Each time through the loop, K is compared with the last data entered. If the new data is larger than K, then K is changed to the larger number. In that way, no matter when the maximum value is entered, it's always stored in the variable. To establish a ratio, using *R*, 39.9 is divided by K. (See lines 100–200.)

To toggle between the raw data and the graphic requires that the program have two arrays. The first array (D) stores the raw data, and the second array (G%) stores the graphic data. An integer array is used for the graphic data since fractions are ignored in plotting graphics. Finally, the program limits the number of entries to 15 so that it doesn't have to repeat the colors of the bars. If you want, you can change the program to accept up to 40 entries.

Program 6-4.

```
 10 TEXT : HOME
 20 INVERSE
 30 K = 0
 40 VTAB 10
 50 PRINT " HOW MANY ENTRIES:<15=MAX> ";
 60 NORMAL
 70 INPUT MAX
 75 IF MAX > 15 THEN PRINT CHR$ (7): GOTO 10
 80 HOME
 90 DIM D(MAX): DIM G%(MAX)
100 REM **********
110 REM DATA ENTRY
120 REM **********
130 FOR X = 1 TO MAX
140 INVERSE
150 PRINT " ENTER VALUE=> ";
160 NORMAL
170 INPUT " ";D(X)
180 IF D(X) > K THEN K = D(X)
190 NEXT X
200 R = 39.9 / K
300 REM ***************
310 REM CONVERT TO SCALE
320 REM ***************
330 FOR X = 1 TO MAX
340 G%(X) = INT (D(X) * R)
350 NEXT
400 REM **********
410 REM MAKE GRAPH
420 REM **********
430 GR
440 FOR X = 1 TO MAX
450 COLOR= X
460 P% = 40 - G%(X)
470 VLIN P%,39 AT X * 2
480 NEXT
490 INVERSE
500 PRINT " PRESS ANY KEY FOR NUMERIC DATA ";
510 NORMAL
520 WAIT - 16384,128: POKE - 16368,0
600 REM ****************
610 REM VIEW NUMERIC DATA
620 REM ****************
630 TEXT : HOME
640 VTAB 4
650 FOR X = 1 TO MAX
660 X$ = STR$ (X)
670 PRINT X$ + ".=> ";D(X)
```

```
680 NEXT
690 PRINT
700 INVERSE
710 PRINT " PRESS ANY KEY FOR CHART DATA "
720 PRINT " PRESS 'Q' TO QUIT "
730 WAIT - 16384,128
740 IF PEEK (49168) = 209 OR PEEK (49168) = 241 THEN
    NORMAL : END
750 POKE 49168,0
760 GOTO 400
```

You could have simply toggled the graphics on and off with a POKE instead of redrawing each time. That creates the graphic representation of the text from the numeric data. To see this at work, change line 760 as shown below and add four new lines.

```
760 POKE 49232,0
770 POKE 49235,0
780 POKE 49236,0
790 POKE 49238,0
800 NORMAL
```

In the next section you'll see how to toggle between high-resolution and low-resolution graphics.

Labeling Charts

The difficulty level in labeling a chart depends on whether you're using 40 or 80 columns. In 40-column mode, numbers will line up directly under the vertical bars. In 80-column mode, though, you'll have to adjust text position a good deal; the advantage to 80 columns, of course, is that you can get more information under the chart.

VTAB position 21 places the text directly under the chart in either 40 or 80 columns. Starting with a simple 40-column text example, you can see that the horizontal alignment of text is the HTAB position, plus one, of the horizontal position of the low-resolution plot.

Program 6-5.

```
 10 TEXT : HOME
 20 A = 30
 40 B = 20
 50 GR
 60 COLOR= 4
 70 VLIN 39 - A,39 AT 10
 75 COLOR= 5
 80 VLIN 39 - B,39 AT 20
 90 VTAB 21
100 HTAB 10 + 1
110 PRINT "A";
120 PRINT SPC(9);"B"
```

With only two plots, labeling the bars is simple. With a greater number of plots, though, it's easier to set up a loop that resets VTAB 21 and uses a variable for HTAB. For example, by modifying one of the above programs, you can change it from a general plot and graph program to a monthly one with both data entry and graph labels. By using READ and DATA, you can put all of the data entry and chart labels together. In that way, it's easy to change the nature of the chart just by altering the data entry and graph labels.

Program 6-6.

```
 10 TEXT : HOME
 20 INVERSE
 30 K = 0
 40 MAX = 12
 50 DIM D(MAX): DIM G%(MAX)
100 REM **********
110 REM DATA ENTRY
120 REM **********
130 FOR X = 1 TO MAX
140 READ D$
150 INVERSE
160 PRINT " ENTER VALUE FOR ";D$
170 NORMAL
```

```
180 INPUT " ";D(X)
190 IF D(X) > K THEN K = D(X)
200 NEXT X
210 R = 39.9 / K
220 HOME
300 REM ***************
310 REM CONVERT TO SCALE
320 REM ***************
330 FOR X = 1 TO MAX
340 G%(X) = INT (D(X) * R)
350 NEXT
400 REM **********
410 REM MAKE GRAPH
420 REM **********
430 GR
440 FOR X = 1 TO MAX
450 COLOR= X
460 P% = 40 - G%(X)
470 VLIN P%,39 AT X * 2
480 NEXT
500 REM ***********
510 REM MONTH LABEL
520 REM ***********
530 FOR X = 1 TO 12
540 READ M$
550 VTAB 21
560 HTAB 2 * X + 1
570 PRINT M$
580 NEXT
600 REM *************
610 REM CALENDER DATA
620 REM *************
630 DATA JANUARY,FEBRUARY,MARCH,APRIL
640 DATA MAY,JUNE,JULY,AUGUST,SEPTEMBER
650 DATA OCTOBER,NOVEMBER,DECEMBER
660 DATA J,F,M,A,M,J,J,A,S,O,N,D
```

In 80-column mode, you can place two digits directly under a horizontal bar. However, since an 80-column text character takes up half the width of a low-resolution plot, it's possible to get two characters directly under a vertical line. Program 6-7 shows how.

Program 6-7.

```
 10 TEXT : HOME
 20 GR
 30 FOR G = 1 TO 15
 40 COLOR= G
 50 VLIN 39 - G,39 AT G * 2
 60 VTAB 21
 70 REM *********************************
 80 REM CALCULATE FOR 80-COLUMN CHARACTER
 90 REM *********************************
100 HTAB (G * 4) + (G < 10) + 1
110 PRINT G
120 NEXT
```

Notice line 100—an additional space was added by summing the truth value of G < 10. As long as G is less than ten, the truth value is one, which is added to the HTAB value until two digits are present. The first digit of the two-digit number replaces the space which preceded numbers less than ten. (Remember, if the truth value is false, the value is zero.)

High-Resolution Graphic Charts

Although high-resolution graphics provide greater resolution, you still need to respect proportions. The work area is roughly 279 × 159 on the combined high-resolution graphic/four-line text page. This gives you a finer (higher resolution) plotting map than the low-resolution 40 × 40 screen.

Horizontal Spacing

With low-resolution graphs, bars are relatively fat. In high-resolution graphics, however, there are 280 horizontal plots on the screen. If you're not careful, you'll find yourself using only a small portion of the screen, crowding the plots. For instance, this next program plots ten different pieces of random data.

Program 6-8.

```
 10 TEXT : HOME
 20 HGR
 30 HCOLOR= 3
 40 FOR X = 1 TO 10
 50 REM ********************
 60 REM Y POSITION FROM 1-158
 70 REM ********************
 80 Y% = INT ( RND (1) * (159))
 90 REM ********************
100 REM X POSITION FROM 1-10
110 REM ********************
120 HPLOT X,Y%
130 NEXT
```

In this program, the *y*-axis was randomly generated to be between 1 and 158, but the *x*-axis was set between 1 and 10. Even if every other horizontal position were used, it would take up less than ten percent of the horizontal screen. As with determining the ratio with a maximum value, you need to find the ideal spacing between plots on the horizontal screen.

Divide 279 by the number of plots to find this ideal spacing. It can be represented as

S% = INT(279/NP)

where S% is the spacing variable and NP is the number of plots.

To see how this works, let's make a scatter graph. Each plot in the scatter graph is a relative plot point on the high-resolution screen. As with low-resolution graphic charts, you need to come to a ratio based on the maximum value entered and multiply each vertical point by that ratio. The formula

Ratio=158.9/Max Value

is used to establish the ratio (see line 130 below).

Program 6-9.

```
 10 TEXT
 20 HOME
 30 K = 0
 40 INVERSE
```



```
 50 PRINT " HOW MANY PLOTS? ";
 60 NORMAL
 70 INPUT " ";P%
 80 DIM P(P%),G%(P%)
 90 FOR X = 1 TO P%
100 INPUT "PLOT VALUE";P(X)
110 IF P(X) > K THEN K = P(X)
120 NEXT X
130 R = 158.9 / K
140 FOR X = 1 TO P%
150 G%(X) = INT (159 - (P(X) * R))
160 NEXT
170 S% = INT (279 / P%)
200 REM *************
210 REM SCATTER GRAPH
220 REM *************
230 HGR
240 HCOLOR= 3
250 HPLOT S% - (S% - 1),G%(1)
260 FOR X = 2 TO P%
270 Y% = G%(X)
280 HPLOT ((X * S%) - (S% - 1)),Y%
290 NEXT X
```

Line Graphs

Depending on what values you entered, the scatter graph either looked like a random set of dots scattered across the screen or something more meaningful. By drawing lines between the plot points, you can more readily see a trend or something that looks more sensible. To do that, use the HPLOT TO statement.

After plotting the initial point, all the other points are plotted with HPLOT TO so that a line is drawn from the last point to the next point. For example, the following program generates a random line graph.

Program 6-10.

```
 10 TEXT : HOME
 20 HGR
 30 HCOLOR= 3
```

```
 40 S% = 279 / 10
 50 HPLOT S%, INT ( RND (1) * (159))
 60 FOR X = 2 TO 10
 70 REM ********************
 80 REM Y POSITION FROM 1-158
 90 REM ********************
100 Y% = INT ( RND (1) * (159))
110 REM ********************
120 REM DRAW LINE TO NEXT PLOT
130 REM ********************
140 HPLOT TO X * S%,Y%
150 NEXT
```

Keep typing **RUN** to watch it generate all kinds of plots.

Vertical Grid

Look at the following two line graphs.

Figure 6-1. Jagged Plot Line

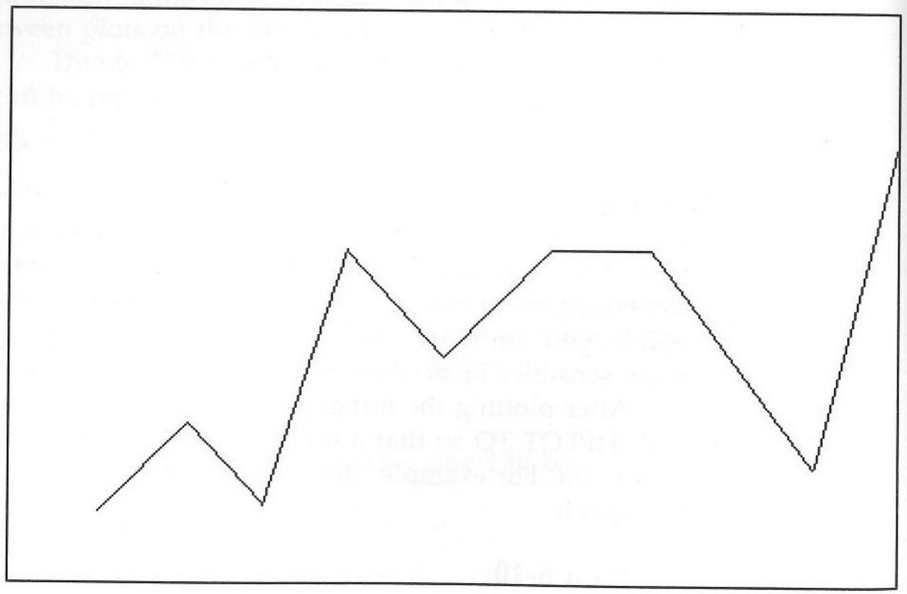
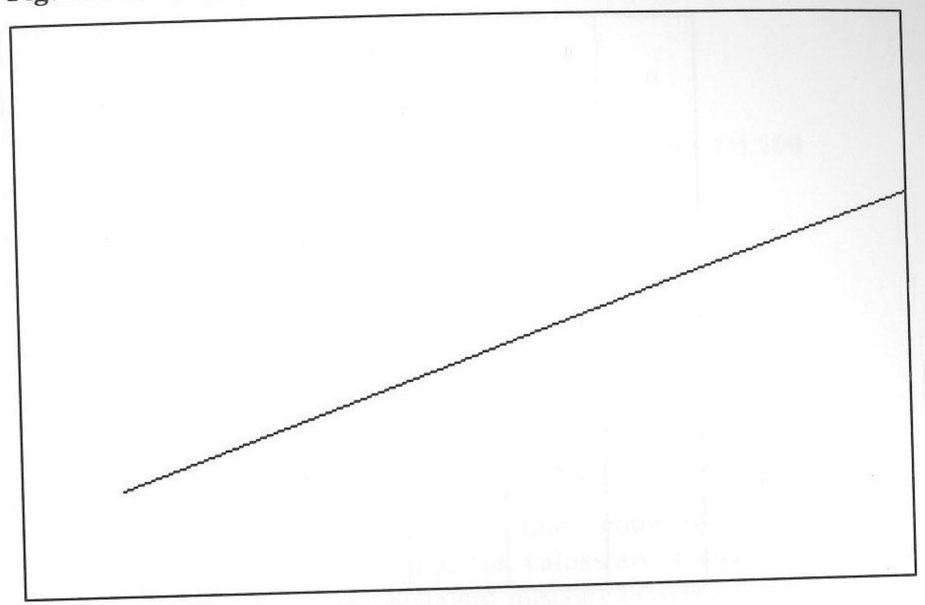


Figure 6-2. Smooth Plot Line

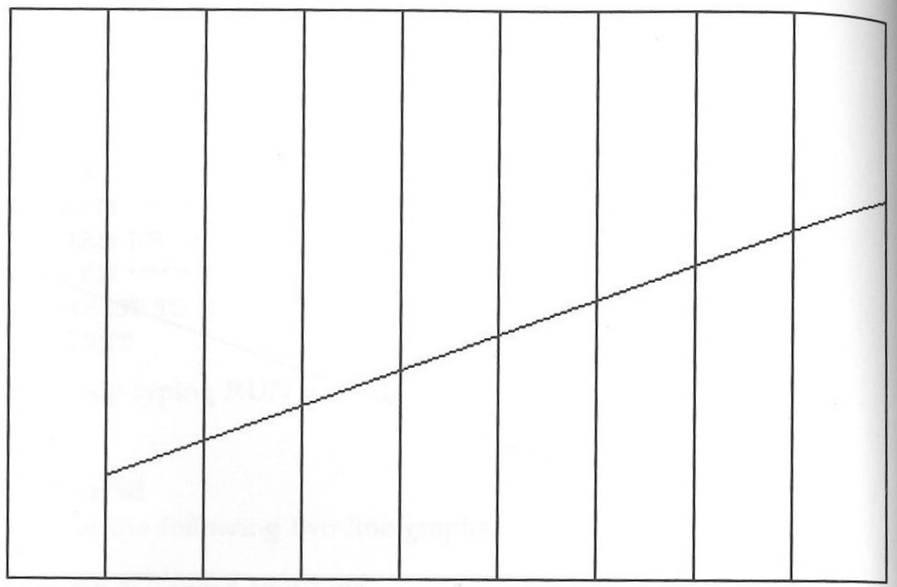


In Figure 6-1, it's clear where the plot points are since they go up and down from plot to plot. However, in Figure 6-2, you can't tell where the break points are, since there's a steady increase in the plot values. To make it easier to see the plot points, it would be useful to superimpose a vertical grid on the chart so that it looks like Figure 6-3.

Now it's clear where the plot points are located.

You can use the same spacing variable to place the vertical lines as you used to space the horizontal screen. At each horizontal plot point, a vertical line is drawn from the top of the screen to the bottom. Also notice that the HCOLOR for the vertical lines is a different white than the HCOLOR for the graph lines. The vertical lines will either be all orange or orange and blue, depending on the number of plots. (They're not actually white since two horizontally adjacent pixels must be lit to generate white, and only single horizontal pixels are used in the lines. To make white, use two adjacent vertical lines to make the grid lines.)

Figure 6-3. Plot with Vertical Grid



Program 6-11.

```
 10 TEXT
 20 HOME
 30 K = 0
 40 INVERSE
 50 PRINT " HOW MANY PLOTS? ";
 60 NORMAL
 70 INPUT " ";P%
 80 DIM P(P%),G%(P%)
 90 FOR X = 1 TO P%
100 INPUT "PLOT VALUE";P(X)
110 IF P(X) > K THEN K = P(X)
120 NEXT X
130 R = 158.9 / K
140 FOR X = 1 TO P%
150 G%(X) = INT (159 - (P(X) * R))
160 NEXT
170 S% = INT (279 / P%)
200 REM **********
```



```
210 REM LINE GRAPH
220 REM **********
230 HGR
240 HCOLOR= 7
250 FOR X = 1 TO P%
260 HPLOT ((X * S%) - (S% - 1)),0 TO ((X * S%) - (S% - 1)),159
270 NEXT
280 HCOLOR= 3
290 HPLOT S% - (S% - 1),G%(1)
300 FOR X = 2 TO P%
310 Y% = G%(X)
320 HPLOT TO ((X * S%) - (S% - 1)),Y%
330 NEXT X
```

Horizontal Grid Lines

Now that the graph has vertical grid lines, you can insert horizontal ones as well. Since the horizontal values are relative to the maximum plot value, just use a standard measure between the horizontal lines. If you put 16 spaces between each line, that would roughly divide the vertical axis into ten groups. The following program generates ten random plots and places the horizontal bars across the screen in the same loop which generates the vertical grid bars.

Program 6-12.

```
 10 TEXT : HOME
 20 K = 0
 30 FOR X = 1 TO 10
 40 P(X) = INT ( RND (1) * (159))
 50 IF P(X) > K THEN K = P(X)
 60 NEXT X
 70 R = 158.9 / K
 80 FOR X = 1 TO 10
 90 G%(X) = INT (159 - (P(X) * R))
100 NEXT
110 S% = INT (279 / 10)
200 REM ************
210 REM X AND Y GRID
220 REM ************
```

```
230 HGR
240 HCOLOR= 7
250 FOR X = 1 TO 10
260 HPLOT ((X * S%) - (S% - 1)),0 TO ((X * S%) - (S% - 1)),159
270 HPLOT 1,X * 16 TO 279,X * 16 : REM HORIZONTAL LINES
280 NEXT X
290 HCOLOR= 3
300 HPLOT S% - (S% - 1),G%(1)
310 FOR X = 2 TO 10
320 Y% = G%(X)
330 HPLOT TO ((X * S%) - (S% - 1)),Y%
340 NEXT
```

Multiple Charts

There may be instances where you'll want to create several charts using the same set of data. For example, you may need a low-resolution bar chart with a high-resolution line chart. The next program shows how to use two types of charts and how to mix low-resolution and high-resolution graphics in the same program with the same set of data. Once all of the data are entered, and the charts are drawn, all you have to do is switch viewing screens to toggle the charts.

Program 6-13.

```
 10 TEXT
 20 HOME
 30 K = 0
 40 INVERSE
 50 PRINT " HOW MANY PLOTS? <MAX=15> ";
 60 NORMAL
 70 INPUT " ";P%
 80 DIM P(P%),G%(P%)
 90 FOR X = 1 TO P%
100 INPUT "PLOT VALUE ";P(X)
110 IF P(X) > K THEN K = P(X)
120 NEXT X
130 R = 158.9 / K
140 FOR X = 1 TO P%
150 G%(X) = INT (P(X) * R)
```



```
160 NEXT
170 S% = INT (279 / P%)
200 REM **************
210 REM HI-RES GRAPHICS
220 REM **************
230 HGR
240 HCOLOR= 3
250 FOR X = 1 TO P%
260 HPLOT ((X * S%) - (S% - 1)),0 TO ((X * S%) - (S% - 1)),159
270 NEXT
280 HCOLOR= 3
290 HPLOT S% - (S% - 1),159 - G%(1)
300 FOR X = 2 TO P%
310 Y% = 159 - G%(X)
320 HPLOT TO ((X * S%) - (S% - 1)),Y%
330 NEXT
340 WAIT - 16384,128 : POKE 49168,0
400 REM ***************
410 REM LO-RES GRAPHICS
420 REM ***************
430 R = 39.9 / K
440 FOR X = 1 TO P%
450 G%(X) = INT (P(X) * R)
460 GR
470 NEXT
480 FOR X = 1 TO P%
490 COLOR= X
500 P% = 40 - G%(X)
510 VLIN P%,39 AT X * 2
520 NEXT
530 WAIT - 16384,128
600 REM ***************
610 REM TOGGLE GRAPHICS
620 REM ***************
630 INVERSE
640 PRINT " PRESS ANY KEY TO TOGGLE SCREEN "
650 PRINT " PRESS 'Q' TO QUIT ";
660 NORMAL
670 WAIT - 16384,128: IF PEEK ( - 16368) = 209 OR PEEK ( -
    16368) = 241 THEN END
```

```
680 POKE 49239,0 : REM HI-RES
690 WAIT - 16384,128: POKE - 16368,0
700 POKE 49238,0 : REM LO-RES
710 GOTO 670
```

Circles

Drawing circles involves using an *algorithm,* or specified set of commands and techniques. Once you know the algorithm, it's a simple matter to place the circle anywhere you want on your high-resolution screen.

Let's get started—this following program draws a circle near the middle of your screen.

Program 6-14.

```
 10 TEXT : HOME
 20 HGR
 30 HCOLOR= 3
100 REM ****************
110 REM DEFINE PARAMETERS
120 REM ****************
130 RADIUS = 40
140 XSPOT = 100
150 YSPOT = 75
200 REM **********
210 REM DRAW CIRCLE
220 REM **********
230 FOR CIRCLE = 0 TO 6.3 STEP .007
240 X = RADIUS * COS (CIRCLE) + XSPOT
250 Y = (RADIUS / 4) * SIN (CIRCLE) / .3 + YSPOT
260 HPLOT X,Y
270 HPLOT TO X,Y
280 NEXT CIRCLE
290 WAIT - 16384,128
300 TEXT
```

The core of the algorithm is in lines 230–250, where the *x* and *y* values are calculated. To change the precision of the circle, and the speed at which it's drawn, change the step value in line 230.

Try changing it to .1 for a rapidly-drawn circle.

By making a few changes, you can control the size and placement of your circle.

Program 6-15.

```
 10 TEXT : HOME
 20 REM **********
 30 REM ENTER DATA
 40 REM **********
 50 INPUT "Radius ";RADIUS
 60 INPUT "Horizontal position ";XSPOT
 70 INPUT "Vertical position ";YSPOT
100 REM **********
110 REM DRAW CIRCLE
120 REM **********
130 HGR
140 HCOLOR= 3
150 FOR CIRCLE = 0 TO 6.3 STEP .007
160 X = RADIUS * COS (CIRCLE) + XSPOT
170 Y = (RADIUS / 4) * SIN (CIRCLE) / .3 + YSPOT
180 HPLOT X,Y
190 HPLOT TO X,Y
200 NEXT CIRCLE
210 WAIT - 16384,128
220 TEXT
```

To fill a circle with color, all that's required is a line from the center of the circle (XSPOT,YSPOT) to the side of the circle. All you need to change is the HPLOT TO after the HPLOT X,Y. Instead of using HPLOT TO X,Y from the last plot, change it to plot from the center of the circle to X,Y.

In addition, let's do something with color. Change the step on the loop to .0088 so that there will be about 720 elements in the loop (2 * 360) and then after 120 times through the loop, change the color. That will place all the colors, including black, in a circle segment. In turn, that should give you a hint as to how to make a pie chart. (Determine which proportion of the pie any single set of data requires, then make the pie portion to that size.)

Program 6-16.

```
 10 K = 1
 20 TEXT : HOME
 30 HGR
 40 REM ****************
 50 REM DEFINE PARAMETERS
 60 REM ****************
 70 RADIUS = 40
 80 XSPOT = 100
 90 YSPOT = 75
100 REM ****************
110 REM DRAW COLORED ARC
120 REM ****************
130 FOR CIRCLE = 0 TO 6.3 STEP .0088
140 C = C + 1: IF C = 120 THEN C = 0:K = K + 1
150 HCOLOR= K
160 X = RADIUS * COS (CIRCLE) + XSPOT
170 Y = (RADIUS / 4) * SIN (CIRCLE) / .3 + YSPOT
180 HPLOT X,Y
190 HPLOT XSPOT,YSPOT TO X,Y
200 NEXT CIRCLE
210 WAIT - 16384,128
220 TEXT
```

Line 190 draws from the center of the circle to the edge. Line 140 calculates when it's time to change colors.

Summary

One of the best possible programming exercises is one which includes both data and graphic representations of that data. Such exercies do two things.

First, they provide experience in working with graphics. Second, and more important, graphs and charts teach about translating data into new formats.

Chapter 7

Super High-Resolution Graphics

Super high-resolution graphics on your Apple IIGS is much like a good news/bad news joke.

The good news is that super high-resolution graphics are controlled by a set of routines built into your computer. Once you understand how to use these routines, collectively called the *Apple IIGS Toolbox,* it's relatively easy to do all sorts of things.

The bad news is that machine language—or a language such as C that gives more direct access to the system routines—is required to really work with the Toolbox. Quite simply, there is no easy way to use super high-resolution graphics from BASIC.

Let's Look

However, to get started and to give you something to look at in super high-resolution graphics, you'll see a BASIC program which creates a machine language program that uses the Apple IIGS's QuickDraw routines. What's more, you'll examine programs written in assembly language to see how to do it yourself.

You'll look at two programs. The first will be as simple as possible and will use the super high-resolution tools. The second will be a drawing program which uses the mouse. The first program shows you the fundamentals of programming at this level; the second shows you what's possible.

Simple Super High-Resolution

This first program draws two lines of different brush widths using separate colors for the lines and background color.

Type in the following BASIC program to see what super high-resolution graphics look like on your screen.

Program 7-1.

```
 10 TEXT : HOME
 20 FOR X = 0 TO 277
 30 READ D
 40 POKE X + 32768,D
 50 NEXT
 60 CALL 32768
100 DATA 32,88,252,24,251,194,48,244,0,0,244,0,0,244,225,0,244
110 DATA 0,32,162,2,26,34,0,0,225,104,133,6,133,157,104,133,8
120 DATA 133,159,160,0,0,169,0,0,151,157,200,200,151,157,162,1,2
130 DATA 34,0,0,225,162,3,2,34,0,0,225,244,0,0,244,0,16
140 DATA 162,3,32,34,0,0,225,104,141,128,2,244,0,0,162,2,2
150 DATA 34,0,0,225,104,244,0,134,244,0,0,244,0,0,244,64,1
160 DATA 244,0,0,244,200,0,173,128,2,72,162,6,2,34,0,0,225
170 DATA 244,0,135,244,0,0,244,0,0,173,128,2,72,162,4,2,34
180 DATA 0,0,225,244,5,0,162,4,55,34,0,0,225,244,5,0,244
190 DATA 5,0,162,4,44,34,0,0,225,244,119,119,162,4,21,34,0
200 DATA 0,225,244,100,0,244,0,0,162,4,60,34,0,0,225,244,11
210 DATA 0,162,4,55,34,0,0,225,244,10,0,244,10,0,162,4,44
220 DATA 34,0,0,225,244,100,0,244,128,0,162,4,60,34,0,0,225
230 DATA 56,251,32,12,253,24,251,194,48,162,4,3,34,0,0,225,244
240 DATA 0,0,162,3,33,34,0,0,225,162,6,3,34,0,0,225,162
250 DATA 3,3,34,0,0,225,162,2,3,34,0,0,225,162,1,3,34
260 DATA 0,0,225,56,251,96
```

Unless you know machine language programming using decimal values (instead of the more normal hexadecimal numbers), this program probably didn't make much sense. All it did was to POKE in a series of values and execute the program with CALL 32768, the beginning address of the values stored in memory.

You should see a horizontal green line across the top of the screen, a light blue vertical line, and an orange background.

To better see how this program works, here's a simple explanation of how to use the tools on your Apple IIGS, as well as a commented listing of the machine language source code for this simple graphics routine.

Use an Assembler

First, get a good assembler for the Apple IIGS. Merlin/816 (Roger Wagner Publishing, Inc.) was used for this example. Many Apple

programmers are familiar with Merlin or the Big Mac assembler family, and the listing should look familiar. The program is organized in blocks to make it easier to read.

The first step is to trick your Apple IIGS into giving up the handle that "owns" the super high-resolution screen. This lets you get an ID which can be used for your own programs. Once that's completed, you can begin your path to the QuickDraw II Toolbox. The path requires you to:

1. Start the Tool Locator.
2. Start miscellaneous Tools.
3. Get your ID and store it somewhere.
4. Start the Memory Manager.
5. Start the Event Manager.

Normally, the Memory Manager would be used to determine what area of memory to use. For this and the next example, $8600 was used, since it was simpler and more illustrative than a Memory Manager call for the same thing. In larger programs, use the Memory Manager—it takes care of it automatically.

In QuickDraw II

Once the path to QuickDraw II has been laid down, you're ready to start up QuickDraw. The following sequence does that.

1. Establish the beginning of direct page memory.
2. Set screen size (0 = visible screen).
3. Push your ID onto the stack and start QuickDraw.

Most of the work using the Toolbox requires that you push values onto the stack, then jump to the Toolbox ($E10000) with a long jump, JSL. The PEA is used to push parameters for various tools onto the stack.

The value of the specific tool is loaded into the X register; then you JSL to $E10000. If there's to be information returned on the stack with the particular tool being used, you have to push space onto the stack. This is done with PEA $0000.

Overall, the sequence is quite simple.

1. Push space onto the stack (if required).
2. Push parameters onto the stack (if required).
3. Load the X register in the immediate mode with the tool value.

4. **Jump to $E10000 (sometimes using a tool requires only the last two steps).**

Let's take a look at using the QuickDraw Toolbox by closely examining how the first line was drawn.

Set the pen color

1. **Push the color value ($05) onto the stack.**
2. **Load the X register with pen color setting routine number.**
3. **Jump to $E10000.**

Set the pen size

1. **Push the pen width on the stack.**
2. **Push the pen height on the stack.**
3. **Load the X register with the pen size tool number.**
4. **Jump to $E10000.**

Clear screen to background color

1. **Push background color onto stack (all values must be the same for a solid color background).**
2. **Load the X register with the background color tool number.**
3. **Jump to $E10000.**

Draw a line

1. **Push the ending X (horizontal) position of the line onto the stack.**
2. **Push the ending Y (vertical) position of the line onto the stack.**
3. **Load the X register with the line-to tool number.**
4. **Jump to $E10000.**

The same thing was done with the second line—by changing the values, the line's color, width, and direction were changed. All you have to do is insert some more drawing routines, and add more lines.

Program 7-2.

```
 1 ****************************************
 2 *                                      *
 3 *   Simple QuickDraw Lines             *
 4 *                                      *
 5 ****************************************
 6
 7 ID    EQU  $280
 8 TOOLS EQU  $E10000
 9
10       JSR  $FC58
11       XC
12       XC
13
14       CLC
15       XCE
16       REP  $30
17 ****************************************
18
19       PEA  $0000     ;Fix ID
20       PEA  $0000
21       PEA  $00E1
22       PEA  $2000
23       LDX  #$1A02
24       JSL  TOOLS
25 ****************************************
26       PLA             ;Set pointers
27       STA  $06
28       STA  $9D
29       PLA
30       STA  $08
31       STA  $9F
32 ****************************************
33       LDY  #$00       ;Reset handle
34       LDA  #$00
35       STA  [$9D],Y
36       INY
37       INY
38       STA  [$9D],Y
39
```

```
40 ****************************************
41 *                                      *
42 *       QuickDraw II Path              *
43 *                                      *
44 ****************************************
45         LDX   #$0201     ;Tool locator
46         JSL   TOOLS
47 ****************************************
48         LDX   #$0203     ;Start misc tools
49         JSL   TOOLS
50 ****************************************
51         PEA   $0000      ;Get ID from misc tools
52         PEA   $1000
53         LDX   #$2003
54         JSL   TOOLS
55         PLA              ;Pull ID off stack and put it
56         STA   ID         ;in ID
57 ****************************************
58         PEA   $0000
59         LDX   #$0202     ;Start memory manager
60         JSL   TOOLS
61         PLA
62 ****************************************
63         PEA   $8600      ;Start address for one page work area
64         PEA   $0000      ;Number of event records (0=20)
65         PEA   $0000      ;Minimum X clamp for mouse
66         PEA   320        ;Max X clamp for mouse
67         PEA   $00        ;Minimum Y clamp for mouse
68         PEA   200        ;Max Y clamp for mouse
69         LDA   ID         ;Get ID
70         PHA              ;Push it to the stack
71         LDX   #$0206     ;Event manager start up
72         JSL   TOOLS
73 ****************************************
74 *                                      *
75 *       Tools for Drawing              *
76 *                                      *
77 ****************************************
78
79         PEA   $8700      ;Begin direct page
```



```
80         PEA   $0000
81         PEA   $0000      ;Screen size
82         LDA   ID         ;Give ID
83         PHA
84         LDX   #$0204     ;QDStartup
85         JSL   TOOLS
86 ****************************************
87 *                                      *
88 *           First Line                 *
89 *                                      *
90 ****************************************
91         PEA   $05
92         LDX   #$3704     ;SetSolidPen Color
93         JSL   TOOLS
94 ****************************************
95         PEA   $0005      ;SetPenWidth
96         PEA   $0005      ;SetPenHeight
97         LDX   #$2C04     ;SetPenSize
98         JSL   TOOLS
99 ****************************************
100        PEA   $7777      ;Clear to scrn color
101        LDX   #$1504     ;Solid bkgnd requires
102        JSL   TOOLS      ;all 4 values to be same
103 ****************************************
104        PEA   $0064      ;X pos of line end
105        PEA   $0000      ;Y pos of line end
106        LDX   #$3C04     ;LineTo
107        JSL   TOOLS
108 ****************************************
109 *                                      *
110 *          Second Line                 *
111 *                                      *
112 ****************************************
113        PEA   $0B
114        LDX   #$3704     ;SetSolidPen Color
115        JSL   TOOLS
116 ****************************************
117        PEA   $000A      ;SetPenWidth
118        PEA   $000A      ;SetPenHeight
119        LDX   #$2C04     ;SetPenSize
```

```
120         JSL   TOOLS
121 ************************************
122         PEA   $0064     ;X pos of line end
123         PEA   $0080     ;Y pos of line end
124         LDX   #$3C04    ;LineTo
125         JSL   TOOLS
126 ************************************
127 *                                  *
128 *  Hold Screen Until Keypress      *
129 *                                  *
130 ************************************
131         SEC
132         XCE
133         JSR
134         CLC
135         XCE
136         REP   $30
137 ************************************
138 *                                  *
139 *     Back out of QuicKDraw        *
140 *                                  *
141 ************************************
142         LDX   #$0304    ;QUIT QD
143         JSL   TOOLS
144 ************************************
145         PEA   $0000     ;Drop ID
146         LDX   #$2103
147         JSL   TOOLS
148 ************************************
149         LDX   #$0306    ;Quit event manager
150         JSL   TOOLS
151 ************************************
152         LDX   #$0303    ;Quit misc tools
153         JSL   TOOLS
154 ************************************
155         LDX   #$0302    ;Quit memory manager
156         JSL   TOOLS
157 ************************************
158         LDX   #$0301    ;Quit tool locator
159         JSL   TOOLS
```



```
160 ************************************
161         SEC
162         XCE
163         RTS
```

If you don't have an assembler, you can enter the program with your monitor or mini-assembler. If you're using the monitor, just type in the address, a colon, and the machine language values following each address. For example, to type in the first three lines of the program below from the monitor, you'd enter:

```
*8000: 20 58 FC (press Return)
*8003: 18 (press Return)
*8004: FB (press Return)
. . . .
```

From the mini-assembler, there are two steps. First, it's necessary to set your processor, along with your registers and accumulator, to the 16-bit mode. Do the following, *using lowercase characters only.*

```
*0=e 0=x 0=m (press Return)
```

Next, enter the mini-assembler and type in the program:

```
*! (press Return)
!8000: jsr fc58 (press Return)
! clc (press Return)
! xce (press Return)
. . . .
```

After the first line, you don't have to keep entering the address; but be sure to put a space between the exclamation point prompt and the opcode (the three-letter combination). No space goes between the beginning address (8000) and the prompt.

It doesn't matter which process you use—monitor or mini-assembler. Once the program is typed in, save it with the following command.

BSAVE QD1,A$8000,L$116 (press Return)

When you want to run the program, just type **BRUN QD1**.

Program 7-3.

```
00/8000: 20 58 FC     JSR FC58
00/8003: 18           CLC
00/8004: FB           XCE
00/8005: C2 30        REP #30
00/8007: F4 00 00     PEA 0000
00/800A: F4 00 00     PEA 0000
00/800D: F4 E1 00     PEA 00E1
00/8010: F4 00 20     PEA 2000
00/8013: A2 02 1A     LDX #1A02
00/8016: 22 00 00 E1  JSL E10000
00/801A: 68           PLA
00/801B: 85 06        STA 06
00/801D: 85 9D        STA 9D
00/801F: 68           PLA
00/8020: 85 08        STA 08
00/8022: 85 9F        STA 9F
00/8024: A0 00 00     LDY #0000
00/8027: A9 00 00     LDA #0000
00/802A: 97 9D        STA [9D],Y
00/802C: C8           INY
00/802D: C8           INY
00/802E: 97 9D        STA [9D],Y
00/8030: A2 01 02     LDX #0201
00/8033: 22 00 00 E1  JSL E10000
00/8037: A2 03 02     LDX #0203
00/803A: 22 00 00 E1  JSL E10000
00/803E: F4 00 00     PEA 0000
00/8041: F4 00 10     PEA 1000
00/8044: A2 03 20     LDX #2003
00/8047: 22 00 00 E1  JSL E10000
00/804B: 68           PLA
00/804C: 8D 80 02     STA 0280
00/804F: F4 00 00     PEA 0000
00/8052: A2 02 02     LDX #0202
00/8055: 22 00 00 E1  JSL E10000
00/8059: 68           PLA
00/805A: F4 00 86     PEA 8600
00/805D: F4 00 00     PEA 0000
```



```
00/8060: F4 00 00     PEA 0000
00/8063: F4 40 01     PEA 0140
00/8066: F4 00 00     PEA 0000
00/8069: F4 C8 00     PEA 00C8
00/806C: AD 80 02     LDA 0280
00/806F: 48           PHA
00/8070: A2 06 02     LDX #0206
00/8073: 22 00 00 E1  JSL E10000
00/8077: F4 00 87     PEA 8700
00/807A: F4 00 00     PEA 0000
00/807D: F4 00 00     PEA 0000
00/8080: AD 80 02     LDA 0280
00/8083: 48           PHA
00/8084: A2 04 02     LDX #0204
00/8087: 22 00 00 E1  JSL E10000
00/808B: F4 05 00     PEA 0005
00/808E: A2 04 37     LDX #3704
00/8091: 22 00 00 E1  JSL E10000
00/8095: F4 05 00     PEA 0005
00/8098: F4 05 00     PEA 0005
00/809B: A2 04 2C     LDX #2C04
00/809E: 22 00 00 E1  JSL E10000
00/80A2: F4 77 77     PEA 7777
00/80A5: A2 04 15     LDX #1504
00/80A8: 22 00 00 E1  JSL E10000
00/80AC: F4 64 00     PEA 0064
00/80AF: F4 00 00     PEA 0000
00/80B2: A2 04 3C     LDX #3C04
00/80B5: 22 00 00 E1  JSL E10000
00/80B9: F4 0B 00     PEA 000B
00/80BC: A2 04 37     LDX #3704
00/80BF: 22 00 00 E1  JSL E10000
00/80C3: F4 0A 00     PEA 000A
00/80C6: F4 0A 00     PEA 000A
00/80C9: A2 04 2C     LDX #2C04
00/80CC: 22 00 00 E1  JSL E10000
00/80D0: F4 64 00     PEA 0064
00/80D3: F4 80 00     PEA 0080
00/80D6: A2 04 3C     LDX #3C04
```

```
00/80D9: 22 00 00 E1 JSL E10000
00/80DD: 38          SEC
00/80DE: FB          XCE
00/80DF: 20 0C FD    JSR FD0C
00/80E2: 18          CLC
00/80E3: FB          XCE
00/80E4: C2 30       REP #30
00/80E6: A2 04 03    LDX #0304
00/80E9: 22 00 00 E1 JSL E10000
00/80ED: F4 00 00    PEA 0000
00/80F0: A2 03 21    LDX #2103
00/80F3: 22 00 00 E1 JSL E10000
00/80F7: A2 06 03    LDX #0306
00/80FA: 22 00 00 E1 JSL E10000
00/80FE: A2 03 03    LDX #0303
00/8101: 22 00 00 E1 JSL E10000
00/8105: A2 02 03    LDX #0302
00/8108: 22 00 00 E1 JSL E10000
00/810C: A2 01 03    LDX #0301
00/810F: 22 00 00 E1 JSL E10000
00/8113: 38          SEC
00/8114: FB          XCE
00/8115: 60          RTS
```

Mouse QuickDraw

This next program is a simple super high-resolution drawing routine using the mouse. Certain key parameters have been isolated in the BASIC program so that you can easily change the background color, pen colors, and pen size.

Change *only* the first and last DATA values of line 192 to alter the height and width of your pen. (If you change any other values, the program will crash.) See how fine or how fat you can make the pen.

Program 7-4.

```
10 TEXT : HOME
20 FOR X = 0 TO 303
30 READ QD
35 K = K + 1
40 POKE 32768 + X,QD
50 NEXT
60 CALL 32768
100 DATA 32,88,252,24,251,194,48,244,0,0,244,0,0,244,225,0
110 DATA 244,0,32,162,2,26,34,0,0,225,104,133,6,133,157,104
120 DATA 133,8,133,159,160,0,0,169,0,0,151,157,200,200,151,157
130 DATA 162,1,2,34,0,0,225,162,3,2,34,0,0,225,244,0
140 DATA 0,244,0,16,162,3,32,34,0,0,225,104,141,128,2,244
150 DATA 0,0,162,2,2,34,0,0,225,104,244,0,133,244,0,0
160 DATA 244,0,0,244,64,1,244,0,0,244,200,0,173,128,2,72
170 DATA 162,6,2,34,0,0,225,244,0,134,244,0,0,244,0,0
180 DATA 173,128,2,72,162,4,2,34,0,0,225,244
185 DATA 9: REM PEN COLOR
190 DATA 0,162,4,55,34,0,0,225,244
192 DATA 9,0,244,9: REM FIRST AND LAST VALUES = PEN X
    AND Y
194 DATA 0,162,4,44,34,0,0,225,244
200 DATA 119,119: REM BACKGROUND COLOR
205 DATA 162,4,21,34,0,0,225,162,4,145,34
210 DATA 0,0,225,244,0,0,162,3,24,34,0,0,225,244,1,0
220 DATA 162,3,25,34,0,0,225,244,0,0,244,0,0,244,0,0
230 DATA 162,3,23,34,0,0,225,104,104,141,2,2,104,141,0,2
240 DATA 173,0,2,72,173,2,2,72,162,4,60,34,0,0,225,173
250 DATA 0,2,201,255,0,208,208,56,251,32,12,253,24,251,194,48
260 DATA 162,4,3,34,0,0,225,244,0,0,162,3,33,34,0,0
270 DATA 225,162,6,3,34,0,0,225,162,3,3,34,0,0,225,162
280 DATA 2,3,34,0,0,225,162,1,3,34,0,0,225,56,251,96
```

Here's the commented source code.

Program 7-5.

```
1 ****************************************
2 *                                      *
3 *     QUICKDRAW MOUSE                  *
4 *                                      *
5 ****************************************
6
7 ID     EQU   $280
8 TOOLS  EQU   $E10000
9 XPOS   EQU   $200
```

```
10 YPOS   EQU   $202
11        JSR   $FC58
12        XC
13        XC
14
15        CLC
16        XCE
17        REP   $30
18 ***************************************
19 *                                     *
20 *   Recover ID Ownership              *
21 *   So that program can run           *
22 *   from BASIC.SYSTEM                 *
23 *                                     *
24 ***************************************
25
26        PEA   $0000     ;Find Handle
27        PEA   $0000
28        PEA   $00E1
29        PEA   $2000
30        LDX   #$1A02
31        JSL   TOOLS
32 ***************************************
33        PLA             ;Set pointers
34        STA   $06
35        STA   $9D
36        PLA
37        STA   $08
38        STA   $9F
39 ***************************************
40        LDY   #$00      ;Reset handle
41        LDA   #$00
42        STA   [$9D],Y
43        INY
44        INY
45        STA   [$9D],Y
46
47 ***************************************
48 *                                     *
```



```
49 *   Path to QuickDraw II          *
50 *   Notice that this is the       *
51 *   same as the first prgm        *
52 *                                 *
53 ***************************************
54        LDX   #$0201    ;Tool locator
55        JSL   TOOLS
56 ***************************************
57        LDX   #$0203    ;Start misc tools
58        JSL   TOOLS
59 ***************************************
60        PEA   $0000     ;Get ID from misc tools
61        PEA   $1000
62        LDX   #$2003
63        JSL   TOOLS
64        PLA             ;Pull ID off stack and put it
65        STA   ID        ;in ID
66 ***************************************
67        PEA   $0000
68        LDX   #$0202    ;Start memory manager
69        JSL   TOOLS
70        PLA
71 ***************************************
72        PEA   $8500     ;address for one page work area
73        PEA   $0000     ;Number of event records (0=20)
74        PEA   $0000     ;Minimum X clamp for mouse
75        PEA   320       ;Max X clamp for mouse
76        PEA   $00       ;Minimum Y clamp for mouse
77        PEA   200       ;Max Y clamp for mouse
78        LDA   ID        ;Get ID
79        PHA             ;Push it to the stack
80        LDX   #$0206    ;Event manager start up
81        JSL   TOOLS
82 .....................................
83 *                                   *
84 *      QuickDraw II Tools           *
85 *                                   *
86 .....................................
87
88        PEA   $8600     ;Direct pagespace
```

```
        PEA   $0000
        PEA   $0000     ;screen size
        LDA   ID        ;Give ID
        PHA
        LDX   #$0204    ;QDStartup
        JSL   TOOLS
*************************************
        PEA   $0B
        LDX   #$3704    ;SetSolidPen Color
        JSL   TOOLS
*************************************
        PEA   $0004     ;SetPenWidth
        PEA   $0002     ;SetPenHeight
        LDX   #$2C04    ;SetPenSize
        JSL   TOOLS
*************************************
        PEA   $4444     ;Clear to color
        LDX   #$1504
        JSL   TOOLS
*************************************
        LDX   #$9104    ;Show cursor
        JSL   TOOLS
*************************************
        PEA   $0000
        LDX   #$1803
        JSL   TOOLS     ;InitMouse
*************************************
        PEA   $0001
        LDX   #$1903
        JSL   TOOLS     ;SetMouse
*************************************
MOUSE   PEA   $0000     ;ReadMouse
        PEA   $0000
        PEA   $0000
        LDX   #$1703
        JSL   TOOLS
        PLA
        PLA
        STA   YPOS
```



```
        PLA
        STA   XPOS
*************************************
        LDA   XPOS
        PHA             ;X pos of line end
        LDA   YPOS
        PHA             ;Y pos of line end
        LDX   #$3C04    ;LineTo
        JSL   TOOLS
*************************************
        LDA   XPOS
        CMP   #$FF
        BNE   MOUSE
*************************************
*                                   *
* It is important to                *
* remember to switch to             *
* emulation mode when using         *
* non-toolbox routines              *
*                                   *
*************************************
        SEC
        XCE
        JSR   $FD0C
        CLC
        XCE
        REP   $30
*************************************
*                                   *
*       Exit QuicKDraw              *
*                                   *
*************************************
        LDX   #$0304    ;QUIT QD
        JSL   TOOLS
*************************************
        PEA   $0000     ;Drop ID
        LDX   #$2103
        JSL   TOOLS
*************************************
```

```
167         LDX   #$0306     ;Quit event manager
168         JSL   TOOLS
169 ***************************************
170         LDX   #$0303     ;Quit misc tools
171         JSL   TOOLS
172 ***************************************
173         LDX   #$0302     ;Quit MM
174         JSL   TOOLS
175 ***************************************
176         LDX   #$0301     ;Quit TL
177         JSL   TOOLS
178 ***************************************
179         SEC
180         XCE
181         RTS
```

And here's the listing you'll use if you're entering the program with the monitor or mini-assembler.

Program 7-6.

```
00/8000: 20 58 FC     JSR FC58
00/8003: 18           CLC
00/8004: FB           XCE
00/8005: C2 30        REP #30
00/8007: F4 00 00     PEA 0000
00/800A: F4 00 00     PEA 0000
00/800D: F4 E1 00     PEA 00E1
00/8010: F4 00 20     PEA 2000
00/8013: A2 02 1A     LDX #1A02
00/8016: 22 00 00 E1  JSL E10000
00/801A: 68           PLA
00/801B: 85 06        STA 06
00/801D: 85 9D        STA 9D
00/801F: 68           PLA
00/8020: 85 08        STA 08
00/8022: 85 9F        STA 9F
00/8024: A0 00 00     LDY #0000
00/8027: A9 00 00     LDA #0000
00/802A: 97 9D        STA [9D],Y
00/802C: C8           INY
```



```
00/802D: C8           INY
00/802E: 97 9D        STA [9D],Y
00/8030: A2 01 02     LDX #0201
00/8033: 22 00 00 E1  JSL E10000
00/8037: A2 03 02     LDX #0203
00/803A: 22 00 00 E1  JSL E10000
00/803E: F4 00 00     PEA 0000
00/8041: F4 00 10     PEA 1000
00/8044: A2 03 20     LDX #2003
00/8047: 22 00 00 E1  JSL E10000
00/804B: 68           PLA
00/804C: 8D 80 02     STA 0280
00/804F: F4 00 00     PEA 0000
00/8052: A2 02 02     LDX #0202
00/8055: 22 00 00 E1  JSL E10000
00/8059: 68           PLA
00/805A: F4 00 85     PEA 8500
00/805D: F4 00 00     PEA 0000
00/8060: F4 00 00     PEA 0000
00/8063: F4 40 01     PEA 0140
00/8066: F4 00 00     PEA 0000
00/8069: F4 C8 00     PEA 00C8
00/806C: AD 80 02     LDA 0280
00/806F: 48           PHA
00/8070: A2 06 02     LDX #0206
00/8073: 22 00 00 E1  JSL E10000
00/8077: F4 00 86     PEA 8600
00/807A: F4 00 00     PEA 0000
00/807D: F4 00 00     PEA 0000
00/8080: AD 80 02     LDA 0280
00/8083: 48           PHA
00/8084: A2 04 02     LDX #0204
00/8087: 22 00 00 E1  JSL E10000
00/808B: F4 0B 00     PEA 000B
00/808E: A2 04 37     LDX #3704
00/8091: 22 00 00 E1  JSL E10000
00/8095: F4 04 00     PEA 0004
00/8098: F4 02 00     PEA 0002
00/809B: A2 04 2C     LDX #2C04
00/809E: 22 00 00 E1  JSL E10000
```

```
00/80A2: F4 33 33     PEA 3333
00/80A5: A2 04 15     LDX #1504
00/80A8: 22 00 00 E1  JSL E10000
00/80AC: A2 04 91     LDX #9104
00/80AF: 22 00 00 E1  JSL E10000
00/80B3: F4 00 00     PEA 0000
00/80B6: A2 03 18     LDX #1803
00/80B9: 22 00 00 E1  JSL E10000
00/80BD: F4 01 00     PEA 0001
00/80C0: A2 03 19     LDX #1903
00/80C3: 22 00 00 E1  JSL E10000
00/80C7: F4 00 00     PEA 0000
00/80CA: F4 00 00     PEA 0000
00/80CD: F4 00 00     PEA 0000
00/80D0: A2 03 17     LDX #1703
00/80D3: 22 00 00 E1  JSL E10000
00/80D7: 68           PLA
00/80D8: 68           PLA
00/80D9: 8D 02 02     STA 0202
00/80DC: 68           PLA
00/80DD: 8D 00 02     STA 0200
00/80E0: AD 00 02     LDA 0200
00/80E3: 48           PHA
00/80E4: AD 02 02     LDA 0202
00/80E7: 48           PHA
00/80E8: A2 04 3C     LDX #3C04
00/80EB: 22 00 00 E1  JSL E10000
00/80EF: AD 00 02     LDA 0200
00/80F2: C9 FF 00     CMP #00FF
00/80F5: D0 D0        BNE 80C7 {-30}
00/80F7: 38           SEC
00/80F8: FB           XCE
00/80F9: 20 0C FD     JSR FD0C
00/80FC: 18           CLC
00/80FD: FB           XCE
00/80FE: C2 30        REP #30
00/8100: A2 04 03     LDX #0304
00/8103: 22 00 00 E1  JSL E10000
00/8107: F4 00 00     PEA 0000
```



```
00/810A: A2 03 21     LDX #2103
00/810D: 22 00 00 E1  JSL E10000
00/8111: A2 06 03     LDX #0306
00/8114: 22 00 00 E1  JSL E10000
00/8118: A2 03 03     LDX #0303
00/811B: 22 00 00 E1  JSL E10000
00/811F: A2 02 03     LDX #0302
00/8122: 22 00 00 E1  JSL E10000
00/8126: A2 01 03     LDX #0301
00/8129: 22 00 00 E1  JSL E10000
00/812D: 38           SEC
00/812E: FB           XCE
00/812F: 60           RTS
```

Summary

This chapter just begins to touch on the power of the QuickDraw II routines and super high-resolution graphics. An entire book could easily be written on using just the QuickDraw routines; volumes would be required to explain how to use all of the Apple IIGS Toolbox routines. For the time being, though, this should be enough to launch you on a discovery path of QuickDraw II and super high-resolution graphics.

To work most efficiently with these powerful new Apple IIGS tools, it's strongly recommended that you learn machine language programming on the 65816 microprocessor. In the meantime, though, you can enjoy seeing your own creations with the simple programs provided.

Chapter 8

Sound and Music on the Apple IIGS

You can create two types of sound on your Apple IIGS. The first kind of sound clicks the speaker more or less directly by accessing a special address in your computer. A wide range of tones, sounds, and music can be generated by timing the speaker clicks.

The second kind of sound is produced the Apple IIGS Toolbox and the Ensoniq sound chip. This type of sound is extremely powerful when accessed through assembly language programs in ProDOS 16. Unfortunately, it's difficult to access while the BASIC.SYSTEM is in memory. What's more, it requires a development assembler—the current assembler requires a megabyte of memory and prefers that a hard disk drive be connected. Because of the elaborate hardware and software needed to access the sound elements through the Toolbox, let's concentrate on sound generation by tweaking the speaker.

Speaker Tweaking

Imagine that the speaker inside your IIGS is a diaphram. It's either out (full of air) or in (devoid of air.) Each time the diaphragm pulls in or pushes out, it makes a sound, just as people do when they exaggerate inhaling or exhaling. The speaker is just a paper cone that pops in and out, making a click each time it pops.

To get the speaker to pop one way or the other (and thus make a sound), all you need do is to access address $C030 (49200). The easiest way to do that is to define a variable as the contents of 49200. For example, do the following:

```
P=PEEK(49200) (press Return)
```

You should have heard a click when you pressed the Return key. By controlling the timing of the clicks, you can make different sounds.

For example, the following program clicks the speaker 30 times for a buzzer sound.

Program 8-1.

```
10 TEXT : HOME
20 FOR X=1 TO 30
30 P=PEEK(49200)
40 NEXT X
```

To change that sound to something a bit different, install a delay loop between the times the speaker is clicked.

Program 8-2.

```
10 TEXT : HOME
20 FOR X=1 TO 30
30 P=PEEK(49200)
40 FOR PAUSE = 1 TO 40
50 NEXT PAUSE
60 NEXT X
```

That loop changed the sound to something akin to a deck of cards being thumbed.

To find the range of clicks which can be generated by a pause loop, generate a variable-length pause based on the number of times the initial loop is run. Here's how.

Program 8-3.

```
10 TEXT : HOME
20 FOR X=1 TO 100
30 P=PEEK(49200)
40 REM ************
50 REM VARIABLE LOOP
60 REM ************
70 FOR PAUSE = 1 TO X
80 NEXT PAUSE
90 NEXT X
```


As you can see, depending on how the speaker tweaking is spaced, different sounds emerge.

Speed and Sound Control

Depending on whether you have your system speed set to Fast or Normal on your Control Panel, different sounds will emerge. Check your Control Panel to see what speed your IIGS is set at now. In case you don't remember how to do that, press and hold down the Open Apple and Control keys, then press the Esc key. When the Desk Accessories window appears, move the cursor using the arrow keys. Once in the Control Panel window, select *System Speed* using the arrow keys again, and press Return. The default condition is Fast speed, but whatever it is, press the right arrow key to toggle it to the opposite. Press Return and then back out of the Control Panel and Desk Accessories by choosing the Quit option.

Once you're back to BASIC, run the last program once again. If the computer was set on Fast, and now is on Slow, the sound will be lower. If the computer is now on Fast, the sound will be higher.

In addition to changing the system speed in the Control Panel, you can also change the volume and pitch. Try different volume levels and pitches.

Making a Racket

To really work with sound, it helps to use assembly language programs. Machine language routines give you finer control since they run a good deal faster than programs written in BASIC. The following programs are simple and serve to illustrate different effects. They're set to exit to BASIC as soon as you press any key.

Alarm or Alien?

The first two sets of programs illustrate how different system speeds create different sound effects. The first program sounds like a European police car if the system speed is set to Normal, but

sounds like a space ship landing if the speed is set to Fast. (Since these programs are so short, the monitor or mini-assembler can be used to enter them. Of course, you may also use the accompaning BASIC listing instead.)

Program 8-4.

```
00/0300: AC FF 02   LDY 02FF
00/0303: AD 30 C0   LDA C030
00/0306: 88         DEY
00/0307: D0 FD      BNE 0306 {-03}
00/0309: CE FF 02   DEC 02FF
00/030C: AD 00 C0   LDA C000
00/030F: C9 80      CMP #80
00/0311: 90 ED      BCC 0300 {-13}
00/0313: 60         RTS
```

Program 8-5.

```
 10 TEXT : HOME
 20 REM *****
 30 REM ALARM
 40 REM *****
 50 FOR X = 0 TO 19
 60 READ D
 70 POKE 768 + X,D
 80 NEXT
 90 CALL 768
100 DATA 172,255,2,173,48,192,136,208,253,206,255,2,173,0,192,
    201,128,144,237,96
```

Since so many sound effects and routines have been written for earlier versions of the Apple II, you may want to change them so that they sound right using the Apple IIGS with a fast system speed. The following program shows how to put a delay loop in the above program to slow it down so that when it's run with a Fast system speed, it sounds similar to the first Alarm program with a slow system speed.

Program 8-6.

```
00/0300: AC FF 02   LDY 02FF
00/0303: AD 30 C0   LDA C030
00/0306: 88         DEY
```



```
00/0307: D0 FD      BNE 0306 {-03}
00/0309: CE FF 02   DEC 02FF
00/030C: A2 FF      LDX #FF       <Delay loop begin
00/030E: CA         DEX
00/030F: D0 FD      BNE 030E {-03}<Delay loop end
00/0311: AD 00 C0   LDA C000
00/0314: C9 80      CMP #80
00/0316: 90 E8      BCC 0300 {-18}
00/0318: 60         RTS
```

Program 8-7.

```
 10 TEXT : HOME
 20 REM ******
 30 REM ALARM2
 40 REM ******
 50 FOR X = 0 TO 24
 60 READ D
 70 POKE 768 + X,D
 80 NEXT
 90 CALL 768
100 DATA 172,255,2,173,48,192,136,208,253,206,255,2,162,255,
    202,208,253,173,0,192,201,128,144,232,96
```

Sound Tricks

Since the basic way of creating sound is to vary the speed and frequency of speaker tweaking, one trick is to pick an address in memory and then use it as an offset for generating random values. For example, a well-known address among assembly language programmers on the Apple II series is $FC58. That address is the beginning of a routine which clears the screen and homes the cursor. By using it, or some other address where there's an assured collection of different values, it's possible to create a "saw-tooth" type of wave form for sound effects.

This next program uses $FC58 as an offset to generate a sound resembling that of a jackhammer. There are three loops in this program. By using the X and Y registers along with an address (00 was used in this example), it's possible to decrement (or increment) three values at once. The accumulator is busy tweaking the speaker with LDA $C030, so it's out of commission as an added index register.

Program 8-8.

```
00/0300: A2 F0      LDX #F0
00/0302: A9 0C      LDA #0C
00/0304: 85 00      STA 00
00/0306: AD 30 C0   LDA C030
00/0309: BC 58 FC   LDY FC58,X
00/030C: 88         DEY
00/030D: D0 FD      BNE 030C {-03}
00/030F: CA         DEX
00/0310: D0 F4      BNE 0306 {-0C}
00/0312: C6 00      DEC 00
00/0314: D0 F0      BNE 0306 {-10}
00/0316: 60         RTS
```

Program 8-9.

```
 10 TEXT : HOME
 20 REM **********
 30 REM JACKHAMMER
 40 REM **********
 50 FOR X = 0 TO 22
 60 READ D
 70 POKE 768 + X,D
 80 CALL 768
100 DATA 162,240,169,12,133,0,173,48,192,188,88,252,136,208,
    253,202,208,244,198,0,208,240,96
```

Experiment with different loops and values to see what other sounds you can create on your own.

Musical Tones

Making musical notes and music on your Apple IIGS requires that you find the right combination of *loops*. The primary loops you have to establish are:

- Pitch
- Duration

The pitch loop is the sound produced and represents the inside loop. The number of times the pitch loop is repeated is the duration. Thus, the duration loop is the outside loop. The longer the duration, the more times the pitch loop is repeated.

This next program is a combination of BASIC and machine language. The machine language program is automatically generated with the BASIC DATA statements, but it's important to see how it works. The zero page addresses $OD and $OE are used to store the duration and pitch values respectively. The duration value is loaded into the X register and the pitch value into the Y register. The contents of $OD and $OE are POKEd in from the BASIC program, controlled by an INPUT statement that allows you to control the duration and pitch of each note. Using this program and a piano or some other comparative instrument, you can create an entire musical scale. Just keep putting in different values until you get the right pitch, and then record the value you used to get that pitch. Later you can use it in a program to recreate a song if you want. (The program will produce sounds about as high as you want, but for lower pitches, change the system speed from Fast to Normal.)

Program 8-10.

```
00/0300: A6 0D      LDX 0D      <Duration value stored here
00/0302: A4 0E      LDY 0E      <Pitch value stored here
00/0304: AD 30 C0   LDA C030
00/0307: 88         DEY
00/0308: D0 FD      BNE 0307 {-03} <Inside Loop Terminal
00/030A: CA         DEX
00/030B: D0 F5      BNE 0302 {-0B} <Outside Loop Terminal
00/030D: 60         RTS
```

Program 8-11.

```
 10 TEXT : HOME
 20 FOR X = 0 TO 13
 30 READ M
 40 POKE 768 + X,M
 50 NEXT
 60 INPUT "Duration ";DUR
 70 INPUT "Pitch ";P
 80 IF P> 255 THEN 70
 90 POKE 13,DUR
100 POKE 14,P
110 CALL 768
```

```
120 IF P = 0 THEN END
130 GOTO 70
140 DATA 166,13,164,14,173,48,192,136,208,253,202,208,245,96
```

This next program shows how to translate the pitch values into keyboard notes. Beginning with middle C in a single octave, this next program shows how the notes C–B (C,D,E,F,G,A,B) can be generated from the keyboard. Use the Normal (slow) system speed for this program and be sure to press the Caps Lock key so that only uppercase letters will be read.

Program 8-12.

```
 10 TEXT : HOME
 20 FOR X = 0 TO 13
 30 READ M
 40 POKE 768 + X,M
 50 NEXT X
 60 DUR = 255
 70 INPUT "Note A-G <Q> to quit ";N$
 80 IF N$ = "A" THEN N = 113
 90 IF N$ = "B" THEN N = 101
100 IF N$ = "C" THEN N = 191
110 IF N$ = "D" THEN N = 168
120 IF N$ = "E" THEN N = 151
130 IF N$ = "F" THEN N = 145
140 IF N$ = "G" THEN N = 128
150 POKE 14,N
160 CALL 768
170 IF N$ = "Q" THEN END
180 GOTO 70
200 DATA 166,13,164,14,173,48,192,136,208,253,202,208,245,96
300 PRINT X
```

Using the Keyboard to Generate Pitch and Duration

Since all keys have an associated ASCII value, it's possible to use the keyboard to make a musical instrument. Just about everyone who has ever played with the sound routines on the Apple has made one version or another of this next program. Called the

Cheap Organ, the program emits the pitch of the note generated by the ASCII value of the key pressed. The higher the ASCII value, the lower the note. Since uppercase letters have lower ASCII values than lowercase ones, the lower notes are generated by lowercase characters. (Just remember *lower* notes and lowercase.) The notes will continue until another key is pressed—that's why it sounds something like an organ. Press the Esc key to exit the program, and use the Normal setting on the system speed for a fuller range of low notes and Fast for a fuller range of high notes.

Program 8-13.

```
 10 TEXT : HOME : GOSUB 200
 20 REM **********
 30 REM CHEAP ORGAN
 40 REM **********
 50 FOR X = 0 TO 20
 60 READ MUSIC
 70 POKE X + 768,MUSIC
 80 NEXT
 90 CALL 768
100 DATA 172,0,192,152,170,224,155,240,11,174,48
110 DATA 192,136,192,1,208,251,76,0,3,96
120 END
200 S$ = "PRESS ESC TO END"
210 HTAB 20 - LEN (S$) / 2
220 PRINT S$
230 RETURN
```

Program 8-14.

```
00/0300: AC 00 C0   LDY C000
00/0303: 98         TYA
00/0304: AA         TAX
00/0305: E0 9B      CPX #9B
00/0307: F0 0B      BEQ 0314 {+0B}
00/0309: AE 30 C0   LDX C030
00/030C: 88         DEY
00/030D: C0 01      CPY #01
00/030F: D0 FB      BNE 030C {-05}
00/0311: 4C 00 03   JMP 0300
00/0314: 60         RTS
```

Mixing Sound and Animated Graphics

Anyone who ever played an arcade game will recognize the importance of combining sound and graphics. The trick is to coordinate movement with sound so that one seems to go with the other. For example, this next program creates a yellow beam that looks and sounds like it's drilling its way across the screen.

Program 8-15.

```
10 GR
20 COLOR= 13
30 FOR X = 0 TO 39
40 FOR V=1 TO 20
50 P = PEEK (49200)
60 NEXT V
70 PLOT X,20
80 NEXT X
```

Change the value of the delay loop in line 40 and you can get sounds ranging from that of a ray gun to the pecking of a bird. Not only does the loop affect the nature of the sound, it affects the animation of the line as well.

With a few more changes, you can make a single animated character racing across the screen. The delay loop holds the major character on the screen a bit longer.

Program 8-16.

```
 10 GR
 20 FOR X = 1 TO 30
 30 FOR V = 1 TO 12
 40 P = PEEK(49200)
 50 NEXT V
 60 COLOR= 0
 70 PLOT X - 1,20
 80 COLOR= 13
 90 PLOT X, 20
100 NEXT X
```

By changing the value of a single loop, not only do you change the speed of the animation, but you also change the sound.

For bigger projects, more planning and care is required. An intermediate level of animated programming can be found in double-low-resolution graphics. There's lots of color and somewhat better resolution than simple low-resolution.

This next program simulates an ambulance driving through the night with its siren on. Although its movement is slower and jerkier than you'd see in a commercial program, it gives you an idea of how to meld complex animated graphics with sound. (It would be a lot smoother and faster if it were written in machine language.)

Program 8-17.

```
 10 TEXT : HOME
 20 GR
 30 POKE 49246,0: REM DOUBLE-LO-RES
 40 FOR D = 0 TO 44
 50 READ V
 60 POKE 768 + D,V
 70 NEXT
100 REM **************
110 REM DATA FOR SOUND
120 REM **************
130 DATA 127,0,173,48,192,136,208,5,206,1,3,240,9,202,208,245
140 DATA 174,0,3,76,2,3,255,0,173,48,192,136,208,5,206,23
150 DATA 3,240,9,202,208,245,174,22,3,76,24,3,96
200 REM ************************
210 REM ANIMATED NIGHT AMBULANCE
220 REM ************************
230 FOR X = 1 TO 59
240 COLOR= 0
250 PLOT X,32
260 COLOR= 15
270 HLIN X + 1,X + 9 AT 32
280 COLOR= 0
290 PLOT X,33
300 COLOR= 15
310 HLIN X + 1,X + 8 AT 33
320 COLOR= 9
330 PLOT X + 9,33
340 COLOR= 0
350 PLOT X,34
```

```
360 COLOR= 15
370 HLIN X + 1,X + 5 AT 34
380 COLOR= 11
390 PLOT X + 6,34
400 COLOR= 15
410 HLIN X + 7,X + 8 AT 34
420 COLOR= 9
430 PLOT X + 9,34
440 COLOR= 0
450 PLOT X,35
460 COLOR= 15
470 HLIN X + 1,X + 4 AT 35
480 COLOR= 11
490 HLIN X + 5,X + 7 AT 35
500 COLOR= 15
510 HLIN X + 8,X + 12 AT 35
520 COLOR= 13
530 HLIN X + 13,X + 20 AT 35
540 COLOR= 0
550 PLOT X,36
560 COLOR= 15
570 HLIN X + 1,X + 5 AT 36
580 COLOR= 11
590 PLOT X + 6,36
600 COLOR= 15
610 HLIN X + 7,X + 12 AT 36
620 COLOR= 0
630 PLOT X,37
640 COLOR= 15
650 PLOT X + 1,37
660 COLOR= 7
670 HLIN X + 2,X + 3 AT 37
680 COLOR= 15
690 HLIN X + 4,X + 8 AT 37
700 COLOR= 7
710 HLIN X + 9,X + 10 AT 37
720 COLOR= 15
730 HLIN X + 11,X + 12 AT 37
740 COLOR= 0
750 PLOT X + 1,38
```



```
760 COLOR= 7
770 HLIN X + 2,X + 3 AT 38
780 COLOR= 0
790 HLIN X + 4,X + 8 AT 38
800 COLOR= 7
810 HLIN X + 9,X + 10 AT 38
900 REM **************
910 REM CALL THE SOUND
920 REM **************
930 CALL 768
940 NEXT X
```

That should be enough to give you a start with sound and some different things you can do with it.

Using the Sound Tools

This next example is long and somewhat complex, but it will provide a beginning to using the full power of your Apple IIGS. To start with, there's no way that this program can be run from the BASIC.SYSTEM environment. It requires one 800K drive and a one-megabyte RAM card, in addition to a second disk drive (either 5¼- or 3½-inch drive). Alternatively, a hard drive and a one megabyte RAM card will suffice. All of the RAM must be allocated to the program, none may be used for a RAM drive.

This example was created using the Apple IIGS Programmer's Workshop Version 1.0 from Byte Works, Inc. and Apple Computer, Inc. If you don't have that assembler, you may have to make some adjustments.

The program has extensive comments, but so that you'll better understand what's going on, let's go over some of the key features.

First of all, this is a speaker tweaker program, but as you'll see when you run it, it does things with the speaker that are truely astounding. That's because it uses DOC, the Digital Oscillator Chip. It uses the system tools, which must be on a disk in the system when you run the program. In particular, it requires Tool #025. (It's over 250 lines, and without the tools, it would be even longer.) Like the QuickDraw II Toolbox mentioned in Chapter 7, there's a

different protocol required to write programs which use the tools. However, there are several more tools in QuickDraw II than there are in the Sound Toolbox. Also, there are more data elements in this program (DC is something like DATA in BASIC.)

You can change the notes by changing the values in the DC statements beginning with those labeled TRK1. The first value is the note itself, and the second value is the duration of the note. For example

DC I2'$5F,$8000'

indicates two integer values, the first (note) $5F and the second (duration) $8000. When you first run the program, listen carefully how the notes rise at an increasing speed and then accelerate more when they fall. That's because the notes begin with a higher value—$FFFF—than all the others. Further down the list, the values lower to $C000, $8000, and finally $4000. Try changing the values to see what notes and note lengths you can create.

For the sound of the note, you can change the waveform and other elements that affect the sound, but this is a little trickier, and unless you have a full understanding of using the APW assembler (or the *Orca* assembler), you might crash the program in doing so. If you want to, though, keep it simple and change the decay rate or something else that merely involves changing a single value in the program. Then if it bombs, you can easily repair it.

Finally, as a suggestion for debugging and using APW assembler, save two copies of your program. Save one as TUNE.S and as second as TUNE. When you assemble your program, enter

ASML TUNE (press Return)

If it indicates an error, first DELETE TUNE, then EDIT TUNE.S and after debugging it, save the two programs again under the two respective names. (Note: At the beginning of the program, it reads *KEEP TUNE*. If you want to try different variations, change TUNE to TUNE1, TUNE2, and so on, so that you can test different arrangements of the program. Save the program you wish to assemble and link under the same name as the KEEP name.)

Program 8-18.

```
********* **          Speed Scale          **
************************************************
            KEEP    TUNE
            MSB     ON
MAIN        START
P16         EQU     $E100A8             ; P16
            PHK
            PLB
            LDX     #$0201              ; Tool Locator
            JSL     $E10000
            PEA     $0000               ; Start memory manager
            LDX     #$0202
            JSL     $E10000
            PLA                         ; Get ID
            STA     ID                  ; stick it somewhere
            LDX     #$0203              ; Start misc. tools
            JSL     $E10000
            PEA     TOOLTABL+16         ; Tooltable bank
            PEA     TOOLTABL            ; tooltable address
            LDX     #$0E01              ; Get the tools
            JSL     $E10000
            PEA     $0000               ; Results space
            PEA     $0000               ; Block size
            PEA     $0000               ; $000100 - Space required
            PEA     $0100               ; 1 page required
            LDA     ID                  ; Push your ID
            PHA                         ; onto the stack
            PEA     $C001               ; MemAttributes - locked; fixed
            PEA     $0000               ; Use zero bank
            PEA     $0000               ; 0 = Let MemMan handle it
            LDX     #$0902              ; NewHandle
            JSL     $E10000
            PLA                         ; TheHandle - bank
            STA     HNDL                ; Put it in HNDL
            PLA                         ; TheHandle - address
            STA     HNDL+2              ; Put it in HNDL+2
            LDA     0                   ; Get DP value
            PHA
            LDA     2
            PHA
            LDA     HNDL
            STA     0                   ; Direct page
            LDA     HNDL+2              ; pointer setup
```

```
            STA     2
            LDA     [0]             ; Pointer to low word
            STA     SNDDP           ; Direct page sound
            LDX     #$0308          ; SoundShutdown first
            JSL     $E10000
            LDA     SNDDP           ; WAPT - addrs of work area ptr.
            PHA                     ; put it on the stack
            LDX     #$0208          ; SoundStartup
            JSL     $E10000
            LDX     #0              ;Begin making waveform
            LDA     #40
            SEP     #$30            ;Eight-bit words
            LONGA   OFF
            LONGI   OFF
SUSTAIN     STA     WAVEFORM
            INC     A               ; Wave up from $40 to $C0
            INX
            CPX     #128
            BNE     SUSTAIN
RELEASE     STA     WAVEFORM
            DEC     A
            INX
            CPX     #256
            BNE     RELEASE
            REP     #$30
            LONGA   ON
            LONGI   ON
            PEA     WAVEFORM        ; Waveform bank
            PEA     WAVEFORM        ; Waveform address
            PEA     $0000           ; DOCstart - start addr of DOC buff
            PEA     $0100           ; Write 100 Bytes
            LDX     #$0908          ; WriteRamBlock
            JSL     $E10000
            BCC     STRTNOTE
            BRK
STRTNOTE    PEA     150             ; LFO update rate
            PEA     $0000           ; no interrupt routine
            PEA     $0000           ; (4 BYTES)
            LDX     #$0219          ; Start up note synthesizer
            JSL     $E10000
            PEA     MSG1            ; Bank of first msg
            PEA     MSG1            ; Address of first msg
            LDX     #$200C          ; TextTool command
            JSL     $E10000
            BCC     BEGIN
            BRK
BEGIN       STZ     BYTENUM         ; Beginning of track
PLAY        LDX     BYTENUM
```



```
            LDA     TRK1,X
            AND     #$007F          ; Check that it's less than $80
            STA     TONE
            INX
            INX                     ; Increment two bytes - 1 word
            LDA     TRK1,X
            STA     DURTN
            STX     BYTENUM         ; Bytenum update
GENER       PEA     $0000           ; Room on stack for result
            PEA     $0040           ; Mid priority
            LDX     #$0919          ; Sound generator
            JSL     $E10000
            BCC     G2
            BRK
G2          PLA                     ; Pull generator number of stack
            STA     GENNUM          ; put it somewhere
NOTEON      LDA     GENNUM          ; Generator number
            PHA
            LDA     TONE            ; C= $60 for tone value
            PHA
            PEA     $007F           ; Volume (<$80)
            PEA     INSTRUM         ; Instrument loc. def.
            PEA     INSTRUM
            LDX     #$0B19          ; NoteOn
            JSL     $E10000
TAP         DEC     DURTN           ; Tap your feet to keep
            BNE     TAP             ; count. (Beat-duration)
            LDA     GENNUM
            PHA
            LDA     TONE
            PHA
            LDX     #$0C19          ; Turn note off
            JSL     $E10000
            BCC     NEXT
            BRK
NEXT        LDA     BYTENUM         ; Note list position
            CMP     #DONE-TRK1      ; Length of note list
            BCC     PLAY
            LDX     #$0319          ; Shutdown note synthesizer
            JSL     $E10000
            BCC     S2
            BRK
S2          LDX     #$0308          ; SoundShutdown
            JSL     $E10000
            BCC     QUIT
            BRK
QUIT        JSL     P16             ; Stop this nonsense
            DC      I2'$29'         ; Quit code
```

```
                DC      I4'PARMBL'              ; Param table address
                BCS     ERROR
                BRK
PARMBL          DC      I4'$0000'               ; Pathname pointer
FLAG            DC      I2'$00'                 ; absolute quit
ERROR           BRK                             ; Error break
MSG1            DC      C'This is a simple scale..'
                DC      I1'13,10,13,10'
                DC      C'not bad for an Apple'
                DC      I1'0'
ID              DC      I2'0'                   ; ID space
TOOLTABL        DC      I2'1'                   ; 1 TOOL
                DC      I2'25,0
HNDL            DC      I4'0'
SNDDP           DC      I4'0'                   ; 4 bytes for storage
WAVEFORM        DS      256                     ; $100 storage
INSTRUM         DC      I1'$7F,0,$7F'           ; Ramp: $7F00 to $7F
                DC      I1'$00,$60,&0'          ; Decay at rate of $0060
                DC      I1'0,0,0'               ; to $0000. Use this stage
                DC      I1'0,0,0'               ; for release as well
                DC      I1'0,0,0'               ; STAGE 5
                DC      I1'0,0,0'               ; STAGE 6
                DC      I1'0,0,0'               ; STAGE 7
                DC      I1'0,0,0'               ; 8 STAGES
                DC      I1'1'                   ; Release segment - 1
                DC      I1'32'                  ; increment priotiry
                DC      I1'2'                   ; pitch bend range
                DC      I1'75'                  ; vibrado rate
                DC      I1'85'                  ; and speed
                DC      I1'0'                   ;
                DC      I1'1'                   ; No. of wavepoints of osc a
                DC      I1'1'                   ; No. of wavepoints for osc b
ALIST           DC      I1'127,0,0,0,0,0'       ;topkey,addr,size,ctrl,pitch
BLIST           DC      I1'127,0,0,016,0'
TONE            DC      I2'0'                   ; note value storage
GENNUM          DC      I2'0'                   ; generator number storage
DURTN           DC      I2'$0000'               ; current duration counter
BYTENUM         DC      I2'$0000'               ; current note number
TRK1            DC      I2'$54,$FFFF'
                DC      I2'$55,$FFFF'
                DC      I2'$56,$FFFF'
                DC      I2'$57,$FFFF'
                DC      I2'$58,$FFFF'
                DC      I2'$59,$FFFF'
                DC      I2'$5A,$FFFF'
                DC      I2'$5B,$C000'
                DC      I2'$5C,$C000'
```



```
                DC      I2'$5D,$C000'
                DC      I2'$5E,$C000'
                DC      I2'$5F,$C000'
                DC      I2'$60,$C000'
                DC      I2'$61,$C000'
                DC      I2'$60,$8000'
                DC      I2'$5F,$8000'
                DC      I2'$5E,$8000'
                DC      I2'$5D,$8000'
                DC      I2'$5C,$8000'
                DC      I2'$5B,$8000'
                DC      I2'$5A,$8000'
                DC      I2'$59,$4000'
                DC      I2'$58,$4000'
                DC      I2'$57,$4000'
                DC      I2'$56,$4000'
                DC      I2'$55,$4000'
                DC      I2'$54,$4000'
                DC      I2'$53,$4000'
DONE            ANOP                            ; Note list finished
END
```

Summary

The Apple IIGS is an incredibly flexible and powerful computer. On a relatively simple level, it's possible to generate a whole plethora of sounds and even musical notes. On a more complex level, it can play music and talk to you. Like everything else you've examined in this book, the key to working with sound is to start with something simple, then work your way into the more complex designs.

Unfortunately, there's not a simple programming method (at this time) that takes advantage of the DOC features, but with practice and patience—and a lot of RAM—you can develop spectacular programs with the machine.

Chapter 9

PostScript Graphics

PostScript is a programming language, just like BASIC or machine language. However, *PostScript* is used only for calculating and arranging what will be sent to a printer; thus it's called a *page description language.*

PostScript has become the de facto standard page-description language for laser printers. Both the Apple LaserWriter and the LaserWriter Plus printers have *PostScript* built into them, as do many other brands of laser printers.

The best thing about *PostScript* is that it's computer independent. As long as you can get the output into an ASCII text file (TXT filetype), the *PostScript* interpreter inside the printer can translate the text and print the desired results. Using a modem or null modem cable and a communications program, you can send the file from the Apple IIGS to the printer through the RS-232 port on the laser printer. Alternatively, the text file can be sent to a Macintosh and printed on the laser printer from the Mac.

The important question, of course, is why bother with *PostScript* in the first place. Primarily, it has to do with the printing resolution possible with *PostScript* output. On an 8½-inch screen, the Apple IIGS in the 640-pixel mode produces about 75 dots per inch (dpi). Output to a standard printer is about the same. With *PostScript,* the output resolution is determined by the printer, not the screen. The Apple LaserWriter and LaserWriter Plus printers can print at a resolution of 300 dpi output, or four times what you can get on your screen. Some *PostScript*-based typesetting machines achieve resolutions of up to 2400 dpi. The great thing is that the same program which produces 300 dpi on a LaserWriter will produce 2400 dpi on an advanced machine.

If you've used a programming language called FORTH, *PostScript* will seem familiar. Both languages use the stack and develop *words* that contain instructions. Both also use what's called *reverse*

Polish or *postfix* notation. (Reverse Polish refers to the Polish mathematician who developed the particular stack arrangement for calculations; the name *PostScript* was derived from *postfix*.)

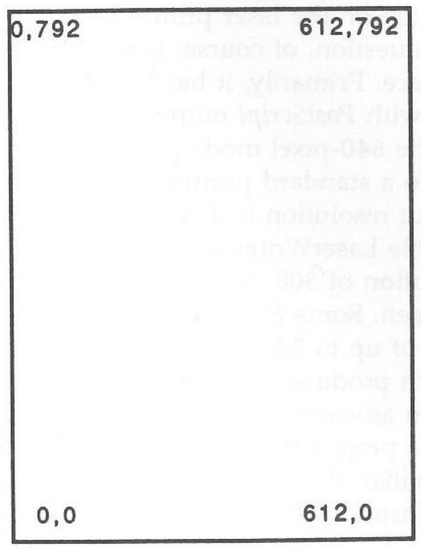
The Work Area

In the world of typesetting, the basic unit of measurement is the *point*. There are 72 points to the inch. (Notice how close that is to the number of pixels per inch on an 8½-inch screen in the 640 super high-resolution graphics mode on the Apple IIGS.)

In *PostScript* programs, the 72 dpi measure is used as well. On a standard 8½ × 11 inch sheet of paper, you're dealing with a 612 (8.5 × 72) by 792 (11 × 72) matrix. That's about half a million points per page. Since *PostScript* is a very smart language, it can translate those 72 dpi into 300 dpi on the LaserWriter, or into an even higher dpi resolution on a larger printer.

Unlike most matrices, such as your computer screen which begins the X,Y 0,0 position in the upper left corner of a page, *PostScript* begins in the *lower* left. The following figure shows the relationship of point positions on an 8½ × 11 inch page.

Figure 9-1. *PostScript* Point Positions



Positive values indicate a move upwards and to the right. To move down and to the left, negative numbers are used. Thus, to move one inch (72 points) to the right and two inches (144 points) down would be expressed as

```
72  -144
```

in a *PostScript* program.

PostScript Conventions

Before you actually start programming in *PostScript*, there are some key conventions you need to know.

Lowercase statements: All commands and statements in *PostScript* are in lowercase.

Comments: Use the percent sign (%) to indicate a comment line. Comments must begin with %. Everything is ignored by the *PostScript* interpreter from that point to the end of the line.

Word definitions: Word definitions begin with a slash (/) and end with the statement *def*. All fonts also begin with a slash.

To run a *PostScript* program, the statement *showpage* is placed at the bottom of the program. When the interpreter reaches that point, the laser printer prints everything which precedes the showpage statement.

The easiest way to write a *PostScript* program is to use a text or word processor that can save a file in text (ASCII) format. If your word processor can't save a file as a text file or you don't have a word processor program, use the following text editor program to write your *PostScript* programs.

Program 9-1.

```
 10 DIM A$(300): D$ = CHR$(4)
 20 INPUT "Name of file ";NF$
 30 PRINT "<N>ew OR <A>ppend ";
 40 GET AN$
 50 IF AN$ = "N" OR AN$ = "n" THEN OP$ = "OPEN"
 60 IF AN$ = "A" OR AN$ = "a" THEN OP$ = "APPEND"
 70 INPUT "PS=> ";A$(X)
 80 IF A$(X) = "Q" OR A$(X) = "q" THEN 200
 90 X = X + 1
100 GOTO 70
```

```
200 PRINT D$;OP$;NF$
210 PRINT D$;"WRITE"NF$
220 FOR N = 0 TO X - 1
230 PRINT A$(N)
240 NEXT
250 PRINT D$;"CLOSE"
```

This crude editor doesn't give you much editing ability, but as a last resort, it works.

Moving and Drawing

To get started, the *operator*, **newpath**, declares the current path to be empty. The *path* refers to the course the imaginary pen takes as you draw, place text, or move. After each activity, this imaginary pen is at the end of the current path, and newpath simply indicates it's starting a new path. Thus, the current point is not defined.

Having declared newpath, you can move to a *current point* with the **moveto** operator. Remember that moving up and right is a positive value and moving left and down is a negative value, since you begin in the *lower* left corner of a page.

The moveto operator requires two values: the horizontal *(x)* and vertical *(y)* coordinates preceding the operator. Thus, the line

```
100 150 moveto
```

places the current point 150 points from the bottom and 100 points to the right. Figure 9-2 shows approximately where the current point will be.

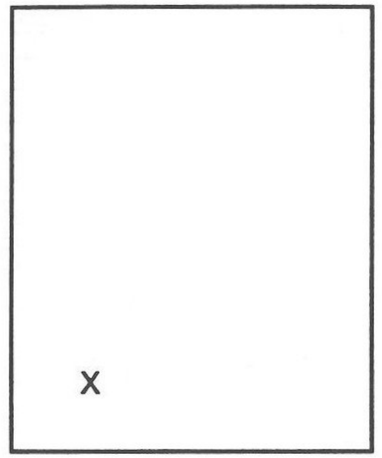
For many applications, you'll want some points away from the lower left corner as a point of reference. For example, your page margin may be one inch all the way around. Therefore, instead of having the 0 0 point in the lower left corner, you want it 72 points (one inch) to the right and 72 points above the bottom.

The **translate** operator redefines, or *translates*, the new 0 0 point. Thus,

```
72 72 translate
```

would make the 0 0 point one inch from the bottom and one inch to the right. When more advanced programming is required, this operator will be very handy.

Figure 9-2. Up and to the Right



Lines

To draw something, you'll need a statement to tell the printer to create lines, arcs, and other shapes, and then to draw them on the screen. Let's start with lines.

There are two basic types of lines, *absolute* and *relative*. The **lineto** operator draws an absolute line from the current point to the point specified. For example, if the current point were 100 100, and the lineto statement were

```
150 150 lineto
```

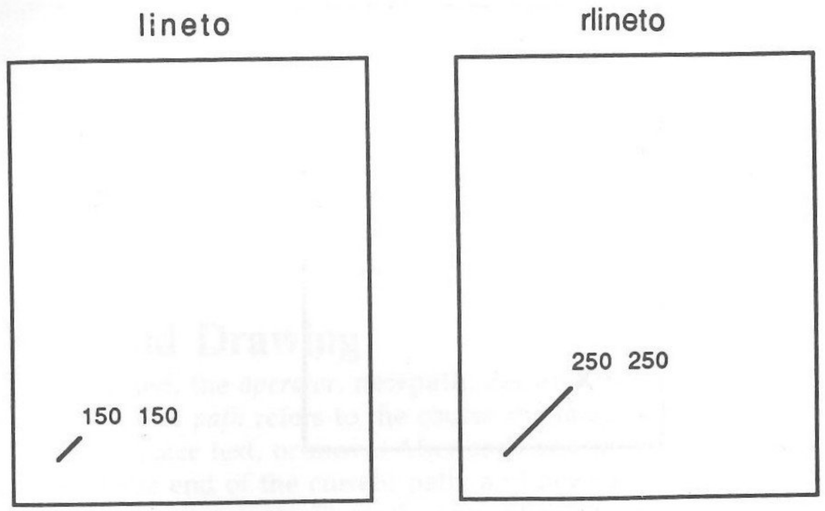
there would be a line drawn from 100 100 to 150 150.

In contrast, **rlineto** creates a relative line of a specified length. The start of the line is the current point. Beginning at the same 100 100 point, a rlineto statement of

```
150 150 rlineto
```

creates a line from 100 100 to a point 150 points up and 150 points to the right. The result is a line from 100 100 to 250 250.

Figure 9-3. lineto and rlineto



Actually, the lineto and rlineto operators don't really draw a line, but instead define the path of the line. The **stroke** operator tells the printer to draw. Several lines at once can be *stroked* with a single stroke operator.

When you want an actual drawing to appear on paper, use the operator **showpage.** As already mentioned, showpage is usually the final operator in a *PostScript* program.

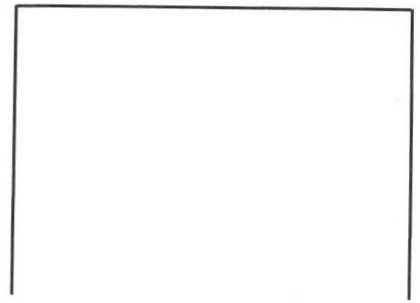
At this point, you have enough information to write a *PostScript* program. This first short *PostScript* program draws three lines, making three sides of a box using both lineto and rlineto.

Program 9-2.

```
                %Three lines
newpath
72 72 translate %Now the 0 0 point is 1 inch up and to the right
0 0 moveto      %Starting point is actually 72 72
0 72 lineto     %One inch straight up
72 0 rlineto    %One inch to the right
0 -72 rlineto   %One inch straight down
stroke          %Draw the lines
showpage        %Out to the printer
```

If you wrote the program correctly, the output should look like Figure 9-4.

Figure 9-4. Three-Sided Box



At first glance, you might wonder why the lineto line and the rlineto lines are the same length. The reason is that the first lineto operator sent a line to vertical point 72 from the 0 0 position (which previously had been translated to be 72 72). The absolute vertical point 72 is the same as the relative point 72, since the first current point was 0 0. If the current point had been anything else, the first line would have been a different length than the second two.

To close the box, all that's required is the **closepath** operator. This is a handy operator—once two nonstraight connecting lines have been drawn, a single closepath operator will make a polygon.

Add the operator closepath right before *stroke* in the above program and the fourth side is drawn.

Program 9-3.

```
                %Box
newpath
72 72 translate %Now the 0 0 point is 1 inch up and to the right
0 0 moveto      %Starting point is actually 72 72
0 72 lineto     %One inch straight up
72 0 rlineto    %One inch to the right
0 -72 rlineto   %One inch straight down
closepath       %Completes the box
stroke          %Draw the lines
showpage        %Out to the printer
```

Before moving to curves, let's look at some other polygons using more or less random lines.

Program 9-4.

```
                %Strange shape
72 72 translate
0 0 moveto
30 90 lineto
70 15 rlineto
-20 -40 rlineto
closepath       %Who knows what it looks like?
stroke
showpage
```

Experiment with lines and shapes to see what you can draw.

Curves

Curves require a different approach. Instead of two parameters, there are five.

X Y Radius Begin End

The **arc** operator draws arcs counterclockwise, while the **arcn** operator draws them clockwise. The X and Y positions plot the arc's center. It's best to think of the arc as a side of a circle and the X Y position as the circle's center.

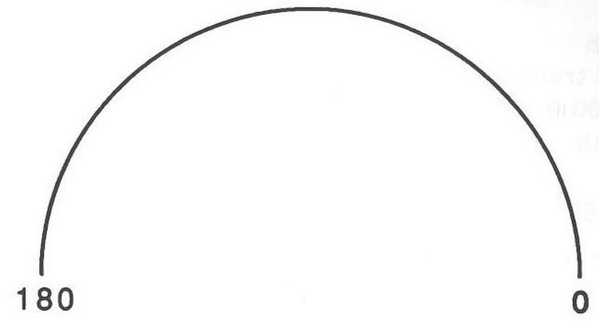
If the moveto operator is used before drawing the arc, there will be a straight line from the current position to the beginning point of the arc. It's best to plot the relative position with the translate operator, and then begin your arc. Let's take a look at an example.

Program 9-5.

```
                %Arc one
newpath
200 200 translate    %Give it some room
50 50 100 0 180 arc  %From 0 to 180. Also try arcn
stroke
showpage
```

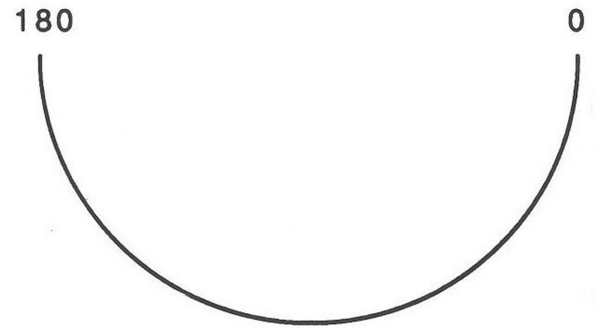
This program makes an arc like the one shown in Figure 9-5.

Figure 9-5. Half-Circle



Notice that the zero (0) point is at the 3 o'clock position and 180 is at the 9 o'clock positon. Change the operator from arc to arcn, and the curve will look like the one in Figure 9-6.

Figure 9-6. Half-Circle Reversed



Since there are 360 degrees in a circle, a beginning point of 0 and an ending point of 360 makes a complete circle. The following line does that.

```
50 50 50 0 360 arc
```

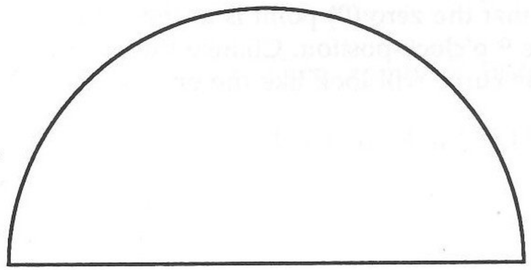
What would happen if the closepath operator were used? Would it close the circle or draw a line from the beginning of the path to the current point? The best way to find out is to try it.

Program 9-6.

```
                        %Moon over Miami
newpath
200 200 translate
50 50 100 0 180 arc
closepath
stroke
showpage
```

The results should show an arc with a straight line from the beginning to the ending point of the arch.

Figure 9-7. Moon Over Miami



Fills and Line Width

An enclosed area can be filled with various shades of gray. A full black is 0 and a full white is 1. Using the **setgray** operator and a fraction, it's possible to establish a particular shade of gray. By using **fill**, the enclosed area can be shaded with that gray scale.

To illustrate, let's draw a fan shape made up of an arc and two straight lines. Neither the lineto nor the rlineto operator will be used. This will also illustrate what happens if moveto is used before an arc is drawn.

Program 9-7.

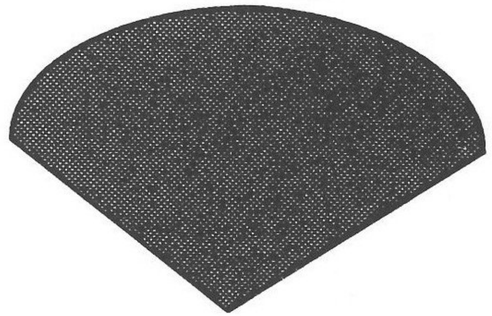
```
                        %Fan
newpath
200 200 translate
0 0 moveto              %This will cause a line to begin-point of arc
0 0 50 36 144 arc       %Curve over the top
closepath               %This will draw second straight line
```



```
.25 setgray             %25% white/75% black
fill                    %Fill enclosed area
stroke
showpage
```

The program draws a fan-shaped object something like the one in Figure 9-8.

Figure 9-8. Shaded Fan

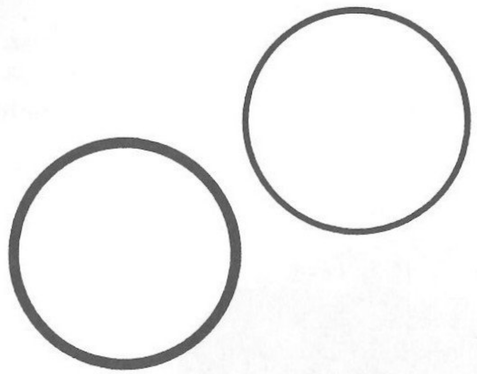


PostScript operators can make different-sized line thicknesses. The operator **setlinewidth** does exactly what it says. The default is 1 point, but by specifying size, you can make any line width desired. For instance, this next program draws circles with different line widths.

Program 9-8.

```
                        %Fat and skinny circles
100 100 translate
50 50 50 0 360 arc
5 setlinewidth          %Before the stroke set the line width
stroke
150 150 50 0 360 arc
2 setlinewidth          %Before the stroke set the line width
stroke
showpage
```

Figure 9-9. Thick and Thin Circles



Text

Now that you have a way of expressing lines and curves, the next step is to print text on the screen. The fonts must be in the laser printer before they can be effectively used in a *PostScript* program. Also, the fonts must be called exactly as they exist in your printer's ROM or RAM. For example, the LaserWriter Plus has 35 fonts called by the following names.

AvantGarde-Book
AvantGarde-BookOblique
AvantGarde-Demi
AvantGarde-DemiOblique
Bookman-Demi
Bookman-DemiItalic
Bookman-Light
Bookman-LightItalic
Courier
Courier-Bold
Courier-BoldOblique
Courier-Oblique
Helvetica
Helvetica-Bold
Helvetica-BoldOblique
Helvetica-Narrow
Helvetica-Narrow-Bold
Helvetica-Narrow-BoldOblique
Helvetica-Narrow-Oblique
Helvetica-Oblique
NewCenturySchlbk-Bold
NewCenturySchlbk-BoldItalic
NewCenturySchlbk-Italic
NewCenturySchlbk-Roman
Palatino-Bold
Palatino-BoldItalic
Palatino-Italic
Palatino-Roman
Symbol
Times-Bold
Times-BoldItalic
Times-Italic
Times-Roman
ZapfChancery-MediumItalic
ZapfDingbats

Remember, *PostScript* doesn't require that these fonts be in your *computer*. As long as they're in the *printer*, you can use them in writing *PostScript* programs on your Apple IIGS.

First set up a font. To do that, use this formula

/Font-Name **findfont N scalefont setfont**

where **N** is the size of the font in points.

For example, suppose you want to use boldface Helvetica in a 14-point size. You need to locate Helvetical-Bold with findfont, then set the size with a value preceding **scalefont**. Finally, **setfont** sets the type face and font size. The entire statement would be

```
/Helvetica-Bold findfont 14 scalefont setfont
```

Once the font is set, use moveto to place the first letter of the text, then place in parentheses ( ) the string of characters you want to print. The operator **show** places the text, and showpage, as always, prints it.

Program 9-9.

```
                                        %Simple text printing

/Helvetica-Bold findfont 14 scalefont setfont
72 720 moveto                           %Top of page with 1 inch
                                        margin
(Printing text is as easy as 1,2,3.) show
showpage                                %Message out
```

And your message looks like this:

Printing text is as easy as 1,2,3.

Angled Text

Besides printing in a straight line, you can angle your text using the **rotate** operator, and change its shape using **scale.**

Using the rotate operator, you can start in the middle of a page.

Program 9-10.

```
                                %Flip out
/Helvetica findfont 8 scalefont setfont
72 396 translate                %Middle of page with 1 inch left margin
0 0 moveto
30 rotate
```

```
(Flip out!    ) show   %Note 3 spaces before second parenthesis
40 rotate
(Flip out!   ) show
50 rotate
(Flip out!    ) show
showpage
```

Figure 9-10. Flip Out

Note the arrangement of the string *Flip out!* and how each string begins near the end of the preceding string. The current point is always the end of a string until it's moved.

The **scale** operator expects two values: the *x*- and *y*-scale of a string or graphic. Let's stick with text strings to see how it works. By using a larger value for *y* than for *x*, it's possible to create tall, narrow characters. For short squatty ones, a larger *x* value is in order. The following program shows both and introduces the operator **rmoveto**.

Program 9-11.

```
                %The Long and Short of It
/Times-Roman findfont 12 scalefont setfont
newpath
72 720 translate
0 0 moveto
1 1 scale
(Normal Nelly) show
10 0 rmoveto
1 4 scale
(Long tall Sally) show
showpage
```

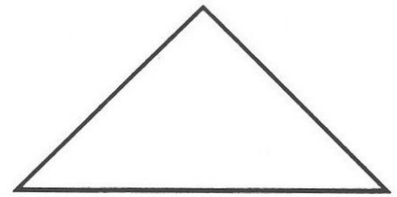
Figure 9-11. Long, Tall Sally

By combining text and graphics, it's possible to label your graphics. To label a triangle, for instance, draw the triangle, then label it. Notice that in Program 9-12 the label is set under the base of the triangle.

Program 9-12.

```
                %Labeled Triangle
newpath
72 72 translate
0 0 moveto
72 72 lineto
72 -72 rlineto
closepath       %Back to 0 0 position
stroke
/Times-Roman findfont 12 scalefont setfont
40 -14 moveto
(Triangle) show
showpage
```

Figure 9-12. Triangle and Triangle



Modifying Text

You can change text characters in other ways with *PostScript*. Two placement operators are important here—**gsave** and **grestore**.

As you create more and more complicated graphics, you need operators which can keep track of where you've been. The gsave operator saves the current point and grestore restores it. For example, if you want to perform two different operations on the same graphic, you may need to go back to the beginning point. The gsave operator keeps the *x,y* position until it encounters grestore.

The operator **charpath** refers to the path of a character or *character path*. Since all fonts are actually characters drawn in memory

which can be stroked just like lines, text is actually outlines of characters which can be drawn and filled. Using the true charpath, the path can then be stroked (drawn) and filled with whatever value of setgray you wish. To create a character with different shades, it's necessary to first create them with true charpath instead of using the show operator. The fill operator will fill the character with the chosen setgray color, and stroke will treat it as a drawn character.

Since you first have to fill the character, then stroke the path, it's necessary to return to the character's starting position. Before the fill, gsave is used—after the fill, grestore sets the positon for stroke. Program 9-13 shows how to do all this.

Program 9-13.

```
                              %Creative characters
/Helvetica-Bold findfont 72 scalefont setfont
72 396 translate
0 0 moveto
(Compute!) true charpath  %Use instead of show
gsave                     %Save this point
.7 setgray
fill
grestore                  %Restore saved point
0 setgray
stroke                    %Do it again
showpage
```

Figure 9-13. COMPUTE! Filled

Defining Words

The structure of *PostScript* is centered around developing variables and procedures. These are placed into a dictionary, then executed when the variable or procedure name is placed in the program. In other words, you can build a dictionary of words which consist of a *PostScript* program. To see how this works, let's start with a simple figure, a triangle, and define it as a procedure. Each procedure definition begins with a slash (/) and ends with *def*. Braces ({ }) enclose the defined procedure.

Program 9-14.

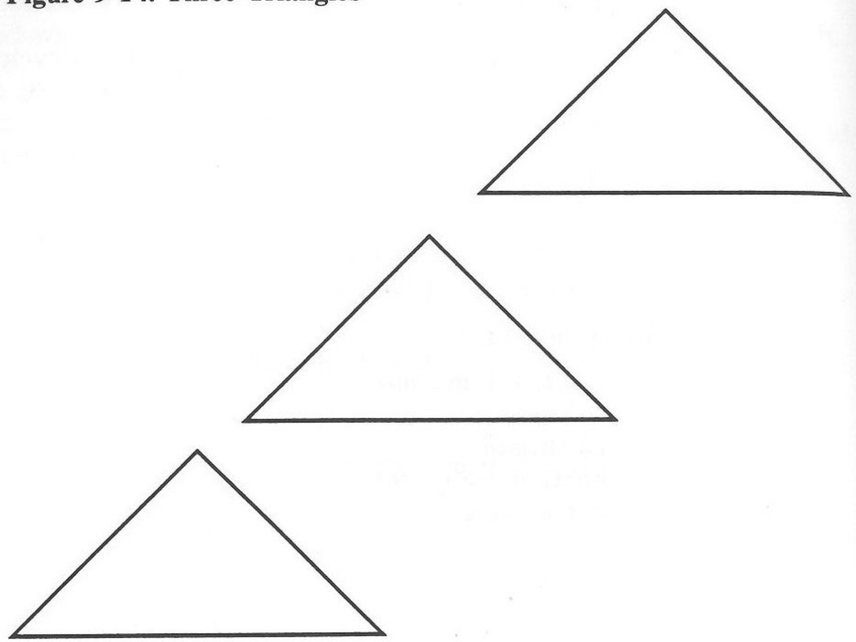
```
/Triangle {0 0 moveto
72 72 lineto
72 -72 rlineto
closepath stroke } def
72 72 translate
Triangle
showpage
```

By itself, this isn't very interesting. However, if you have a program that requires lots of triangles—or any other shape for that matter—it's far easier to write *Triangle* several times than to rewrite the entire triangle routine. To see how this works, let's change the program to draw several triangles.

Program 9-15.

```
/Triangle {0 0 moveto
72 72 lineto
72 -72 rlineto
closepath stroke } def
72 72 translate
Triangle
144 144 translate
Triangle
216 216 translate
Triangle
showpage
```

Figure 9-14. Three Triangles



Likewise, once a shape is defined in a procedure, you can rotate and scale that shape.

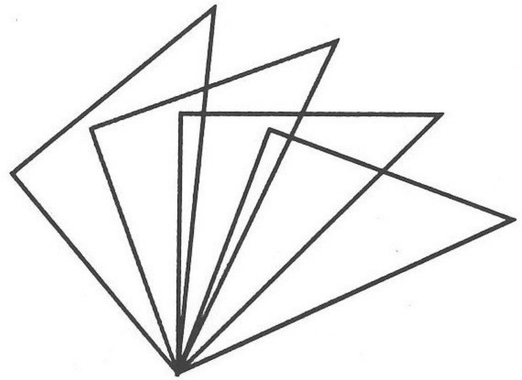
Program 9-16.

```
%Rotating Angles
/Triangle {0 0 moveto
72 72 lineto
72 -72 rlineto
closepath stroke } def
200 396 translate
gsave
Triangle
grestore
gsave
20 rotate
Triangle
grestore
gsave
```



```
40 rotate
Triangle
grestore
gsave
60 rotate
Triangle
showpage
```

Figure 9-15. Rotating Triangles



Summary

This chapter has only given you some elementary *PostScript* information. Two books, *PostScript Language: Tutorial and Cookbook* and *PostScript Language Reference*, both by Adobe Systems, Inc., have a full description of the language. If the information here has touched your interest, take a look at these two books.

Appendices

Appendix A

Error Messages

Whenever you have an error-handling routine using ONERR and PEEK (222), you'll be given a code which represents an error. For example, error 42 means that you are OUT OF DATA when your program contains a READ statement and a number of DATA statements.

The first set of error messages are from DOS 3.3 and Applesoft. The second set consists of ProDOS 8 error messages, and the third set of messages, while you may not encounter them with Applesoft programs, are ProDOS 16 error messages.

Applesoft and DOS 3.3 Error Messages

Error	Message
0	NEXT WITHOUT FOR
1	LANGUAGE NOT AVAILABLE
2,3	RANGE ERROR
4	WRITE-PROTECTED
5	END OF DATA
6	FILE NOT FOUND
7	VOLUME MISMATCH
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
11	DOS SYNTAX ERROR
12	NO BUFFERS AVAILABLE
13	FILE MISMATCH
14	PROGRAM TOO LARGE
15	NOT DIRECT COMMAND
16	PROGRAM SYNTAX ERROR
22	RETURN WITHOUT GOSUB
42	OUT OF DATA
53	ILLEGAL QUANTITY

69	OVERFLOW
77	OUT OF MEMORY
90	UNDEFINED STATEMENT
107	BAD SUBSCRIPT
120	REDIMED AN ARRAY
133	DIVISION BY ZERO
163	TYPE MISMATCH
176	STRING TOO LONG
191	FORMULA TOO COMPLEX
224	UNDEFINED FUNCTION
254	BAD INPUT RESPONSE
255	CONTROL-C INTERRUPT

ProDOS 8 Error Messages

Error	Message
2	RANGE ERROR
3	NO DEVICE CONNECTED
4	WRITE-PROTECTED
5	END OF DATA
6	PATH NOT FOUND
7	PATH NOT FOUND
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
11	INVALID OPTION
12	NO BUFFERS AVAILABLE
13	FILE TYPE MISMATCH
14	PROGRAM TOO LARGE
15	NOT DIRECT COMMAND
16	SYNTAX ERROR
17	DIRECTORY FULL
18	FILE NOT OPEN
19	DUPLICATE FILENAME
20	FILE BUSY
21	FILE(S) STILL OPEN

ProDOS 16 Error Messages

$1	Invalid call number
$5	Call pointer out of bounds
$6	Invalid caller identification
$10	Device not found
$11	Invalid device ref number
$20	Invalid request code
$25	Interrupt table full
$26	Invalid operation
$27	I/O Error (Note: Different code number from DOS 3.3 above.)
$28	No device connected
$2B	Write-protected
$2E	Disk switched
$30–$3F	Device-specific errors
$40	Invalid pathname syntax
$42	FCB full
$43	Invalid file reference number
$44	Path not found
$45	Volume not found
$47	Duplicate file name
$48	Volume full
$49	Directory full
$4A	Version error
$4B	Unsupported storage type
$4C	End of file encountered (EOF)
$4D	Position out of range
$4E	Access not allowed
$50	File is open
$51	Directory structure damaged
$52	Unsupported volume type
$53	Parameter out of range
$54	Out of memory
$55	VCB full
$57	Duplicate volume
$58	Not a block device
$59	Invalid level
$5A	Block number out of range

Fatal Errors
$1 Unclaimed interrupt
$A VCB unusable
$B FCB unusable
$C Block zero allocated illegally

Appendix B

Selected Apple IIGS Toolbox Routines

QuickDraw II Calls

Chapter 7 included a detailed example and explanation of how to get into the QuickDraw II Toolbox to create graphics. An entire book would be required to detail every QuickDraw II Toolbox call, but several are useful as starting points for super high-resolution graphics on the Apple IIGS. This appendix lists by name all of the calls, though some have been explained in greater detail.

The fundamental procedures for using a Toolbox call are:

- Place the necessary paramters on the stack, usually using PEA
- Using LDX, load the call number in the immediate mode into the X register
- Using JSL, jump to the Toolbox address at $E10000

All of this is to be done in 16-bit (65816) mode.

The first set of functions in the QuickDraw II Toolbox are for housekeeping purposes. They set up the various registers and pointers to allow access to the graphics tools. They include **QDBoot-Init** to initialize the QuickDraw II tools when the system is booted, **QDStartup** to initialize QuickDraw II and set the standard port and clear the screen, and **QDShutDown**, which turns QuickDraw II off and frees the buffers. **QDVersion** and **QDStatus** provide information on the version of QuickDraw II and whether or not it's active, respectively.

A second set of calls are considered *global* in that they involve scanning, which sets general properties of graphics. A scan line has a Scanline Control Byte (SCB) that controls the line's characteristics. The call **GetStandardSCB** returns information about the SCB.

The first four bits (0–3) are used for Color Table 0, bit 4 is reserved, bit 5 is Fill (0 = off, 1 = on), bit 6 interrupt (0 = off, 1 = on), and bit 7 is the Color Mode (0 = 320, 1 = 640). The call **SetMasterSCB** is used for setting the low byte of the master SCB, and **GetMaster-SCB** returns the information in the low byte of the master SCB. **SetSCB, GetSCB,** and **SetAllSCBs** are further scan-line control byte calls. For setting the color table, either in the 320 or 640 mode, the **InitColorTable** call is used.

Calls to **SetColorTable, GetColorTable, SetColorEntry,** and **GetColorEntry** all access the routines to set and get information about the colors.

The other global calls deal with the fonts, clearing the screen, and turning the super high-resolution graphics mode on and off. The calls include **SetSysFont, GetSysFont, ClearScreen, GrafOn,** and **GrafOff.** As you can see, the calls are fairly self-explanatory as to their function, making it much easier to use the graphics than more obtuse codes.

QuickDraw Calls

GrafPort Calls $—04

OpenPort	InitPort	ClosePort	SetPort	GetPort
SetPortLoc	GetPortLoc	SetPortRect	GetPortRect	SetPortSize
MovePortTo	SetOrigin	SetClip	GetClip	ClipRect
HidePen	ShowPen	GetPen	SetPenState	GetPenState
SetPenSize	GetPenSize	SetPenMode	GetPenMode	SetPenPat
GetPenPat	SetSolidPenPat	SetPenMask	GetPenMask	SetBackPat
GetBackPat	SetSolidBackPat	SolidPattern	PenNormal	MoveTo
Move	SetFont	GetFont	SetFontID	GetFontID
GetFontInfo	GetFGSize	GetFontGlobals	SetFontFlags	GetFontFlags
SetTextFace	GetTextFace	SetTextMode	GetTextMode	SetSpaceExtra
GetSpaceExtra	SetCharExtra	GetSpaceExtra	GetForeColor	GettForeColor
SetBackColor	GetBackColor	SetBufDims	ForceBufDims	SaveBufDims
RestoreBufDims	SetClipHandle	GetClipHandle	SetVisRgn	GetVisRgn
SetVisHandle	GetVisHandle	SetPicSave	GetPicSave	SetRgnSave
GetRgnSave	SetPolySave	GetPolySave	SetGrafProcs	GetGrafProcs
SetUserField	GetUserField	SetSysField	GetSysField	

Some GrafPort Parameters

In high-, low-, double-high- and double-low-resolution graphics, there's only a single line size, but in super high-resolution graphics, in either the 320 or 640 mode, the size of the pen that draws the line can be controlled by the user.

Using **SetPenSize,** it's possible to vary the size of the pen in

both horizontal and vertical pixel widths. Both the vertical and horizontal parameters must be at least $0001.

SetPenSize Parameters
Toolbox number = $2C04
Pen Width Number of pixels wide
Pen Height Number of pixels deep
Notes: Lines can be very wide to work almost like a background or block.

Setting Colors

The two key calls for setting the color are the **SetSolidPenPat** color routine and **ClearScreen.** ClearScreen sets the screen memory to the specified background color and the other sets the color of the pen which will draw on the background.

ClearScreen Parameters
Toolbox number = $1504
Color code (See below)
Notes: All four hexadecimal values for the color code must be the same. For example, $1111 would give a solid dark gray background. However, $0001 would give a pattern broken by vertical lines.

SetSolidPenPat Parameters
Toolbox number = $3704
Color code (See below)
Notes: Only use a single value such as $E, $2, and so on.

320 Mode

Pixel	Color	Code
$0	Black	0 0 0
$1	Dark Gray	7 7 7
$2	Brown	8 4 1
$3	Purple	7 2 C
$4	Blue	0 0 F
$5	Dark Green	0 8 0
$6	Orange	F 7 0
$7	Red	D 0 0
$8	Flesh	F A 9
$9	Yellow	F F 0
$A	Green	0 E 0
$B	Light Blue	4 D F

$C	Lilac	D A F
$D	Periwinkle Blue	7 8 F
$E	Light Gray	C C C

640 Mode

Pixel	Color	Code
$0	Black	0 0 0
$1	Red	F 0 0
$2	Green	0 F 0
$3	White	F F F
$4	Black	0 0 0
$5	Blue	0 0 F
$6	Yellow	F F 0
$7	White	F F F
$8	Black	0 0 0
$9	Red	F 0 0
$A	Green	0 F 0
$B	White	F F F
$C	Black	0 0 0
$D	Blue	0 0 F
$E	Yellow	F F 0
$F	White	F F F

Drawing Calls

This set of Toolbox calls are what most programmers use most often when creating graphics directly or when writing a drawing program.

Lines

LineTo Line

These two routines operate very much alike.

LineTo Parameters
Toolbox number = $3C04
X position ($0–$0280 or $0–$0154)
Y position ($0–$C8)

Line Parameters
Toolbox number = $3D04
X position ($0–$0280 or $0–$0154)
Y position ($0–$C8)

Rectangles

FrameRect PaintRect EraseRect InvertRect FillRect

Regions

FrameRgn PaintRgn EraseRgn InvertRgn FillRgn

Polygons

FramePoly PaintPoly ErasePoly InvertPoly FillPoly

Ovals

FrameOval PaintOval EraseOval InvertOval FillOval

Rounded-Corner Rectangles

FrameRRect PaintRRect EraseRRect InvertRRect FillRRect

Arcs

FrameArc PaintArc EraseArc InvertArc FillArc

Pixel Transfers

ScrollRect PaintPixels PPToPort

Text Drawing and Measuring

DrawChar DrawText DrawString DrawCString CharWidth
TextWidth StringWidth CStringWidth CharBounds TextBounds
StringBounds CStringBounds

Mapping and Scaling Utilities

MapPt MapRect MapRgn MapPoly ScalePt

Miscellaneous Utilities

Rectangle Calculations
SetRect OffsetRect InsetRect SectRect UnionRect
PtInRect Pt2Rect EquallRect EmptyRect

Point Calculations
AddPt SubPt SetPt EqualPt LocalToGlobal
GlobalToLocal

Region Calculations
NewRgn DisposeRgn CopyRgn SetEmptyRgn SetRectRgn
RectRgn OpenRgn CloseRgn OffsetRgn InsetRgn
SectRgn UnionRgn DiffRgn XorRgn PtInRgn
RectInRgn EqualRgn EmptyRgn

Polygon Calculations
OpenPoly ClosePoly KillPoly OffsetPoly

Other
Random SetRandSeed GetPixel

The Sound Manager

The program in Chapter 8 that showed how to use the sound tools was a simple example of what can be accomplished with the Toolbox routines. These routines take advantage of the 5503 Ensoniq Digital Oscillator Chip (DOC). The DOC has 32 multiplexed digital oscillators to give you everything from beeps and buzzes to a talking computer and symphonic orchestra. However, like the super high-resolution graphics, you need to use the DOC Toolbox to really take advantage of this feature.

To get started, let's take a quick look at the registers used to control the sounds in DOC.

Frequency Control (Low and High)

Each of the 32 digital oscillators is composed of two eight bit registers—joined together they form a 16-bit value used for the 24-bit linear accumulator. The value of this register pair is added to the

current value stored in the 24-bit accumulator.

Address: $00–$1F (low)
$20–$3F (high)

Volume

This register set controls the volume level of the sound created.

Address: $40–$5F

Waveform Data Sample

This reads the last value from the waveform table.

Address: $60–$7F (Address Pointer)

These registers are used to determine where in RAM the waveform tables are located. Each waveform table begins with the first address of a page and must continue upward through RAM. It cannot exceed 64K.

The next register keeps track of where the table ends.

Address: $80–9F

Control Register

Channel assignment, oscillator mode, and halt bit are all controlled by this register. Bits 4–7 make up the channel assignment. Those four bits can assign up to 16 channels for sound. Bit 3 is the interrupt enable used for ordering output when more than a single oscillator has generated output. It helps keep all of the different sounds organized. Bits 1 and 2 set the oscillating mode for each oscillator. Bit 0 is the halt bit, indicating when an oscillator has been stopped by the microprocessor or DOC.

Address: $A0–$BF

Bank Select/Resolution/Waveform Registers

Each register uses seven bits for controlling three major functions. (Bit 7 is not used.)

Bit 6 determines whether the DOC address range is 0–64K

(Bank 0) or 65K–128K (Bank 1). Bits 3–5 specify the size of the waveform table, ranging from 256 bytes to 32K bytes. Finally bits 0–2, called the *resolution determination bits*, actually determine the final address for the waveform table.

Address: $C0–$DF

Oscillator Interrupt, Oscillator Enable, and A/D Converter Registers

These three registers (not bits) control the oscillators and analog-to-digital conversion.

Addresses: $E0–$E2

Sound Tools $—08

It's not simple to crank up the kinds of sound seen in musical demonstrations on the IIGS. The sound tools are provided to assist the programmer.

There are 18 sound function calls and and 6 low-level routines for accessing the power of DOC. It works through a Sound Toolset with a specified number. The ToolLocator finds this number to use the sound tools. The following calls are available.

Function Calls

SoundBootInit	SoundStartup
SoundShutdown	SoundVersion
SoundReset	SoundToolStatus
WriteRamBlock	ReadRamBlock
GetTableAddress	GetSoundVolume
SetSoundVolume	FFStartSound
FFStopSound	FFSoundStatus
FFGeneratorStatus	SetSoundMIRQV
SetUserSoundIRQV	FFSoundDoneStatus

To set the volume, for example, you'd use the **SetSoundVolume** tool. It has two parameters: one for setting the volume and one for the generator to be set.

SetSoundVolume Parameters

Toolbox number = $0D08
Volume setting $0–FF
Generator number $0E–$FF

Notes: If the generator number is from $00–$0E, the pair of generators is changed. But if the value is $0F or greater, the volume of the system is affected.

Low Level Routines

Read Register	Write Register
Read Ram	Write Ram
Read Next	Write Next

Appendix C

Using the *APW Assembler*

The *Apple IIGS Programmer's Workshop Assembler*, better known as the *APW Assembler*, can be used to create assembly language programs for graphics and sound.

Specifically, this assembler is used to create the sound programs which call the built-in Toolbox routines in Chapter 8. It's also useful for accessing the super high-resolution graphics or any other program requiring assembly language programming.

You don't have to use the *APW Assembler*, for there are other Apple IIGS assemblers on the market. The *Merlin 816 Assembler* by Roger Wagner Publishing Company is a very popular macro assembler, for example. The *ORCA/M Assembler* is very similar to the *APW Assembler*.

System Requirements

To use *APW*, you'll need an Apple IIGS with at least one 800K 3½-inch disk drive for the program and one other drive. It will work with the following combinations:

- Two 800K 3½-inch drives
- One 800K 3½-inch drive and one standard 5¼-inch inch drive
- One 800K 3½-inch and one hard disk

The program disk must go on the hard drive or in the 800K 3½-inch drive. A standard 5¼-inch drive doesn't have sufficient room for the files.

It's also necessary to have considerable memory. A minimum of 512K is required (256K on the motherboard plus 256K on a

memory expansion board). More is recommended. Programs for this book, for instance, were developed on a one-megabyte system. For larger programs, especially those making several Toolbox calls, it's a good idea to have lots of memory.

Getting Started with *APW*

No matter what type of system you use, make a backup of your *APW* disk. It's very easy to accidentally erase a disk in the *APW* and assembly language environment by mistake. If you only have a single 800K 3½-inch disk drive, copying is a tedious chore, but it can be done. Here's what you need to do to copy your *APW* disk.

Step 1: Open the write-protect window on your original *APW* disk and insert it in the drive. Boot your system with the *APW* disk.

Step 2: Get a blank disk. Use the INIT command and name the device and file to prepare the blank disk. (If you're copying to a hard disk, it's unnecessary to initialize it.) The device name is accessed by using the commands SHOW UNITS.

For example, with a single disk drive, the sequence will look something like

INIT .D1/APWBU <Return>

Your screen would tell you to insert the disk. Put in the blank disk. (The disk name *APWBU* simply refers to *APW Back Up*.)

Step 3: Put the write-protected original *APW* disk in one drive and the blank disk in the second drive. If you have a single 800K drive and no hard disk, you'll have to swap disks several times. The process will take close to 15 minutes. Once the disks are ready in their drives, type

COPY /APW/= /APWBU /=

Follow the swapping prompts if you have a single 800K drive and no hard disk. Otherwise, just wait until the disk is copied.

Step 4: When you're finished, you'll want to rename your disk from APWBS to APW. To do this, enter

RENAME /APWBU /APW

Using the Editor

Get a blank disk and initialize it for the programs you'll write yourself. There's not enough room to put many on the *APW* disk. Even if you have a RAM disk, there probably won't be enough room there either.

You'll use this blank disk to save and test your assembly language programs. To make it the *key* or *current* disk, use the command

PREFIX ASSEM

The rest of this discussion assumes the name of the disk to be *ASSEM*. Any other legal filename is fine, of course. Now that the ASSEM disk is the current disk, you can begin writing assembly language programs with the editor.

Begin

To begin writing an assembly language program, pick a filename and enter

EDIT *filename*

Even if no such file exists on the current disk, your editor will work. So type

EDIT TEST1 <Return>

and you'll see a blank screen with a cursor at the top and an inverse ruler at the bottom showing the current tab stops for opcodes, operands, and comments. The screen also shows the name of the file you're editing. Use the arrow keys to move the cursor around and the other keys to type in what you need. The following is a selected list of editor commands.

Keys	Function
Command-arrow key	Jump to top or bottom of page or move up or down one screen.
Esc-E	Enter insert mode. Key combination toggles mode on and off.
Tab	Jumps tab stops. Useful, but not essential, for separating label, opcode, operand, and comment fields.

Esc-E/C	Scrolls screen up/down one line.
Delete	Pressing the key erases the character to the immediate left of the cursor.
Command-Z	Undo single character delete.
Command-Delete	Selects line to be deleted. Pressing arrow keys selects more for deletion. When Return key is pressed, all selected material is deleted. Command-Z won't restore this deletion.
Command-C	Copies selected materials. After pressing this key combination, more material can be selected using arrow keys. Material will be placed in SYSTEMP file when Return key is pressed. The *APW* disk must *not* be write-protected for this to work correctly.
Command-X	Deletes selected materials. After pressing this combination, more material can be selected using arrow keys. Material will be deleted from screen and placed in SYSTEMP file when Return key is pressed. The *APW* disk must *not* be write-protected for this to work correctly.
Command-V	Pastes materials in copy buffer to screen. It does not clear copy buffer. This can be used to recover materials cut by the Command-X function. The *APW* disk must *not* be write-protected for this to work correctly.
Command-L/K	Search down/up. Prompt in the bottom bar will query for string to search. ***String Not Found*** message appears if unable to locate string. Otherwise, cursor moves to search string.
Control-Q	Quit. This takes you to a menu with several options. **<R>** Return to editor **<S>** Save to the same name **<N>** Save to a new name **<L>** Load another file **<E>** Exit without updating

Writing Assembly Language Programs

If you're familiar with other assemblers, there are a few important things to note about *APW*. After entering the editor, all you need do is place a space between fields. However, it's clearer and easier to use the leftmost column for labels, then use a tab stop for the other three fields.

For example, the following shows a typical line with label, opcode, operand, and comments.

```
Label  Opcode  Operand  Comment
Loop   sta     $0400,X  ;Decrement loop
```

To get started, let's look at some required formatting procedures.

KEEP

The first line of a program must use the KEEP command to define the object-code name. This is the name the program will use when you run it.

For example, the line

```
KEEP  GRAPH
```

would name the EXE and object (.ROOT) file of the program, as GRAPH. The source code, the SRC file, can be any name you wish, and it doesn't have to be saved as the name in the program.

For instance, you could save the program source code as CHART, and if the KEEP name was GRAPH, when the code is assembled, the EXE and object files will still be called GRAPH. Since it would be confusing to have a lot of different names, it's easiest to save the SRC file with the .S suffix. Then you won't end up with different source- and object-code names.

Once a program is assembled, there are three files, the source, the object, and EXE file. If you mistakenly name the source and object files with the same name, you'll lose your source code. A correctly developed program on *APW* would have files that would look like the following when assembled and linked with the ASML command:

Filename	Type
GRAPH.S	SRC

```
GRAPH.ROOT  OBJ
GRAPH       EXE
```

The OBJ and EXE files are automatically created by the assembling process.

START

After the *Keep* command, the first line of the source code which will be assembled must be *START.* Also, it must have a unique label in the same line. For example, the following line could be used for the Start line:

```
Label  Opcode
alpha  start
```

The Start pseudo-opcode goes in the opcode field. There's no operand. Any legal label can be used. The label MAIN is a common one used by programmers.

END

At the end of the code to be assembled, there must be an End pseudo-opcode. Unlike Start, no label is required. End usually goes after an *rtl* opcode. The following code shows all of the necessary requirements for a program:

```
      keep  tabs
main  start
      ldx   #$0022
loop  sta   $0400,x  ;Decrement loop
      cpx   #0001
      bne   loop
      rtl
      end
```


It's RTL
On the 6502 microprocessor, it's common to end an assembly listing with RTS (ReTurn from Subroutine.) A JSR—RTS pair is used to call and return from a machine language subroutine located in the same bank of memory as your main program. However, with a 65816 microprocessor, you sometimes need to jump to a subroutine in a different memory bank. This is done with a new set of instructions: JSL (Jump to Subroutine Long) and RTL (ReTurn from subroutine Long). These instructions work just like the JSR—RTS pair, except they allow jumping from one bank to another.

In a single program, each subroutine can be segmented by Start and End. That is, while it's necessary to have at least a single pair of Start and End commands, it's also possible to have several. The Link function ties them together into the entire program during the link process when the assemble and link command (ASML) is given.

For example, the following shows how two subroutines are combined in a single program with two sets of Start and End commands.

```
      keep  dubs
alpha start           ;Begin routine #1
      ldx   #$0022
loop  sta   $0400,x
      cpx   #0001
      bne   loop
      jsr   beta      ;Jump to beta
      rtl             ;Back to ProDOS 16 and out of program
      end

beta  start           ;Begin routine #2
      ldy   #$0004
      rts             ;Back to alpha
      end             ;End of subroutine, not program
```

There's much, much more that can be done with *APW,* but the above material should get you started. See the complete *APW* documentation for instructions on creating macros and other special programming capabilities using the Apple IIGS.

Saving, Assembling and Linking Programs

Once a program has been written in the editor, press Control-Q to quit, and, using the S or N key, save the program under the name listed next to *Filename:* or give it a new name.

When developing a program and debugging it one routine at a time, it's a good idea to use the N option to save to a new name. Then, as each part is tested and debugged, there's a sequential list of source code files. Once you have it fully developed and tested, you can delete the unnecessary files and keep a working sequence of source code files.

After the file is saved, choose E to return to the *APW* command mode.

To assemble the source code into something that will run on your Apple IIGS, such as the music program in Chapter 8, use the ASML command in the *APW* command mode. For example, if you saved the program under the name *Tabs.s*, there should be a file

TABS.S SRC

on your disk when you catalog it with the CAT or CATALOG command. Now type in

ASML TABS.S <Return>

and your code will be assembled and linked. If the Keep name was Tabs, a successful assemblage and linkage produces a ROOT and EXE file. The EXE file is executable and can be run by simply typing the program name and pressing Return. For instance

TABS <Return>

runs the program called TABS.

If there is an error in your source code, *APW* will tell you. Sometimes, however, it will still assemble and link a program and give the correct files. For example, if you change the word START in the Dubs program to NOP (No OPeration, a legal dummy opcode), the first part of the program will bomb and there will be error messages. However, since the Beta routine is perfectly correct, the assembler/linker will go ahead and produce a complete set of files on your disk.

Selected *APW* Commands

APW is very powerful. When you're not using the editor, assembler, or linker, there's a very useful set of commands in the ProDOS 16 operating system under which *APW* runs. The following commands can be accessed when the pound sign (#) prompt is visible on the screen.

Catalog or Cat

Lists the files in the current directory to the screen. If a subdirectory or main directory is not specified, the current directory is listed. If one or more subdirectories are specified, the Catalog command finds the desired directory, even if it's on a disk other than the the current one. For example

CAT /COMPUTE/CH9

lists the files in the subdirectory CH9 found on the disk named COMPUTE.

Compress

Compress can do two things. First, it compresses the files on your disk. This function collects the gaps left in the directory when a file is deleted. Second, it can sort the directory in alphabetical order. If you use the Compress command with no parameters, it prompts you to choose either C (Compress) or A (Alphabetize) or both. If both are used, leave a space between them.

Copy

Copy a file from one disk and/or subdirectory to another. It's actually a lot easier to do this from the desktop. To copy a file to the same disk and to the same directory, specify the file to be copied and the filename to assign to the copy. For example

COPY TUNE TUNE.BKU

makes a copy, named TUNE.BKU, of the file TUNE and places it in the same directory.

To copy from one disk to another, specify the pathnames separated by slashes. There's no need to create a new filename. For instance, to copy the file TONE, which is in the current directory CH9 (part of the path named /COMPUTE/CH9), to the root directory of the disk /APWPROGS, you'd use

COPY TONE /APWPROGS

Only the TONE file is copied, not the whole disk. The file is saved on the second disk under the name TONE.

If you wanted to use a different filename, you could append it to the end of the /APWPROGS pathname, as in

COPY TONE /APWPROGS/TONE2

Now the file TONE has been copied to /APWPROGS under the filename TONE2.

Create

Creates a new subdirectory in the current directory. For example, to create a directory called SAMPLES, the command sequence

CREATE SAMPLES

would do.

Delete

Deletes the specified file from the current directory. Although this erases the file from disk, the disk has not yet been compressed. In other words, the file is still recoverable. If several developing programs have been written, it's possible to use a *wildcard* to erase an entire group of files with the wildcard characters. The wildcard indicator is the equal sign (=), and if placed at the beginning or end of a filename, all files with the indicated attributes will be deleted in the current directory.

DELETE =.BKU

deletes all the files with the extender .BKU.

DELETE TON=

deletes all the files with the first three letters, TON. Filenames TONE, TONY, TONAKA, TONIGHT would be deleted from the current directory.

It's possible to delete a subdirectory, but only if that directory is empty. The files in the subdirectory must be deleted *before* the subdirectory itself is erased.

Dumpobj

This command works something like a memory dump from the monitor, except instead of dumping memory, it dumps the contents of an OBJ file. There are several different options, depending on how you wish to see the object code.

The default dump is called Object Module Format (OMF), but if so desired, it can dump the code in hexadecimal code or 65816 disassembly. For example

DUMPOBJ DUBS.ROOT

uses the default OMF format. Alternatively,

DUMPOBJ +X DUBS.ROOT

dumps hexadecimal code with text in the margins.

The following options are available with DUMPOBJ.

Option Identifier	Option
+X	Hexadecimal dump
+D	65816 disassembly format
-H	With +X option, list headers as hex also
-O	Just show the headers
-F	File type suppress
-M	With +D option, assume the accumulator is in eight-bit mode
-I	With +D option, assume the X and Y registers are in eight-bit mode
-A	Just provide operation codes and operands for OMF and 65816 disassembly listings
-S	Only segment name and type in segment headers

More than a single combination can be used, but each must be separated by a space.

Filetype

This command changes a specified file to a specified type. There are three different ways a file can be specified—by decimal code, by hexadecimal code, or by text code. Since the text code is used in the directory, let's use those. They include:

File Type	Definition
TXT	Text
BIN	ProDOS 8 binary load
DIR	Directory
SRC	Source code
OBJ	Object code
LIB	Library
S16	ProDOS 16 System load
RTL	Run-time library
EXE	Shell load
STR	Startup load
SYS	ProDOS 8 system load

To change a program to a startup file, for example, you would enter

FILETYPE HELLO STR

and the file named HELLO becomes a startup file.

Help

By itself, the Help command lists all available commands to the screen. If a command name follows Help, then information about that command is brought to the screen. To find out more about the Help command, enter

HELP HELP

Init

This command is used to format a disk. It's easier and somewhat safer from the desktop, but again, you can use it from *APW* if you want.

All the command requires is the device number (use SHOW UNITS to see connected devices) and a filename. For instance

INIT .D2 OLDSPOT <Return>

initializes the disk in device .D2 as OLDSPOT. This command completely erases the contents of a disk.

Move

Move lets you organize your disk by moving files from one directory to another. To use the command, specify the file in a current directory and the name of the directory it's to be moved to. For example

MOVE GRAPH PORTFOLIO

moves the file GRAPH to the directory called PORTFOLIO. More complicated moves can be made through several layers of subdirectories as well.

Prefix

This standard ProDOS command sets the current directory.

PREFIX SAMPLES

makes the directory SAMPLES the current directory.

Quit

Exits *APW* and gives the option of rebooting, executing the Start program, or providing the pathname of an alternative start program.

Rename

A standard ProDOS command which lets you change the name of a file. A space between the old name and the new name is required.

RENAME ONE TWO

changes the name of the file called ONE to a file now called TWO.

Show

The show command can display the following:

Option	Function
Language	Current language
Languages	All languages in language table and their numbers
Time	Shows the date and time
Units	Devices connected to system

Switch

The Switch command swaps the positions of two files in the directory. For example

SWITCH APPLES ZEBRAS

switches the directory position of the files APPLES and ZEBRAS.

Type

Lists SRC and TXT files to the screen. Type is a useful command for checking the contents of a source file. Enter the command and a filename. To see the source code file GRAPH.S, for instance, enter

TYPE GRAPH.S

and its contents are listed to the screen.

Appendix D

Selected Non-ToolBox Built-in Graphic Routines

Graphic/Text Soft Switches

Switch Address	Function
$C050 49232	Graphics mode
$C051 49233	Text mode
$C052 49234	All text or graphics
$C053 49235	Mixed text and graphics
$C054 49236	Page 1
$C055 49237	Page 2
$C056 49238	Low-resolution graphics
$C057 49239	High-resolution graphics

Selected Low-Resolution Routines

Subroutine	Address	Notes
PLOT	$F800	A=vertical position 0–$2F Y=horizontal position 0–$27
HLINE	$F819	A=vertical position 0–$2F Y=left horizontal position 0–$27 Address $2C=right horizontal position 0–$27
VLINE	$F828	A=top vertical position 0–$2F Y=horizontal position 0–$27 Address $2D=right horizontal position 0–$27

CLRTOP	$F836	Clears to 40 lines of lo-res screen
SETCOL	$F864	Sets lo-res color
		A register holds color code
		$0=Black
		$1=Deep red
		$2=Dark blue
		$3=Purple
		$4=Dark green
		$5=Dark gray
		$6=Medium blue
		$7=Light blue
		$8=Brown
		$9=Orange
		$A=Light gray
		$B=Pink
		$C=Light green
		$D=Yellow
		$E=Aquamarine
		$F=White

Selected High-Resolution Routines

Subroutine	Address	Notes
HGR	$F3E2	Starts up HGR1
HGR2	$F3D2	Starts up HGR2
BKGRND	$F3F4	Background color from A register (mask)
		Solid color values for background:
$55		Violet
$2A		Green
$D5		Blue
$AA		Orange
$FF		White
$00		Black
HCOLOR	$F6F0	Color 0–7 in X register
HPLOT	$F457	Vertical position A register
		Horizontal low byte X register
		Horizontal high byte Y register
HRLINE	$F53A	Horizontal low byte A register
		Horizontal high byte X register
		Vertical position Y register

Index

[redacted]
KIRWAN QLD. 4814

Great Sound, Great Graphics

The GS in Apple IIGS stands for ***Graphics*** and ***Sound***, the two most advanced features of the newest Apple II personal computer. With exceptional high-resolution graphics and synthesizer-quality sound, the Apple IIGS can paint the screen with more than 4000 colors and make music like a symphony.

But because the graphics and sounds are so sophisticated, using them in your own programs on the IIGS is more complicated than on other Apple II computers. *COMPUTE!'s Guide to Sound and Graphics on the Apple IIGS* shows you how to access and control this machine's impressive power. With a patient approach and clear writing and programming examples, this book takes you on a tour of the IIGS's capabilities, from the Apple II-like low-resolution graphics to the new super high-resolution screens.

William Sanders, author of the *Elementary Apple IIGS* as well as a number of other programming books, shows you step by step how to create your own programs. Here's just some of what's inside:

- Drawing simple graphics on the text screen.
- How to animate shapes to create moving pictures.
- Using the low- and high-resolution graphics modes.
- Making shapes, shape tables, and creating bitmapped graphics.
- Generating terrific-looking graphs and charts on the IIGS.
- Accessing the new super high-resolution mode of the computer.
- Tweaking the speaker for simple sounds.
- Getting to the Apple IIGS Toolbox to create dazzling sounds.

Scores of program examples—in both BASIC and machine language—illustrate each concept. You can alter the programs at will to make new designs and to explore new areas.

COMPUTE!'s Guide to Sound and Graphics on the Apple IIGS is an instant education in programming on this new and powerful Apple personal computer.

51695

9 780874 550962