1640

# COMMODORE 64 GRAPHICS & SOUND PROGRAMMING

**Explore the amazing graphics and sound potential of your micro . . . with 68 ready-to-run programs!**

BY STAN KRUTE

# COMMODORE 64
# GRAPHICS & SOUND PROGRAMMING

# COMMODORE 64 GRAPHICS & SOUND PROGRAMMING

BY STAN KRUTE

TAB BOOKS Inc.
BLUE RIDGE SUMMIT PA. 17214

To Char, Lady of Magic

FIRST EDITION

FIRST PRINTING



Printed in the United States of America



Library of Congress Cataloging in Publication Data

Krute, Stan.
Commodore 64 graphics and sound programming.

Includes index.
1. Computer graphics. 2. Speech processing systems.
3. Commodore 64 (Computer)—Programming. I. Title.
T385.K77 1983 001.64'43 83-4958
ISBN 0-8306-0140-6
ISBN 0-8306-0640-8 (pbk.)

Cover photograph by the Ziegler Photography Studio of Waynesboro, PA.

# Contents

# List of Programs

# Preface

I confess that I really like computers. They're enjoyable tools. I would like to see two improvements: lower prices and higher quality graphics and sound.

The friendly folks at Commodore keep giving the industry a shove in the right direction. First came the VIC-20, a marvel just two years ago. Now comes the Commodore 64. It offers powerful graphics and sound capabilities, a big hunk of memory, flexible hardware, and a price that's giving the competitors ulcers.

I spend a lot of time teaching people about computers. When I work with kids, they all want to learn how to make pictures and noises. Since this matches my own inclinations, things work out well. From the moment I heard rumors about the 64, I knew I wanted to learn its tricks and share them with others.

For the last eight months, I've had the pleasure of exploring graphics and sound on the Commodore 64. This book lets you in on some of my discoveries. It mixes computers with art, music, logic, and puzzles. I hope it encourages you to launch out on your own creative journeys. The Commodore 64 is a sturdy little vehicle for such enterprises. Use it well, and then share the wonderment with others.

# Introduction

This book is written for the advanced beginner/intermediate level programmer who wants to start learning about graphics and sound on the Commodore 64 computer. The book covers a large subset of the machine's abilities in these two exciting areas. The 68 programs are all written in a clear, clean BASIC.

The only other available book that covers the same ground is Commodore's own *Commodore 64 Programmer's Reference Guide*. It's a great book, one you'll probably want on your bookshelf if you get hooked on this stuff. The only drawback is that it's a bit advanced for most people—intimidating, actually. When you finish the volume you're holding in your hands, you should be able to go at the Commodore book without a paid interpreter.

You'll need a Commodore 64 computer, a good-quality TV set and some kind of program storage device to use the programs in this book. If you appreciate your eyesight, pick up a nice computer monitor. Commodore's color monitor is an excellent unit. For program storage, Commodore's tape recorder works just fine. A disk drive is a luxury you'll want to add to your computer system if you get serious about programming. Commodore's 1541 drive is low-priced and solidly built, although it does have a tendency to heat up and get a bit weird during very long (12+ hours) programming sessions.

The first six chapters of this book cover graphics. You'll learn about sprites, character graphics, and bit mapped graphics. The Commodore 64 makes this kind of programming easier than any other machine currently on the market. You can get exciting images with programs written in BASIC, thanks to the powerful hardware packed into the C-64.

The next three chapters cover sound making on the Commodore 64. The 64's sound chip is a remarkably complete three-voice music synthesizer. Once again, you'll get results from BASIC programs that would require advanced assembly language skills on other popular computers. Finally, in Chapter 10, you'll learn how to bring graphics and sound together.

I believe people learn to program by example and by doing. This book has 63 programs, over half of which are discussed extensively in the text. Each chapter closes with a brief summary and a set of exercises designed to clarify important points. I've included 30 programming problems for you to tackle, with a complete set of possible solutions.

Some of this material can be confusing at first. I've tried to provide lots of figures, charts, and helpful appendices to get you out of tight spots. I know how frustrating it can be when a book tantalizes your interest and then leaves you lost in a forest of impenetrable jargon. I've also provided special coding forms you can copy and then use to design your own sprites and custom characters.

Even though the programs are written in BASIC, I've made every effort to keep them clean and modular. I spend a lot of programming time working with Pascal and assembly language and have tried to bring some of the discipline involved in these languages to the examples. Computer people can get a bit dogmatic about languages. BASIC is simple and quick for beginners, and you can write well in it if you work carefully.

One style of programming I've tried to avoid in the examples is what I call squashed spaghetti code. That's the kind of programming in which every line is packed with tricks, GOTOs, cryptic variable names, and unrelated statements. Supposedly, such code leads to blinding bursts of performance and speed. Hog hooey. If you want real speed, come up with a better algorithm, or translate parts of the program to machine language.

Enough of the preliminaries: you're about to embark on an adventure. Breathe deeply, stay calm, and have a ball.

## HOW TO USE THIS BOOK

If you haven't used a Commodore 64 very much, I suggest you do so now. Go through the first few chapters of the *Commodore 64 User's Guide*, which comes with the machine. It's a good introduction to your computer's fundamental operations. If you haven't spent much time programming in BASIC, pick up one of the excellent introductory books on that language and go through it. Come on back when you've got these preliminaries taken care of.

This book is designed for active, hands-on learning. It's really an explorer's toolkit, complete with explanations of various topics, lots of programming examples and exercises, and a supply of useful reference materials.

The ten chapters share a similar structure. Each revolves around three to six related topics. A short introduction usually introduces each topic. Then comes a programming example for you to run on your computer. A detailed discussion of the example comes next, followed by suggestions for modifying the original. At the end of the chapter you'll find short review questions and several programming exercises. Answers to the questions and possible solutions to the exercises are provided.

By using the order form at the back of the book, you can buy a disk or tape that will relieve you of the chore of typing in the example programs; just load them from the disk and then run them.

If you don't purchase disk or tape, you'll need to type in the programs by hand. It's a pretty straightforward process: Simply type in what you see in the printed listing. The only problem you may have is when you run into a display icon.

Let me explain. The Commodore 64 gives you extensive control over where and how information gets displayed on the screen. Among other things, you can easily move the cursor, clear the screen, change the color of the characters, and display them in reversed colors. You can do these things right from the keyboard, as covered in pages 14 through 17 of the *Commodore 64 User's Guide*. More exciting, you can do them from inside a program.

How? You can set up a string constant containing the display commands. Just type them inside quotes, either in an assignment statement or a print statement. When the string is displayed, the display commands work just as if they'd been typed from the keyboard.

The problem arises when you type or list a program that uses this technique. The display commands show up in strange ways: they are printed as reversed letters and graphics characters. For example, clearing the screen shows up as a reversed heart, and moving the cursor to the left, as a reversed vertical line. I call these display icons.

When you see one of these display icons in a program listing, you've got to figure out which display command it represents and which keys to press to obtain it. The chart in Fig. 1-1 reveals everything. It shows all the display icons I've used in this book, the keys to press to get them, and the commands they represent. If you come to an assignment or print statement with an incongruous character showing inside quotes, refer back to this chart.

One more pointer for those of you who'll be typing in the example programs by hand: save each program on tape or disk before you run it. That way, if you make a typing error that crashes the system, you won't have to retype the whole thing.

If the program you've just typed in doesn't run, you'll have to search it for a typing error. Examine any statements the computer complains about. Then use the 64's wondrous screen editor to make the needed changes. If the program still doesn't run, go over it again line by line against the original. If worst comes to worst, retype the lines the 64 balks at, even if they look right. Eventually you'll get it going.

Once a program's running, whether loaded from a disk or tape or typed by hand, watch it for a while. Then watch it some more, this time referring back to the printed listing. Try to figure out which program lines are controlling particular pictures and sounds. Then come on back to this book and read the detailed discussion of the program.

Then comes the real fun: modification. Load the program in again (you saved a working version, of course). Change a print statement here, a loop counter there. See what

## COLOR ICONS

| Icon | Key(s) to press | What it does | Icon | Key(s) to press | What it does |
|---|---|---|---|---|---|
| | CTRL-1 | Text color black | | C= –1 | Text color orange |
| | CTRL-2 | Text color white | | C= –2 | Text color brown |
| | CTRL-3 | Text color red | | C= –3 | Text color light red |
| | CTRL-4 | Text color cyan | | C= –4 | Text color dark gray |
| | CTRL-5 | Text color purple | | C= –5 | Text color medium gray |
| | CTRL-6 | Text color green | | C= –6 | Text color light green |
| | CTRL-7 | Text color blue | | C= –7 | Text color light blue |
| | CTRL-8 | Text color yellow | | C= –8 | Text color light gray |

## OTHER ICONS

| Icon | Key(s) to press | What it does | Icon | Key(s) to press | What it does |
|---|---|---|---|---|---|
| | CLR/home | Cursor home | | Shift–CLR/home | Clear screen |
| | CRSR (up/down) | Cursor down | | Shift–CRSR (up/down) | Cursor up |
| | CRSR (left/right) | Cursor right | | Shift–CRSR (left/right) | Cursor left |
| | CTRL–9 | Reverse on | | CTRL-0 | Reverse off |

Fig. I-1. Commodore 64 Display Icons.

happens when you rerun the program. Make some more changes. Switch a color code, shift a shape. Run the program again. This is an excellent way to learn the essence of the graphics and sound techniques.

If you want to get really good at programming graphics and sound, you need to spend a lot of time at it. Come up with an exciting image or sound; then try to write a program from scratch that pulls it off. Push the machine to its limits and then go beyond them.

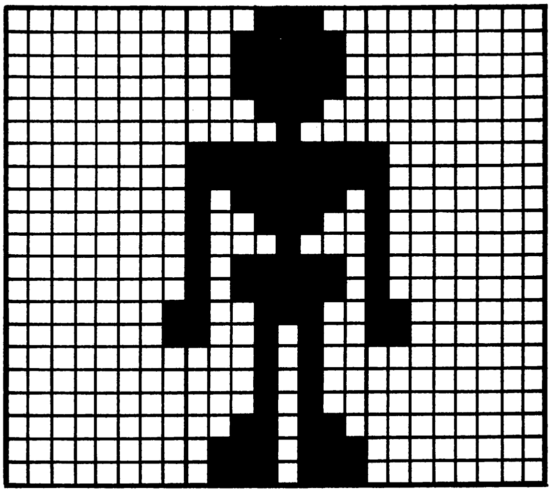
Start writing some longer programs that make use of a variety of graphics and sound techniques. Try things that other people consider useless or impossible. Spend some time reading any published programs you can get your hands on. Try to figure out why the programmer did something a certain way and see if you can come up with a better way. Wander through this book's appendices and figures, and do the same with other books. Watch the computer magazines for interesting articles. Pick up on other people's ideas; then come up with some of your own.

A couple of final thoughts: First, the Commodore 64 starts up with an unreadable blue-on-blue display. I immediately change this to white on black on medium gray by pressing CTRL-2 and typing in these two commands:

```
POKE 53280,12
POKE 53281, 0
```

Second, if you're using the disk drive, you'll find the disk operating commands to be a bit clumsy. Read up in your disk operating manual about the DOS Wedge program and use it whenever you start a session. It will give you disk commands that are more powerful, more versatile, and a lot easier to type in.

# Chapter 1



# A First Look at Sprites

This chapter introduces one of the Commodore 64's most powerful features: sprites. You'll learn how to make a sprite and move it around on the screen. You'll also learn how to change the size and color of your sprite picture.

## 1.1 WHAT'S A SPRITE?

Turn on the TV set. Put your eyes six inches from the screen. You see small dots or rectangles of light. Television pictures are made up of hundreds of thousands of these little pieces.

The smallest dot a computer can put on the TV screen is called a *pixel*. That's short for picture element. A sprite is a pattern of pixels that your Commodore 64 can move around on the screen.

A basic sprite pattern is 24 pixels across and 21 pixels high. Take a look at Fig. 1-1. If you multiply the 21 rows by the 24 columns, you find a total of 504 pixels to play with. If you don't trust multiplication, count the boxes.

In a simple sprite pattern, you can arrange things so that any particular pixel shows up or is invisible. You can see an example of this in Fig. 1-2. You can create many different pictures using those 504 pixels—about 2,207, 107, 920, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, of them: ample room for a touch of creativity.

## 1.2 DEFINING A SPRITE PATTERN

You need a way to tell the computer which pixels in a sprite pattern should show up and which ones should stay invisible. This is done with number codes and groups of eight pixels.

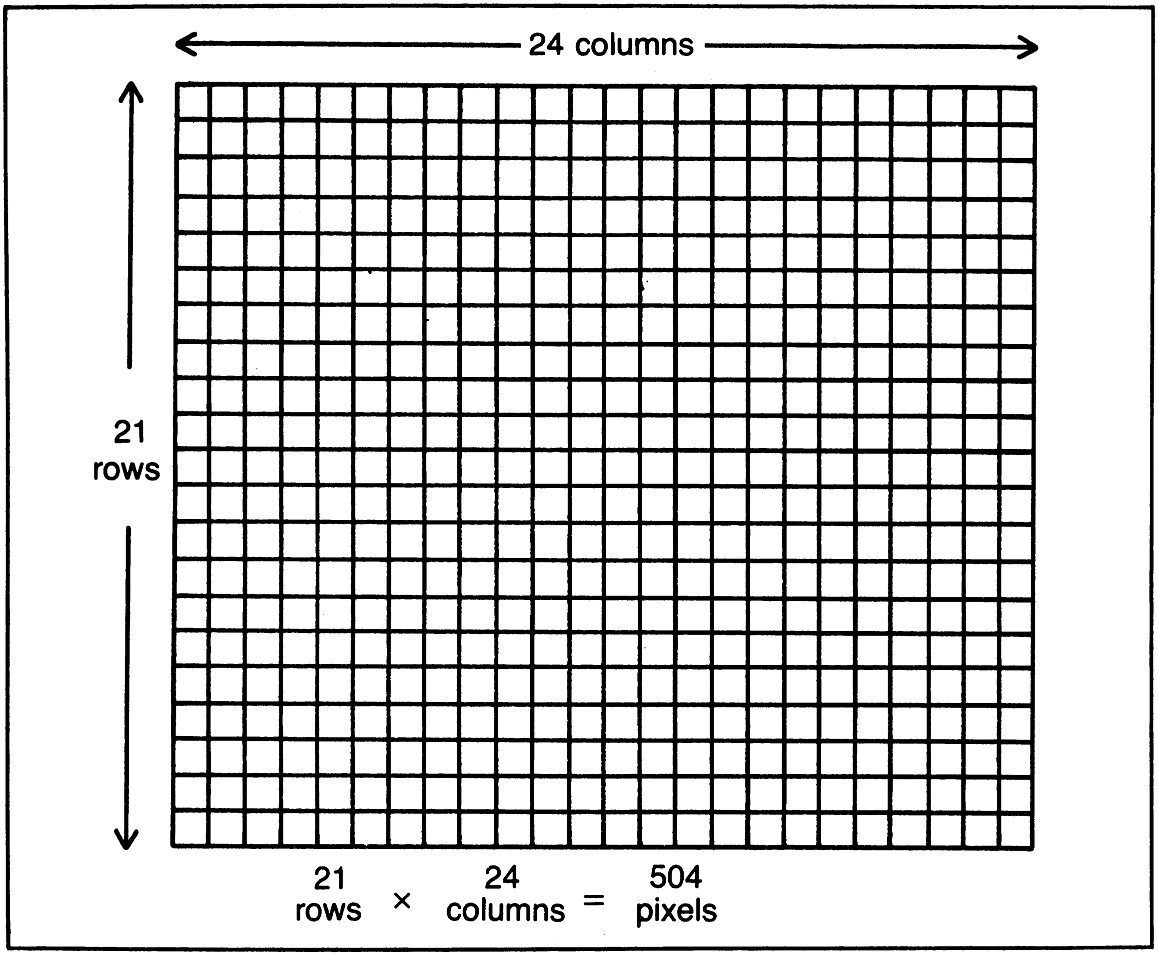



Fig. 1-1. A basic Commodore 64 sprite pattern covers 504 pixels.

Take a look at Fig. 1-3. Each of the eight boxes represents a pixel and has a number above it. The number gives the pixel a value. For example, the leftmost pixel has a value of 128. The rightmost pixel has a value of 1, and so on.

Now take a look at Fig. 1-4. Some of the boxes have been filled in. If you add up the values of the filled-in boxes, you get the number 85: 64 + 16 + 4 + 1 = 85. Figure 1-5 shows some more examples of how filled-in pixel patterns are turned into number codes.

Examine the special sprite coding form shown in Fig. 1-6. It has the required 24 columns and 21 rows. Each row is split into three parts for number coding, each part having eight columns. At the top of each column is that column's number coding value. Each row will

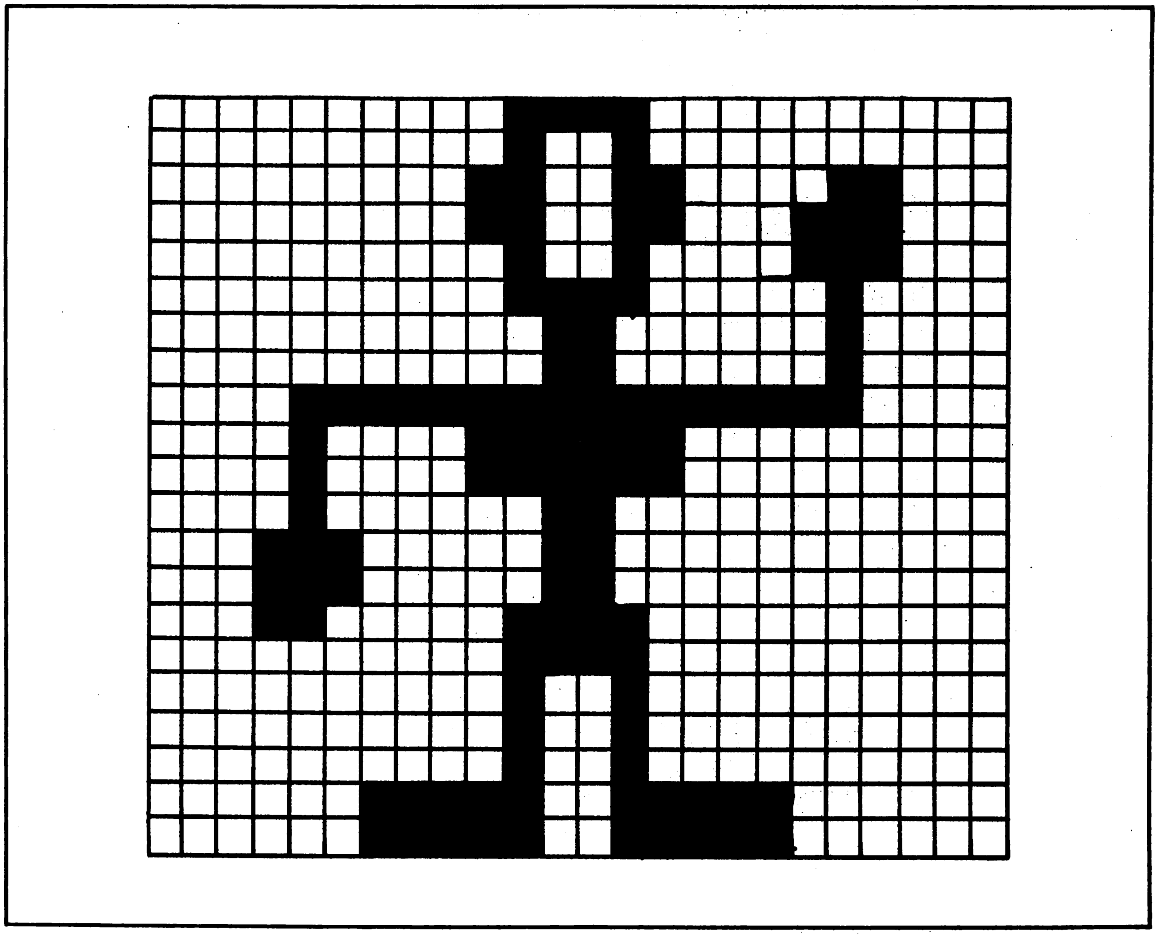


Fig. 1-2. A picture made by making some of the pixels in a sprite pattern visible.

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

Fig. 1-3. Values used to code a group of eight pixels.

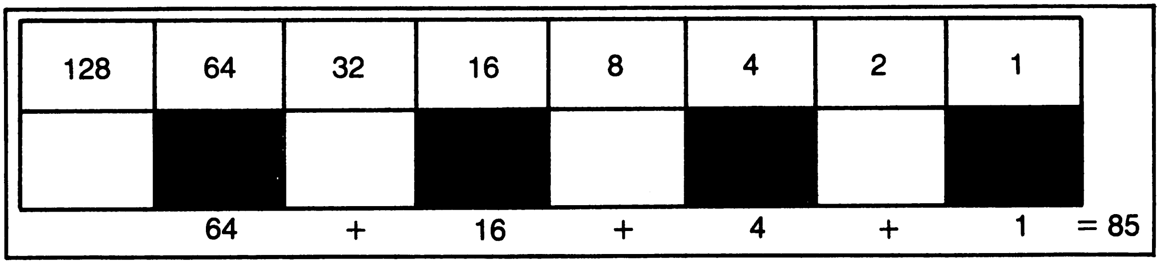



Fig. 1-4. Coding a pattern of eight pixels.

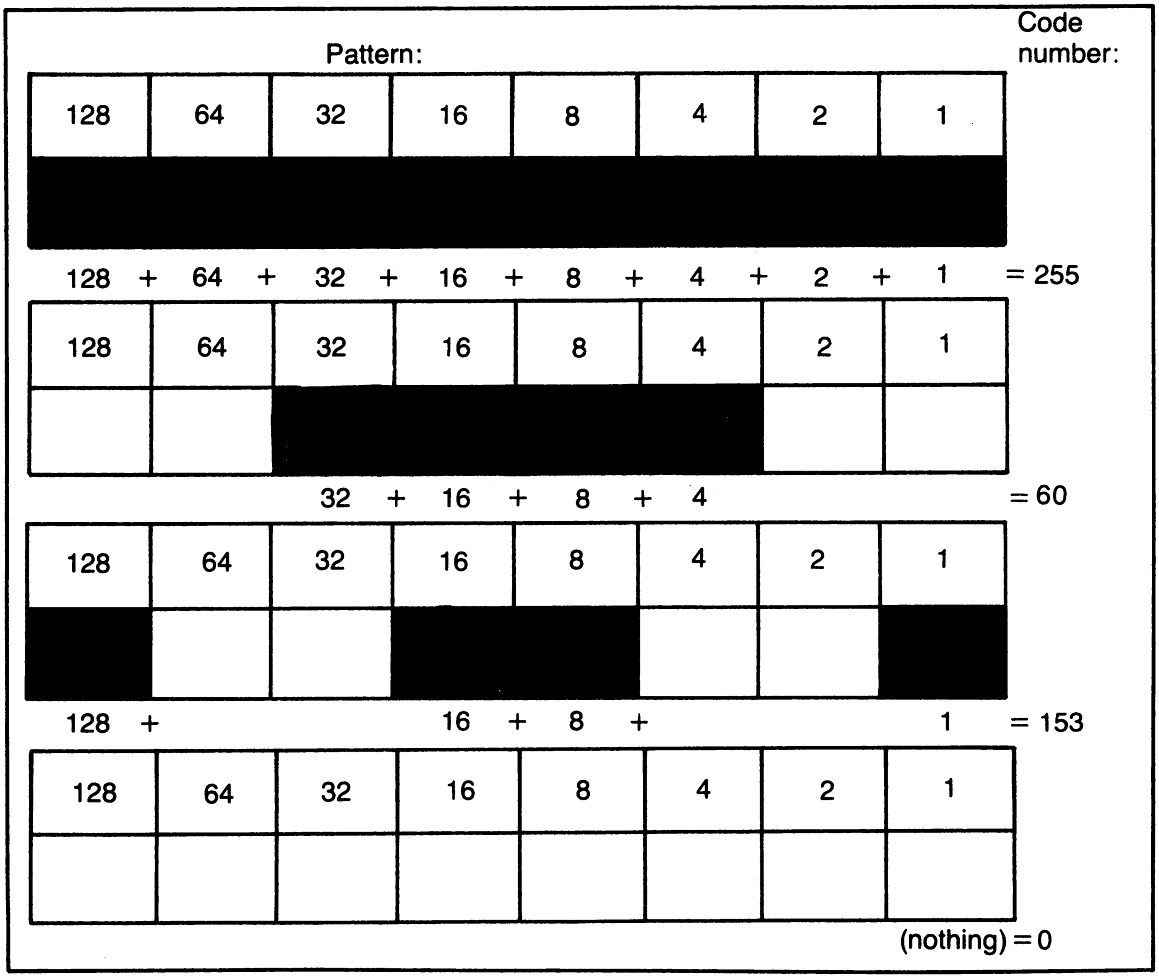



Fig. 1-5. More examples of coding eight-pixel patterns.

| Column number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | Number codes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Values | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | | | |
| Row 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 4 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 6 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 7 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 8 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 9 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 10 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 11 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 12 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 13 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 14 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 15 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 16 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 17 | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Fig. 1-6. A special sprite coding form.

turn into three code numbers, one for every group of eight columns in that row. Since there are 21 rows, you'll end up with 63 code numbers. The code numbers must be put into the Commodore 64 in the proper order: from left to right in each row, starting with the top row and ending with the bottom row.

Here are four steps you need to follow to define a sprite pattern:

1. Make a copy of the sprite coding form.
2. Draw a design by filling in the boxes representing pixels you want to show up.
3. Figure out the 63 number codes, one for each group of eight pixels.
4. Enter the code numbers into the computer in the proper order.

Figure 1-7 shows a filled-in sprite coding form for a friendly little creature. Take a good look, making sure you understand how I figured the number codes. Skim over the last few pages again until things make some sense. Even the brightest computer users, using the clearest of instructions, find that they usually have to read things over many times.

Now it's your turn. Zip out to the nearest copying machine and make some copies of the special sprite coding form. Then draw some sprite designs. When you have one that you like, figure out the 63 number codes. You'll use these codes later in this chapter. Then take a little refreshment break. Come on back to the book when you're ready for some action.

## 1.3 YOUR FIRST SPRITE PROGRAM

You'll start out with a simple program that displays a simple sprite. Figure 1-8 shows the major steps of the program. Figure 1-9 provides a listing of the actual program A Simple Sprite.

Look the figures over carefully. Then type the program in on your Commodore 64. If you don't know how to get the graphics icons on line 160, refer back to How To Use This Book in the Introduction. Make sure you save the program on tape or disk when you're done typing. Then run it. Press any key to end the program.

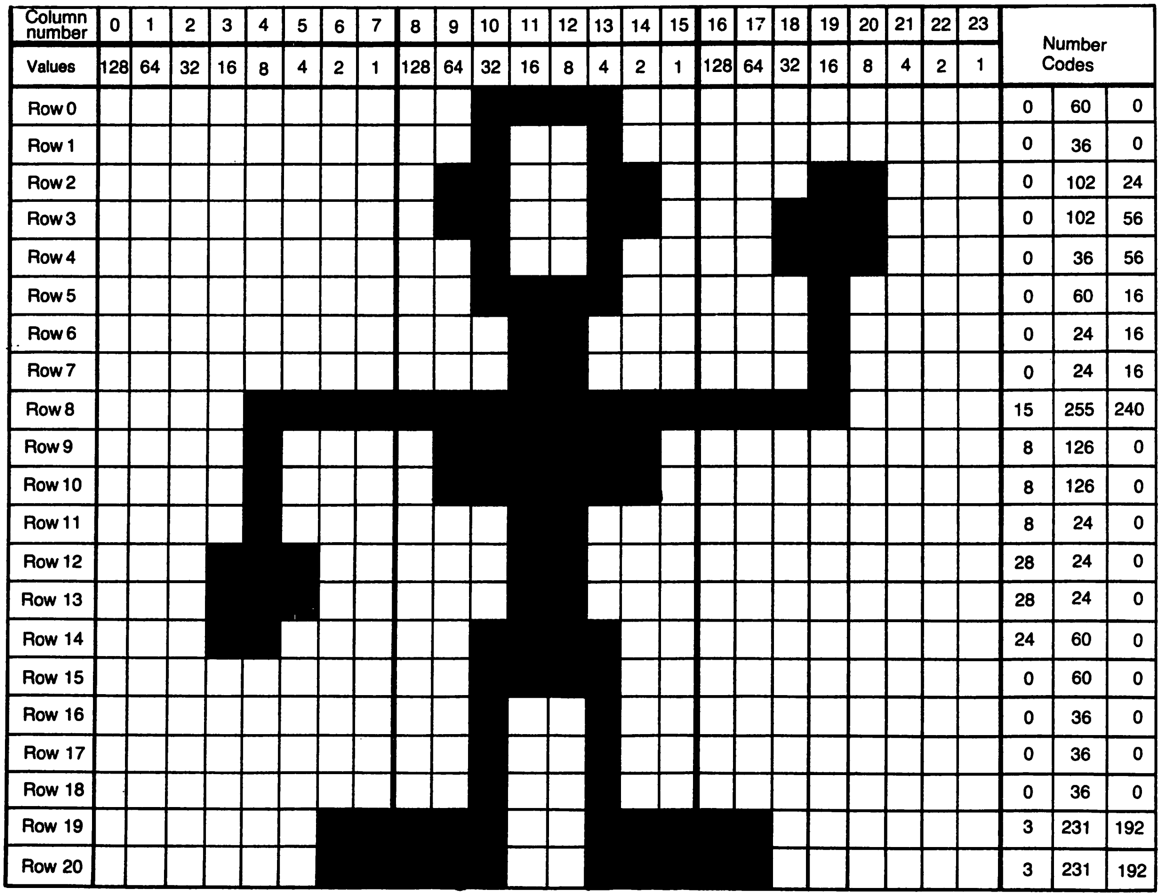



Fig. 1-7. Example of a filled-in sprite coding form.

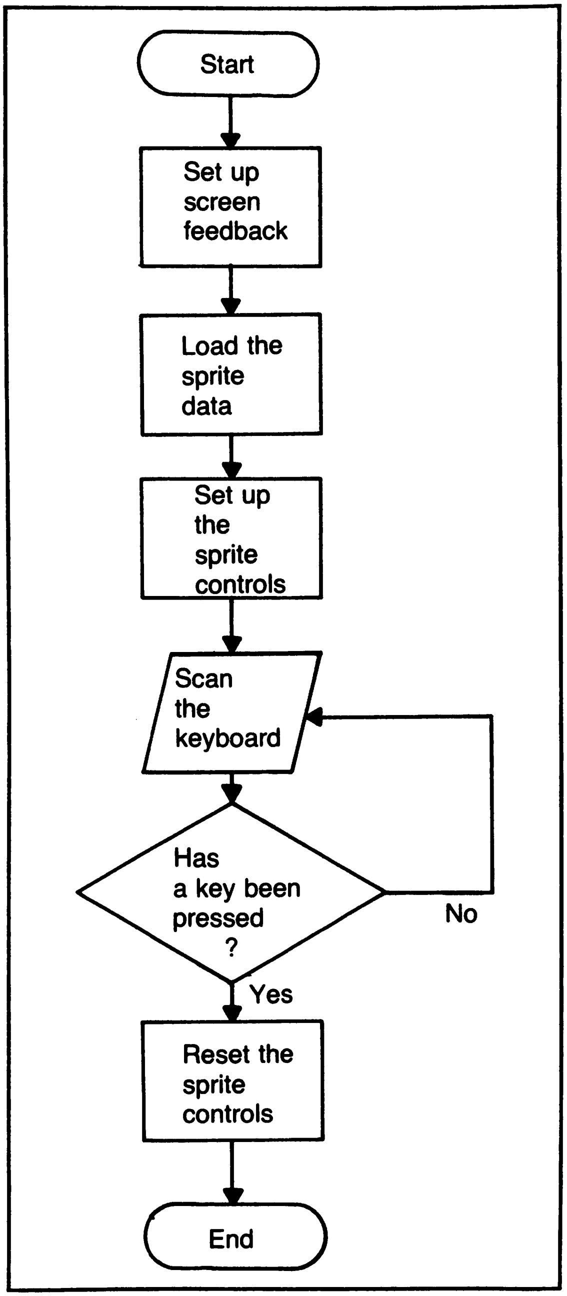



Fig. 1-8. The major steps in the program A Simple Sprite.

### 1.3.1 The Program

Examine this first simple program. The first active section is line 1050.

```
1050 PRINT "[illegible]THINKING ";
```

This BASIC statement clears the TV screen, drops down several screen lines, and prints the message, THINKING. There's nothing more nerve-wracking than a program that shows no sight of activity while it's loading information.

The second program section, lines 1100-1120, loads in 63 sprite data number codes.

```
1100 FOR N = 896 TO 958
1110 :    POKE N, 255
1120 NEXT N
```

To simplify this first sprite program, I designed the simplest visible sprite: one with every pixel turned on. That way, all 63 codes are the same number: 255. The loop in lines 1100-1120 places this code number in 63 consecutive memory locations, addresses 896 through 958.

The third program section, lines 1170-1240, is the workhorse of this program. Look at lines 1170-1200 first:

```
1170 PRINT "[illegible]";          :REM CLEAR SCREEN
1180 POKE 2040,14        :REM POINT TO DATA
1200 VIC = 53248         :REM GRAPHICS CHIP
```

Line 1170 clears the screen. Line 1180 then tells the computer that the sprite data is at locations 896 through 958. How does it do that?

Your Commodore can actually display 8 sprites at a time. They're numbered 0 through 7. When you tell the computer to display sprite #0, it first goes to location 2040 to find out where the pixel number codes for sprite #0 are located. It takes the number it finds there and multiplies it by 64. In this case, it will multiply 14 by 64 and get 896. And that's for the sprite data you stuffed into the machine—pretty slick.

Line 1200 then sets up a variable named VIC, and gives it the value 53248. Who or what is this VIC, anyway?

### 1.3.2 A Little VIC-II Detour

The heart of the Commodore 64's incredible graphics capabilities is a small integrated circuit. It's officially called the 6567 Video Interface Chip—VIC-II for short. (The first VIC was the 6560 chip, used in the VIC-20 computer.) This hardworking gadget puts out several kinds of pictures: the 40 column by 25 line text display, a 320 pixels wide by 200 pixels tall high resolution graphics display, and 8 sprites. If it wouldn't void the warranty, those of us who survived the early days of personal computer graphics would open the box and kiss this chip.

By poking certain numbers into some of the locations inside the VIC-II chip, you can control it. There are 47 addressable locations in the VIC-II chip. These locations are also called registers. The VIC-II registers start at memory address 53248 of the Commodore 64 and go up through address 53294. Appendix A gives more information about the VIC-II registers.

### 1.3.3 Back To The Program

So, line 1200 sets the variable VIC to 53248. You can then get the address of any of the 47 VIC-II registers by adding the register number to the value of VIC. Take a look at the last four lines of our workhorse section:

```
1210 POKE VIC,170    :REM HORIZONTAL POS
1220 POKE VIC+1,120  :REM VERTICAL POS
1230 POKE VIC+39,13  :REM COLOR IT GREEN
1240 POKE VIC+21,1   :REM SPRITE #0 ON
```

Register 0 controls the horizontal position of sprite #0. Line 1210 of our program sets this to 170, about halfway across the screen. Register 1 controls the vertical position of sprite #0. Line 1220 sets this to 120, which is about halfway down the screen. Register 39 of the VIC-II chip sets the color for the pixels of sprite #0 that you want to show up. Color 13 is light green. Take a look at Appendix F for a list of other available colors.

Okay, you've put in the number codes that tell which pixels should show up and told the computer where the codes are. You've given sprite #0 a horizontal and a vertical position. You've also set its color. Now, you just need to tell the VIC-II chip to display sprite #0. Line 1240 does the trick. Register 21 is used to turn sprites on and off. By poking a 1 into it, sprite #0 appears on the screen.

Here's the fourth module of our program:

```
1290 GET KP$
1300 IF KP$ = "" THEN 1290
```

Line 1290 reads the computer's keyboard. Line 1300 tests to see if any key has been pressed. If not, the program just goes back to Line 1290 to read the keyboard again. When a key is finally pressed, the program moves on to a tidy finish.

It's always a good practice to leave things the way you found them, especially when you're programming a computer. Lines 1350-1380 reset the changed VIC-II registers to 0:

```
1350 POKE VIC+21,0 :REM  REVERSE THE
1360 POKE VIC+39,0 :REM  ORDER USED TO
```

```
1370 POKE UIC+1,0   :REM  SET THE SPRITE
1380 POKE UIC,0     :REM  CONTROLS
```

Notice how the order of resetting the registers is the reverse of the setting order.

```
1000 REM *** A SIMPLE SPRITE ***
1010 :
1020 :
1030 REM ** SET UP SCREEN FEEDBACK
1040 :
1050 PRINT "[control characters]THINKING ";
1060 :
1070 :
1080 REM ** LOAD THE SPRITE DATA
1090 :
1100 FOR N = 896 TO 958
1110 :   POKE N, 255
1120 NEXT N
1130 :
1140 :
1150 REM ** SET UP THE SPRITE CONTROLS
1160 :
1170 PRINT "[control character]";     :REM CLEAR SCREEN
1180 POKE 2040,14    :REM POINT TO DATA
1190 :
1200 UIC = 53248     :REM GRAPHICS CHIP
1210 POKE UIC,170    :REM HORIZONTAL POS
1220 POKE UIC+1,120  :REM VERTICAL POS
1230 POKE UIC+39,13  :REM COLOR IT GREEN
1240 POKE UIC+21,1   :REM SPRITE #0 ON
1250 :
1260 :
1270 REM ** WAIT FOR KEYPRESS TO END
1280 :
1290 GET KP$
1300 IF KP$ = "" THEN 1290
1310 :
1320 :
1330 REM ** RESET THE SPRITE CONTROLS
1340 :
1350 POKE UIC+21,0 :REM  REVERSE THE
1360 POKE UIC+39,0 :REM  ORDER USED TO
1370 POKE UIC+1,0  :REM  SET THE SPRITE
1380 POKE UIC,0    :REM  CONTROLS
1390 :
1400 END
```

Fig. 1-9. Listing of the program A Simple Sprite.

## 1.4 SOME PLAY AND EXPLORATION

One of the best ways to learn more about sprite graphics is to play with some of the numbers in this first program. Make a change or two in the program, and then run it to see what happens. Here are a few suggestions to get you going:

Change the number code that's poked in line 1110.

Change the horizontal and vertical position settings in lines 1210 and 1220.

Change the color code in line 1230.

## 1.5 MORE ABOUT POSITIONING THE SPRITE

When you position a sprite, you're really telling the computer where the sprite's upper-left corner should be placed. The normal Commodore 64 display screen shows 320 horizontal positions and 200 vertical positions. With the VIC-II position registers, you can put a sprite in any one of 512 horizontal positions and 256 vertical positions. That way, you can

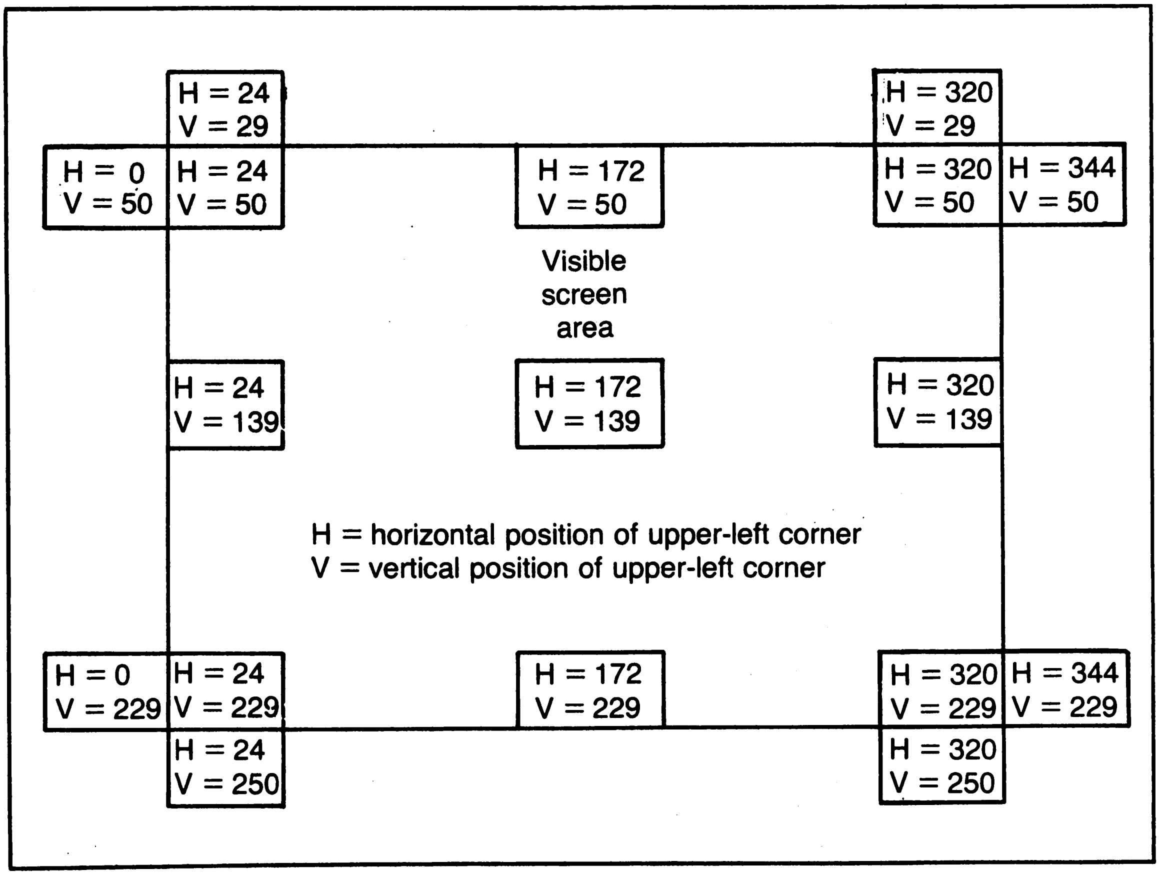



Fig. 1-10. Some important horizontal and vertical position settings for normal-sized sprites.

```
1000 REM *** A SPRITE YO-YO
1220 POKE VIC+1,80   :REM VERTICAL POS
1251 :
1252 REM ** DOWN, THEN UP
1253 :
1254 FOR VP = 80 TO 200
1255 :    POKE VIC+1,VP
1256 NEXT VP
1257 :
1258 FOR VP = 199 TO 81 STEP -1
1259 :    POKE VIC+1,VP
1260 NEXT VP
1261 :
1262 :
1300 IF KP$ = "" THEN 1254
```

Fig. 1-11. Changes and additions that turn A Simple Sprite into the program A Sprite Yo-Yo.

have sprites move smoothly on and off the screen.

Take a good look at Fig. 1-10. It shows the horizontal and vertical sprite position settings that place a sprite in some of the more extreme screen locations. For example, vertical position settings of 29 or less keep a sprite just above the screen viewing area. Horizontal settings between 24 and 320 keep a sprite completely inside the horizontal viewing area, and so on.

## 1-6 A SPRITE YO-YO

Let's play a bit. Load in A Simple Sprite, listed in Fig. 1-9, again. Then type in the lines shown in Fig. 1-11. You're changing a few lines and adding some totally new ones. Be sure to use the line numbers shown. When you're done, save and run the new program.

How did you get the sprite to move like a yo-yo? Look at lines 1254-1256:

```
1254 FOR VP = 80 TO 200
1255 :    POKE VIC+1,VP
1256 NEXT VP
```

This loop tells the computer to change the sprite's vertical position from 80 to 200, one step at a time. The sprite moves down the screen. Then lines 1258-1260 change the vertical position from 199 to 81, again one step at a time:

```
1258 FOR VP = 199 TO 81 STEP -1
1259 :    POKE VIC+1,VP
1260 NEXT VP
```

The sprite moves up. Finally, the new version of line 1300 tells the computer to go back to the top of the yo-yo circuit, at line 1254, if no key has been pressed.

```
1300 IF KP$ = "" THEN 1254
```

## 1.7 DEALING WITH 512 HORIZONTAL POSITIONS

Sharp-eyed readers may have had a question when they read Section 1.5 and looked at Fig. 1-10. Since you can only store numbers between 0 and 255 when you poke information into a memory location, how can you set a sprite's horizontal position to numbers larger than 255?

The VIC-II chip solves this problem by giving you two registers for each sprite's horizontal position. The second register is actually a miniature register and can only hold either a zero or a one. When you want a sprite to be at a position greater than 255, you put a one in that sprite's second horizontal register. Then the sprite's position will be 256 plus whatever number is in its first horizontal register. For example, if a sprite's first horizontal register contains the number 33, and its second horizontal register contains the number 1, the sprite will be at position (256 + 33), or 289. If the second register contains a zero, the sprite's position is based solely on the number in its first horizontal register, with nothing added on. Figure 1-12 gives some examples that show how a sprite's horizontal position can go from 0 through 511.

## 1.8 NOW FOR SOME SIDEWAYS MOTION

Consider the first program, A Simple Sprite, which was listed in Fig. 1-9. Load it in, again then type in the changes and additions shown in Fig. 1-13. Save your new program, and then run it.

Before engaging in a detailed discussion of how the new program works, let's take a little excursion into the world of truth.

### 1.8.1 Coding for True and False

When you try to move a sprite to a new horizontal position, you first must ask if this statement is true or false: "The new position is larger than 255." Depending on the answer,

| If a sprite's first horizontal register is set to . . . | . . . and its second horizontal register is set to . . . | . . . then it will be at horizontal position: |
|---|---|---|
| 0 | 0 | 0 |
| 24 | 0 | 24 |
| 125 | 0 | 125 |
| 255 | 0 | 255 |
| 0 | 1 | 256 |
| 20 | 1 | 276 |
| 64 | 1 | 320 |
| 88 | 1 | 344 |
| 255 | 1 | 511 |

Fig. 1-12. Setting the horizontal registers for some sprite positions between 0 and 511.

```
1000 REM *** SIDEWAYS SPRITE
1210 POKE VIC,64     :REM HORIZONTAL POS
1220 POKE VIC+1,139 :REM VERTICAL POS
1251 :
1252 REM ** RIGHT, THEN LEFT
1253 :
1254 FOR HP = 64 TO 280 STEP 2
1255 :    SF = (HP > 255)
1256 :    POKE VIC,HP + (SF * 256)
1257 :    POKE VIC+16, SF * (-1)
1258 NEXT HP
1259 :
1260 FOR HP = 278 TO 66 STEP -2
1261 :    SF = (HP > 255)
1262 :    POKE VIC, HP + (SF * 256)
1263 :    POKE VIC+16, SF * (-1)
1264 NEXT HP
1265 :
1266 :
1300 IF KP$ = "" THEN 1254
```

Fig. 1-13. Changes and additions that turn A Simple Sprite into the program Sideways Sprite.

you'll put different numbers in the sprite's horizontal position registers.

In Commodore 64 BASIC, you can ask a true-false question and give the answer a special code that stands for true or false. The code for true is −1, and the code for false is 0. Here's an example in BASIC:

```
100 LET AN = (5 > 3)
```

Since 5 *is* greater than 3, the expression

$$(5 > 3)$$

is true. The variable AN will be given the value −1. Here's another example:

```
200 LET XZ = (36 = 21)
```

Since 36 does not equal 21, the expression

$$(36 = 21)$$

is false, and XZ will be given the value 0.

### 1.8.2 Back to the Program: Move to the Right

Now let's see how you got the sprite to move from side to side. Lines 1210-1220 were changed a bit:

```
1210 POKE VIC,170   :REM HORIZONTAL POS
1220 POKE VIC+1,139 :REM VERTICAL POS
```

This starts the sprite out at a new position.

Now take a look at lines 1254-1258:

```
1254 FOR HP = 64 TO 280 STEP 2
1255 :    SF = (HP > 255)
```

```
1256 :    POKE VIC,HP + (SF * 256)
1257 :    POKE VIC+16, SF * (-1)
1258 NEXT HP
```

Lines 1254 and 1258 set up a loop that will run the sprite's horizontal position from 64 up through 280, in steps of 2. Each time through the loop, line 1255 will figure out if the new position is greater than 255. Then, depending on that answer, lines 1256 and 1257 will set the new position.

For example, let's say HP has the value 125. Then line 1255 will set SF (size factor) to 0. Line 1256 will poke the sprite's first horizontal register with 125 + (0 × 256), which is just plain old 125. Line 1257 will poke the sprite's second horizontal register with −1 × 0, or 0. These are the correct pokes for a position less than 256.

Now let's try these formulas on a position larger than 255. Suppose HP has the value 276. Then line 1255 will set SF to −1. Line 1256 will then poke the sprite's first horizontal register with 276 + (−1 × 256), which is 276 − 256, or 20. Line 1257 then pokes the sprite's second horizontal register with −1 × −1, or 1. Once again, the formulas poked the correct values into the horizontal position registers.

### 1.8.3 And Then Move to the Left

If you're not too clear on the explanation of Lines 1254-1258, read the last two sections over again. Then try out the formulas by hand with some values from Fig. 1-12. Convince yourself that they work.

Now look at lines 1260-1264:

```
1260 FOR HP = 278 TO 66 STEP -2
1261 :    SF = (HP > 255)
1262 :    POKE VIC, HP + (SF * 256)
1263 :    POKE VIC+16, SF * (-1)
1264 NEXT HP
```

This time, our loop will take you from position 278 through to horizontal position 66, again in steps of 2. The sprite will move to the left. Lines 1261-1263 are exactly the same as lines 1255-1257. Poking the registers with a new

```
1000 REM *** DESIGN A SPRITE ***
1105 :    READ SPDTA
1110 :    POKE N, SPDTA
1121 :
1122 DATA   0,  60,   0,   0,  36,   0
1123 DATA   0, 102,  24,   0, 102,  56
1124 DATA   0,  36,  56,   0,  60,  16
1125 DATA   0,  24,  16,   0,  24,  16
1126 DATA  15, 255, 240,   8, 126,   0
1127 DATA   8, 126,   0,   8,  24,   0
1128 DATA  28,  24,   0,  28,  24,   0
1129 DATA  24,  60,   0,   0,  60,   0
1130 DATA   0,  36,   0,   0,  36,   0
1131 DATA   0,  36,   0,   3, 231, 192
1132 DATA   3, 231, 192
1230 POKE VIC+39,1   :REM COLOR IT WHITE
```

Fig. 1-14. Changes and additions that turn A Simple Sprite into the program Design a Sprite.

horizontal position is the same task, whether you are moving to the left or to the right.

Finally, you changed line 1300 to jump back to the beginning of the sideways motion section of the program:

```
1300 IF KP$ = "" THEN 1254
```

## 1.9 A SQUARE'S RETIREMENT

This simple sprite design is getting a bit boring. Let's bring in a more interesting character. Load in the first program from Fig. 1-9 one more time. Then type in the new lines and changes that are listed in Fig. 1-14. When you finish, follow the usual procedure of first saving the program and then running it.

Gone is your little square, and in comes the character that was drawn and coded in Fig. 1-7. Take a good look at the new sprite data loading loop, lines 1100-1120:

```
1100 FOR N = 896 TO 958
1105 :    READ SPDTA
1110 :    POKE N, SPDTA
1120 NEXT N
```

Earlier you were poking each memory location with the same value, 255. That turned all the pixels on. Now, you're using a read statement in line 1105 to get pixel number codes from a series of data statements. Each code is read into the variable SPDTA. Then the value of SPDTA is poked into memory.

Now take a look at the eleven data statements. All 63 of the codes computed in Fig. 1-7 are listed. Notice the order the codes are in: row by row, from the top to the bottom, and from left to right within each row.

Finally, the new version of line 1230 changes the color of the sprite to white. This helps the tiny creature show up. Due to the imperfections of color televisions and the Commodore 64's display circuitry, different colors show up with varying degrees of sharpness against certain backgrounds. You'll have to experiment a bit to get combinations that please you. I usually start out with a black background screen with white sprites and work from there.

## 1.10 SOLVING TWO PROBLEMS

There are two problems with the last program. First, the sprite is too small to show all its detail. Second, it's my design, not yours.

Let's solve the second problem. Back at the close of Section 1.2, you drew several sprite designs and then figured out the 63 number codes for your favorite. Now you'll use that hard-won information.

Load in the last program, Design a Sprite. List lines 1122-1132. Then use the Commodore's useful screen editor to change the pixel codes to the ones you came up with in Section 1.2.

Now for the first problem. The VIC-II chip lets us expand a sprite horizontally and vertically. Details are easier to see in an expanded sprite. Just type in the five lines listed in Fig. 1-15. Remember to save your new program, and then run it.

That's a pretty flashy sprite you designed. Pat yourself on the back. Let's talk about expansion for a moment.

## 1.11 SPRITE EXPANSION AND EXPANSION REGISTERS

A sprite can be made to show up twice as wide on the screen, twice as high, or both. All

```
1000 REM *** A BIGGER SPRITE ***
1233 POKE VIC+23,1   :REM ENLARGE VERT.
1236 POKE VIC+29,1   :REM ENLARGE HORZ.
1353 POKE VIC+29,0
1356 POKE VIC+23,0
```

Fig. 1-15. Changes and additions that turn Design a Sprite into the program A Bigger Sprite.

you need to do is tell VIC-II what you want in the way of expansion.

The 30th VIC-II register, located at VIC + 29, handles horizontal expansion for all eight sprites. By poking a one into this register, sprite #0 shows up twice its normal width. If you poke a zero into this register, sprite #0 shows up with its normal width.

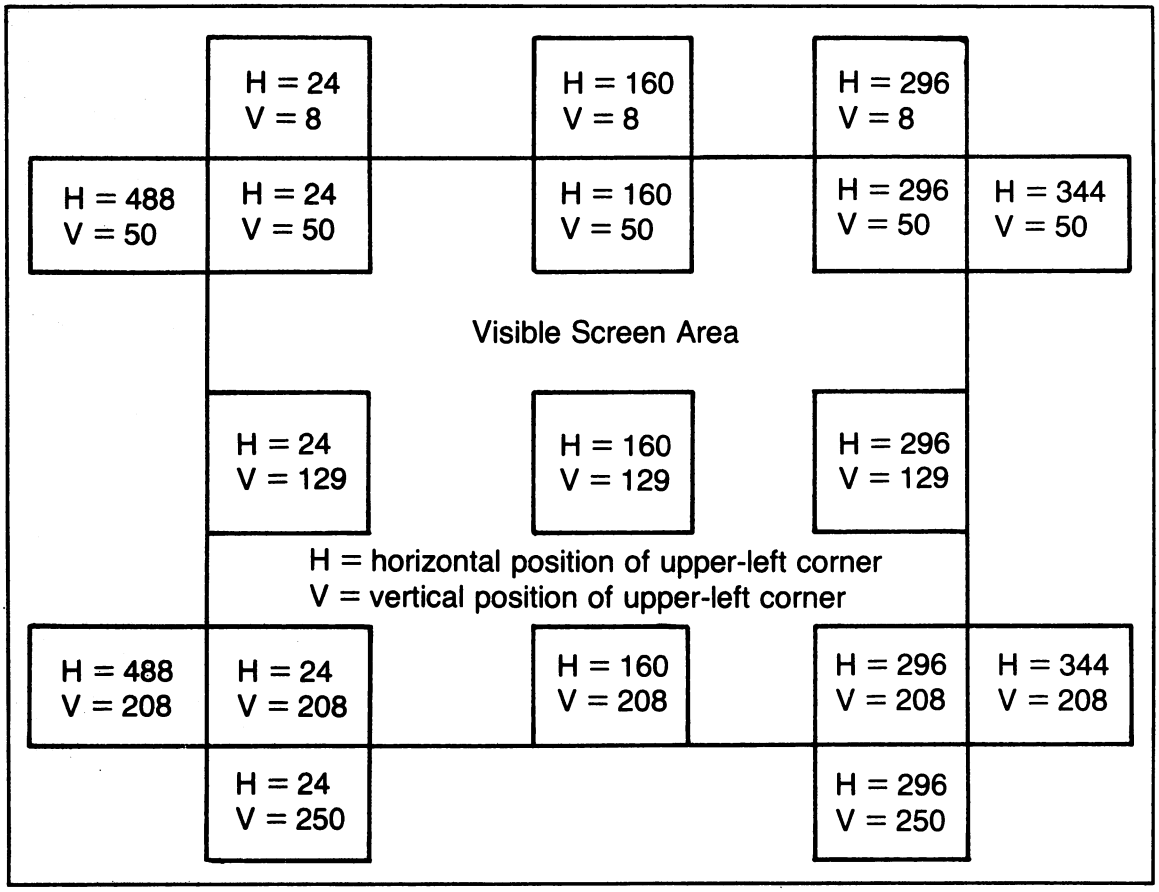



Fig. 1-16. Some important horizontal and vertical position settings for double-sized sprites.

The 24th VIC-II register, located at VIC + 23, handles vertical expansion for the eight sprites. If you poke a one into this location, sprite #0 will double in height. Poking a zero into the register sets sprite #0 to its normal height.

When an expanded sprite is placed on the screen, the numbers in its horizontal and vertical position registers still determine the location of its upper left corner. Figure 1-16 shows how this affects putting the sprite at some of the important screen positions. Compare this figure with Fig. 1-10.

In the last program, A Bigger Sprite, lines 1233 and 1236 poked ones into both expansion registers:

```
1233 POKE VIC+23,1   :REM ENLARGE VERT.
1236 POKE VIC+29,1   :REM ENLARGE HORZ.
```

That made sprite #0 double-sized overall. Then, at the end of the program, lines 1353 and 1356 set sprite #0 back to its usual size by poking zeroes back in:

```
1353 POKE VIC+29,0
1356 POKE VIC+23,0
```

## 1.12 CHAPTER SUMMARY

You've learned quite a bit in this first chapter. By now, you know:

* What pixels and sprites are
* How to design your own sprite and turn the design into 63 coded numbers
* How to load sprite number codes and set the VIC-II registers to display a simple sprite on the screen
* How to set a sprite's position, color, and size
* How to move a sprite sideways or up-and-down

## 1.13 EXERCISES

Now it's time to get a firm hold on your new knowledge. Go through the self-test and write programs for the short exercises. Then write some of your own programs that use the chapter's ideas. Play hard, and you'll become good at it.

### 1.13.1 Self Test

Answers are given in Section 1.13.3. The numbers in parentheses tell you which chapter section to go to for help.

1. (1.1) A sprite is a movable pattern of 504 __________.
2. (1.2) In coding a sprite pattern, you break each of the 21 rows into three groups of __________ pixels.
3. (1.3) To display a sprite, you have to load in 63 number codes, then set up __________ in the __________ chip.
4. (1.5) When you position a sprite, you're actually telling the VIC-II chip where to put the sprite's __________ corner.
5. (1.6) To move a sprite up or down, you just change that sprite's __________ position setting.
6. (1.7) You use ________ registers to set a sprite's horizontal location, because there are __________ possible positions.
7. (1.8) In the following Commodore 64 statement, TV would be set to __________.

```
10 LET TV = (17 < 5)
```

8. (1.9) Rewrite line 1230 of the program Design a Sprite so the sprite shows up yellow. Appendix F may help you. 1230 ______.
9. (1.11) A sprite can be expanded ______ or ______ or in both directions.

### 1.13.2 Programming Exercises

All of these programs can be built upon the program from Fig. 1-9, A Simple Sprite, or if you prefer, you can program them from scratch. Possible solutions are given in Section 1.16. Of course, anything that runs is correct.

1. Have the program move the sprite in a rectangular pattern.
2. Have the sprite change colors every now and then.
3. Cycle the sprite through its four possible sizes: normal, expanded horizontally, expanded vertically, and expanded in both directions.

### 1.13.3 Answers to Self Test

These are just the most obvious (to me) answers. If you've come up with something else, and it makes sense—great!

1. pixels
2. eight
3. registers; VIC-II
4. upper left
5. vertical
6. two; 512
7. zero
8. 1230 POKE VIC+39,7 :REM COLOR IT YELLOW
9. horizontally; vertically (in either order)

### 1.13.4 Possible Solutions to Programming Exercises

My three solutions are all based on the program A Simple Sprite, from Fig. 1-9. Shown here are the lines to change or add to that program in order to solve the exercise.

1. Load in the program A Simple Sprite. Then type in these lines:

```
1000 REM *** RECTANGULAR MOTION ***
1210 POKE VIC,82     :REM HORIZONTAL POS
1220 POKE VIC+1,100 :REM VERTICAL POS
1241 :
1242 :
1243 REM ** MOVE RIGHT, DOWN, LEFT, AND
1244 REM    THEN BACK UP TO STARTING PT
1245 :
1246 REM    (HORIZONTAL MOVES JUMP BY 3
1247 REM     TO MATCH VERTICAL SPEEDS )
1248 :
1249 FOR HP = 84 TO 261 STEP 3 :REM RT.
1250 :    SF = (HP > 255)
1251 :    POKE VIC, HP + (SF * 256)
1252 :    POKE VIC+16, SF * (-1)
1253 NEXT HP
1254 :
1255 FOR VP = 101 TO 179        :REM DOWN
1256 :    POKE VIC+1, VP
1257 NEXT VP
1258 :
1259 FOR HP = 258 TO 87 STEP -3 :REM LF
1260 :    SF = (HP > 255)
1261 :    POKE VIC, HP + (SF * 256)
1262 :    POKE VIC+16, SF * (-1)
1263 NEXT HP
1264 :
1265 FOR VP = 178 TO 100 STEP -1 :REM ↑
1266 :    POKE VIC+1, VP
1267 NEXT VP
1268 :
1269 :
1300 IF KP$ = "" THEN 1249
```

2. Load in the program A Simple Sprite. Then type in these lines:

```
1000 REM *** COLOR CHANGER  ***
1071 REM ** SET STARTING COLOR
1072 :
1073 OC = 13        :REM START WITH GREEN
1074 :
```

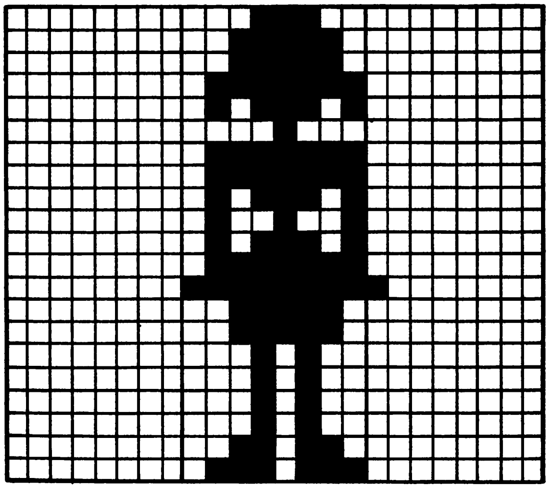
```
1075 :
1230 POKE VIC+39,OC :REM STARTING COLOR
1251 :
1252 REM ** CHANGE COLORS
1253 :
1254 FOR DELAY = 1 TO 500 : NEXT
1255 NC = OC + 1          :REM NEW COLOR
1256 IF NC=16 THEN NC=0   :REM COLORS GO
                               UP TO 15
1257 POKE VIC+39,NC       :REM PUT IT IN
1258 OC = NC              :REM OLD COLOR
1259 :
1300 IF KP$ = "" THEN 1254
```

3. Load in the program A Simple Sprite. Then type in these lines:

```
1000 REM *** GROWTH CYCLE ***
1251 :
1252 REM ** EXPAND HORIZONTALLY
1253 :
1254 FOR DELAY = 1 TO 400 : NEXT
1255 POKE VIC+29,1
1256 :
1257 :
1258 REM ** SHRINK HORIZONTALLY
1259 REM    AND EXPAND VERTICALLY
1260 :
1261 FOR DELAY = 1 TO 400 : NEXT
1262 POKE VIC+29,0
1263 POKE VIC+23,1
1264 :
1265 :
1266 REM ** EXPAND HORIZONTALLY
1267 :
1268 FOR DELAY = 1 TO 400 : NEXT
1269 POKE VIC+29,1
1270 :
1271 :
1272 REM ** SHRINK HORIZONTALLY
1273 REM    AND SHRINK VERTICALLY
1274 :
1275 FOR DELAY = 1 TO 400 : NEXT
1276 POKE VIC+29,0
1277 POKE VIC+23,0
1278 :
1279 :
1280 REM ** WAIT FOR KEYPRESS TO END
1281 :
1300 IF KP$ = "" THEN 1254
```

# Chapter 2

# More Than One Sprite



This chapter shows you how to display more than one sprite on your TV screen. You'll learn how to use the same block of sprite data to make many sprites, and how to alter the way the data is shown. You'll also learn how to put totally different sprites on the screen. Finally, you'll learn one way to get two sprites moving smoothly.

## 2.1 SIMPLE CLONES

The Commodore 64 lets you set up several sprites that use the same block of sprite pixel codes. If you then set the sprites up at different locations and keep them the same size, they look like simple copies of one another, clones.

Figure 2-1 gives a listing of the program Simple Clones. This program will draw four copies of one sprite design.

The sprite design is shown in Fig. 2-2. The program is very similar to the Design A Sprite program from Chapter 1. The main difference is that here you are setting up sprite data pointers, locations, and colors for four sprites. Type the program in. Save it on tape or disk, and then run it. When you're finished, come on back for some explanations.

In Chapter 1 Section 1.3.1 you saw how memory location 2040 is normally used to tell VIC-II where the pixel codes for sprite #0 are located. Memory locations 2041 through 2047 are normally used to tell VIC-II where the pixel data codes for sprites #1 through #7 are located. Figure 2-3 shows which memory location points to data for a particular sprite.

### 2.1.1 Setting Up The Four Sprites

Let's go over the important parts of the Simple Clones program listing. Lines 1000-

```
1000 REM *** SIMPLE CLONES ***
1010 :
1020 :
1030 REM ** SET UP SCREEN FEEDBACK
1040 :
1050 PRINT "{CLR}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}THINKING ";
1060 :
1070 :
1080 REM ** LOAD THE SPRITE DATA
1090 :
1100 FOR N = 896 TO 958
1110 :    READ SPDTA
1120 :    POKE N, SPDTA
1130 NEXT N
1140 :
1150 DATA   6, 102,  96,   6, 102,  96
1160 DATA   6, 102,  96,   7, 255, 224
1170 DATA  15, 255, 240,  28,   0,  56
1180 DATA  56,   0,  28, 113, 195, 142
1190 DATA 225, 195, 135, 193, 195, 131
1200 DATA 192,   0,   3, 192,   0,   3
1210 DATA 192,  60,   3, 192,   0,   3
1220 DATA 204,   0,  51, 198,   0,  99
1230 DATA 195, 255, 195, 192,   0,   3
1240 DATA 224,   0,   7, 127, 255, 254
1250 DATA  63, 255, 252
1260 :
1270 :
1280 REM ** SET UP THE SPRITE CONTROLS
1290 :
1300 PRINT "{CLR}"        :REM CLEAR SCREEN
1310 VIC = 53248        :REM GRAPHICS CHIP
1320 :
1330 POKE 2040,14       :REM #0 DATA POINTR
1340 POKE 2041,14       :REM #1 DATA POINTR
1350 POKE 2042,14       :REM #2 DATA POINTR
1360 POKE 2043,14       :REM #3 DATA POINTR
1370 :
1380 POKE VIC,98        :REM #0 HORZNTL POS
1390 POKE VIC+2,246     :REM #1 HORZNTL POS
1400 POKE VIC+4,98      :REM #2 HORZNTL POS
1410 POKE VIC+6,246     :REM #3 HORZNTL POS
1420 :
1430 POKE VIC+1,95      :REM #0 VERTCAL POS
1440 POKE VIC+3,95      :REM #1 VERTCAL POS
1450 POKE VIC+5,184     :REM #2 VERTCAL POS
```

```
1460 POKE VIC+7,184 :REM #3 VERTCAL POS
1470 :
1480 POKE VIC+39,1  :REM #0 IS WHITE
1490 POKE VIC+40,3  :REM #1 IS CYAN
1500 POKE VIC+41,5  :REM #2 IS GREEN
1510 POKE VIC+42,7  :REM #3 IS YELLOW
1520 :
1530 POKE VIC+21,15 :REM SPRITES 0-3 ON
1540 :
1550 :
1560 REM ** WAIT FOR A KEYPRESS TO END
1570 :
1580 GET KP$
1590 IF KP$ = "" THEN 1580
1600 :
1610 :
1620 REM ** RESET THE SPRITE CONTROLS
1630 :
1640 POKE VIC+21,0
1650 :
1660 END
```

Fig. 2-1. Listing of the program Simple Clones.

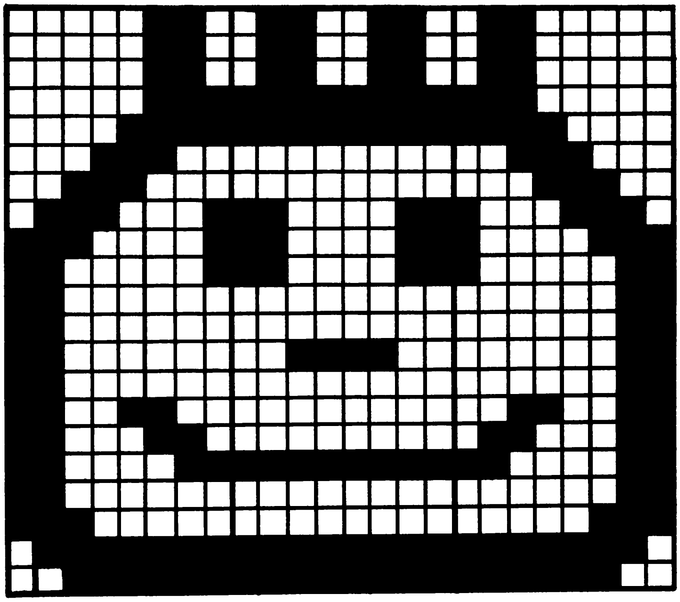


Fig. 2-2. A simple sprite design, ripe for cloning.

| Memory location → | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 | 2046 | 2047 |
|---|---|---|---|---|---|---|---|---|
| Points to pixel data for sprite number → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Fig. 2-3. Memory locations for pointers to sprite data.

1310 should look familiar by now. Feedback is put on the screen; pixel is loaded into memory locations 896-958; the screen is cleared; and the variable VIC is set up with the starting address of the VIC-II chip.

Lines 1330-1360 are the first sign of something new:

```
1330 POKE 2040,14    :REM #0 DATA POINTR
1340 POKE 2041,14    :REM #1 DATA POINTR
1350 POKE 2042,14    :REM #2 DATA POINTR
1360 POKE 2043,14    :REM #3 DATA POINTR
```

You'll be displaying four sprites in this program. Each sprite will be getting its data from the 63 memory locations starting at location (14 × 64), or 896.

Lines 1380-1460 then give each sprite a horizontal and vertical screen position:

```
1380 POKE VIC,98      :REM #0 HORZNTL POS
1390 POKE VIC+2,246   :REM #1 HORZNTL POS
1400 POKE VIC+4,98    :REM #2 HORZNTL POS
1410 POKE VIC+6,246   :REM #3 HORZNTL POS
1430 POKE VIC+1,95    :REM #0 VERTCAL POS
1440 POKE VIC+3,95    :REM #1 VERTCAL POS
1450 POKE VIC+5,184   :REM #2 VERTCAL POS
1460 POKE VIC+7,184   :REM #3 VERTCAL POS
```

Location VIC (53248) is the first horizontal position register for sprite #0, and VIC+1 (53249) is sprite #0's vertical position register. The next fourteen VIC-II registers follow the same pattern for the other seven sprites. VIC+2 (53250) is the first horizontal position register for sprite #1, and VIC+3 (53251) is that sprite's vertical position register. This goes on up through location VIC+15 (53263), which is the vertical position register for sprite #7. Appendix A gives you all the details.

Curious readers are wondering: what about a second horizontal register for each sprite? If you refer to Section 1.7, you will be reminded that each sprite's second horizontal register is actually a miniature register, capable only of holding a one or a zero. Eight of these miniature registers fit into one memory location. That's location VIC+16 (53264). You'll learn more about these miniature registers later in this section.

## 2.1.2 Handing Out Colors and Turning the Sprites On

Lines 1480-1510 give each sprite a color:

```
1480 POKE VIC+39,1    :REM #0 IS WHITE
1490 POKE VIC+40,3    :REM #1 IS CYAN
1500 POKE VIC+41,5    :REM #2 IS GREEN
1510 POKE VIC+42,7    :REM #3 IS YELLOW
```

As you may have guessed, the registers that control the color of each sprite are found in eight consecutive VIC-II locations: VIC+39 (53287) through VIC+ 46 (53294). Again, refer to Appendix A for more detail about the VIC-II registers and to Appendix F for a chart of color codes.

Finally, you come to a moment of truth. Line 1530 turns on four sprites: #0, #1, #2, and #3:

```
1530 POKE VIC+21,15 :REM SPRITES 0-3 ON
```

But what does 15 have to do with 4, or 0, 1, 2, and 3? You'll have to take a short dive into the world of bits and bytes to explain this little mystery. I'll keep it as painless as possible.

### 2.1.3 Bits and Bytes

Remember when you learned to turn pixel designs into number codes? You took the information in groups of eight dots. Why eight, and not nine, or ten, or 24?

The chip that does the Commodore 64's thinking can only handle one number at a time, and that number can't be too large. In fact, it has to be between 0 and 255. Also, the number has to be represented using only the digits 0 and 1.

It turns out that a group of eight 1's and 0's can represent any number between 0 and 255. This brand of number nuttiness is known as base 2, or the *binary number system*. And each binary digit, be it a 1 or a 0, is known as a *bit*.

A group of eight bits is known as a *byte*. Each of the Commodore's many memory locations, including the VIC-II registers, can store one byte, or eight bits. Figures 2-4 and 2-5 give you some bits and bytes to look at.

Many of the VIC-II memory locations can control functions for eight sprites. They do this by assigning one of that location's eight bits to each sprite. Thus, each bit can be thought of as being a miniature register that controls one sprite.

The register at location VIC+21 (53269) is a master control switch for the eight sprites. Any particular sprite may be turned on or off by fiddling with this location. Each bit is a miniature register that turns one sprite on or off.

The eight bits in a byte are numbered 0 through 7. At location VIC+21, bit 0 controls sprite #0, bit 1 controls sprite #1, and so on. To turn on a particular sprite, you just need to put a 1 into its corresponding bit at VIC+21. To turn a sprite off, you put a 0 into its bit at VIC+21.

To get the four sprites numbered 0 through 3 to show up, you've got to poke VIC+21 (the on/off register) with a number that will have 1's in bits 0, 1, 2, and 3, and 0s in the other four bit positions. Sounds tough. Actually, you can use the same chart you used to code a group of eight pixels.

Figure 2-6 shows a byte with its 8 bits numbered 0-7. Each bit is also given a bit value. You first put 1's in the bit positions of

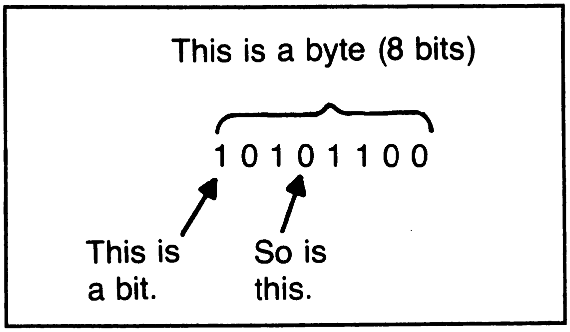



Fig. 2-4. One byte is made up of 8 bits.

| This normal number → | 255 | 240 | 128 | 127 | 60 | 15 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Turns into this binary byte → | 11111111 | 11110000 | 10000000 | 01111111 | 00111100 | 00001111 | 00000001 | 00000000 |

Fig. 2-5. Binary bytes, composed of eight bits, can represent normal (base 10) values between 0 and 255.

the sprites you want on, and 0's where you want sprites off. Then, by adding the values of the bits that contain 1's, you get the number you need to poke into memory to obtain the correct pattern of 1's and 0's.

Figure 2-7 shows some examples of this. Let's look at the one that applies to the Simple Clones program. You want to turn on sprites 0-3, so you need to store 1's in bits 0-3. You add the bit values for those bits—8 + 4 + 2 + 1—and get 15. Your brain may ache a bit, but the mystery of 15 is solved.

### 2.1.4 Wrap It Up

The rest of Simple Clones should be familiar. Lines 1580-1590 wait for a keypress. When one is detected, line 1640 resets the sprite controls. Here a little secret pops out: not every sprite control needs to be reset.

Which controls do you need to reset? Well, the on/off register, at location VIC+21, should be set to 0 so all of the sprites disappear. If you've expanded any sprite horizontally or vertically, the sprite expansion registers at VIC+23 and VIC+29 should be put back to 0. That way, you won't be surprised by sprites stretched in unexpected ways.

## 2.2 COMPLEX CLONES

Even though they use the same pixel data, the four sprites in the last program aren't exactly alike. Each appears on the screen in a different color. You can make them look even less alike by expanding them in different ways.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit value → | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Any 8-bit byte → | 1 or 0 | 1 or 0 | 1 or 0 | 1 or 0 | 1 or 0 | 1 or 0 | 1 or 0 | 1 or 0 |
| Bit number → | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Fig. 2-6. A byte with its 8 bits numbered 0 - 7. Each bit is shown with its place value.

| Sprites on | Sprites off | Register byte (each bit controls sprite with the same #) | Number to poke |
|---|---|---|---|
| – | 0–7 | Bit value: 128 \| 64 \| 32 \| 16 \| 8 \| 4 \| 2 \| 1<br>0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 0<br>Bit number: 7 \| 6 \| 5 \| 4 \| 3 \| 2 \| 1 \| 0 | 0 |
| 0 | 1–7 | Bv: 128 \| 64 \| 32 \| 16 \| 8 \| 4 \| 2 \| 1<br>0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 1<br>Bn: 7 \| 6 \| 5 \| 4 \| 3 \| 2 \| 1 \| 0 | 1=<br>1 |
| 0,1 | 2–7 | Bv: 128 \| 64 \| 32 \| 16 \| 8 \| 4 \| 2 \| 1<br>0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 1 \| 1<br>Bn: 7 \| 6 \| 5 \| 4 \| 3 \| 2 \| 1 \| 0 | 1+2=<br>3 |
| 0–3 | 4–7 | Bv: 128 \| 64 \| 32 \| 16 \| 8 \| 4 \| 2 \| 1<br>0 \| 0 \| 0 \| 0 \| 1 \| 1 \| 1 \| 1<br>Bn: 7 \| 6 \| 5 \| 4 \| 3 \| 2 \| 1 \| 0 | 1+2+4+8=<br>15 |
| 0,2, 4,6 | 1,3, 5,7 | Bv: 128 \| 64 \| 32 \| 16 \| 8 \| 4 \| 2 \| 1<br>0 \| 1 \| 0 \| 1 \| 0 \| 1 \| 0 \| 1<br>Bn: 7 \| 6 \| 5 \| 4 \| 3 \| 2 \| 1 \| 0 | 1+4+16+<br>64=<br>85 |
| 0-7 | – | Bv: 128 \| 64 \| 32 \| 16 \| 8 \| 4 \| 2 \| 1<br>1 \| 1 \| 1 \| 1 \| 1 \| 1 \| 1 \| 1<br>Bn: 7 \| 6 \| 5 \| 4 \| 3 \| 2 \| 1 \| 0 | 1+2+4+8+<br>16+32+64<br>+128=<br>255 |

Fig. 2-7. In these examples, a byte-sized register uses its eight bits to turn sprites on or off. In each case the individual bits are set by poking the values in the right-hand column.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit value → | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Bit → | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| Bit number → | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

8 + 4 = 12

Fig. 2-8. Poking the value 12 into the horizontal expansion register set bits 2 and 3 to 1, causing sprites 2 and 3 to expand horizontally.

As you learned in Section 1.11, location VIC+29 handles horizontal expansion for all eight sprites. Each bit in the byte stored there controls horizontal expansion for one sprite. If you want sprites #2 and #3 to be expanded horizontally, bits 2 and 3 must be set to 1. Using the bit values shown in Fig. 2-6, you can find the number to poke into the register: 8 + 4, or 12. See Fig. 2-8.

The register at VIC+23 handles vertical expansion for all eight sprites in a similar way. If you want sprite #1 and sprite #3 to expanded vertically, for example, you need to set bits 1 and 3 of that register to 1. Adding the bit values, you find the number to poke into VIC+23: 8 + 2, or 10. See Fig. 2-9.

Let's use this new know-how to change the Simple Clones program. Load it into the computer, and then type in the lines listed in Fig. 2-10 to turn it into the program Complex Clones. Save the new program on tape or disk, and then run it.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit value → | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Bit → | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| Bit number → | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

8 + 2 = 10

Fig. 2-9. Poking the value 10 into the vertical expansion register sets bits 1 and 3 to 1, causing sprites 1 and 3 to expand vertically.

```
1000 REM *** COMPLEX CLONES ***
1400 POKE VIC+4,86  :REM #2 HORZNTL POS
1410 POKE VIC+6,234 :REM #3 HORZNTL POS
1440 POKE VIC+3,85  :REM #1 VERTCAL POS
1460 POKE VIC+7,174 :REM #3 VERTCAL POS
1522 POKE VIC+23,10 :REM #1 & #3 TALL
1524 POKE VIC+29,12 :REM #2 & #3 WIDE
1526 :
1642 POKE VIC+23,0
1644 POKE VIC+29,0
```

Fig. 2-10. Changes and additions that turn the program Simple Clones into the program Complex Clones.

Voila! You now have four sprites on the screen, all based on the same block of pixel data, and each one looks very different from the others. It was done quite simply. The new versions of lines 1400, 1410, 1440, and 1460 move the sprites around a little bit. Then lines 1522 and 1524 institute the sprite expansions used as examples up above:

```
1522 POKE VIC+23,10 :REM #1 & #3 TALL
1524 POKE VIC+29,12 :REM #2 & #3 WIDE
```

Sprite #0 stays normal-sized. Sprite #1 gets taller. Sprite #2 gets wider. Sprite #3 is expanded in both directions. When a keypress signals the end of the program. Lines 1642 and 1644 set the expansion registers back to 0.

## 2.3 STORING MORE THAN ONE BLOCK OF SPRITE PIXEL DATA

In many cases, you'll want to have sprites that look very different from one another. In order to do this, you need to load a block of pixel data for each different sprite image. Where should you put the 63 numbers for each one?

If you're using three or fewer different sprite images, you can put the data in these three areas: memory locations 832-894, 896-958, and 960-1022. These areas of memory are used with the Commodore's tape recorder, so they're pretty safe when you're inside a program. The sprite data pointers at 2040-2047 must contain the starting address of the pixel data block divided by 64; so, for these three areas, the pointers would contain 13, 14, or 15 respectively.

If you're using more than three blocks of pixel data, use memory locations starting at 12288. Figure 2-11 gives the locations, along with the pointer number used for each area. More exotic locations are available to advanced programmers who are willing to play around with the Commodore 64's memory map, but that's information for another book.

## 2.4 GETTING TWO VERY DIFFERENT SPRITES

Imagine a program that will put two different sprite images on the screen. How will it differ from a program like Design a Sprite, from Chapter 1?

First, it must load in two blocks of pixel data. Then, it has to set the pointers at 2040 and 2041 to point to the two areas filled with pixel data. Third, it must set up the VIC-II registers to position, color, and size each sprite. Finally, it has to turn both sprites on.

Figure 2-12 shows two new sprite designs. Figure 2-13 is a listing of the program Spritely Couple, which puts them on the screen—such a sweet young couple. Type the program in, then save and run it. Fool around with it, changing parts of the images and register settings; then come on back for a brief explanation of its workings.

## 2.5 ALL ABOUT YOUR YOUNG COUPLE

Nothing in the listing of Spritely Couple should surprise you. Let's go over some of the details. Line 1050 cleans the screen and sets up for feedback. Then two loops load in the two blocks of sprite pixel data. The first set of 63 numbers is put into locations 896-958. Line 1140 signals that the first block is set by putting a period next to the word *THINKING*. The second set of 63 numbers is put into locations 960-1022. Then line 1200 signals the end of that process with another period. The pixel data was figured using a copy of the coding form from Fig. 1-6.

The program then sets the data pointers and VIC-II registers. I decided to make both sprites double-sized since they were so detailed. It's tough to see the detail at normal size. Line 1660 turns on sprites #0 and #1 by putting 1's into bits 0 and 1 of VIC+21. Go back to Section 2.1.3. if you're not certain why 3 was the value poked in.

Lines 1700-1710 wait for our usual keypress to close up shop. Then lines 1770-1790 reset the on/off and expansion registers—very straightforward stuff.

## 2.6 MOVING MORE THAN ONE SPRITE AT A TIME

There are many different techniques you can use to get several sprites in motion. Some are easy to program; some are difficult. Some use lots of the machine's memory; some use very little. Some can only provide simple paths, while others can provide very complex ones. Some give motion that is fast and smooth, while others give slow and jerky results. Some are very straightforward; others are tricky and difficult to understand. There is only room for one example in this chapter; so I've chosen one that's not too tough and yet gives a nice result.

You're going to take the two sprites from the program Spritely Couple and let them

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 63-byte area of memory → | 12288 — 12350 | 12352 — 12414 | 12416 — 12478 | 12480 — 12542 | 12544 — 12606 | 12608 — 12670 | 12672 — 12734 | 12736 — 12798 |
| Set pointer → to | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 |

Fig. 2-11. Areas to store sprite data, along with the appropriate pointer values.

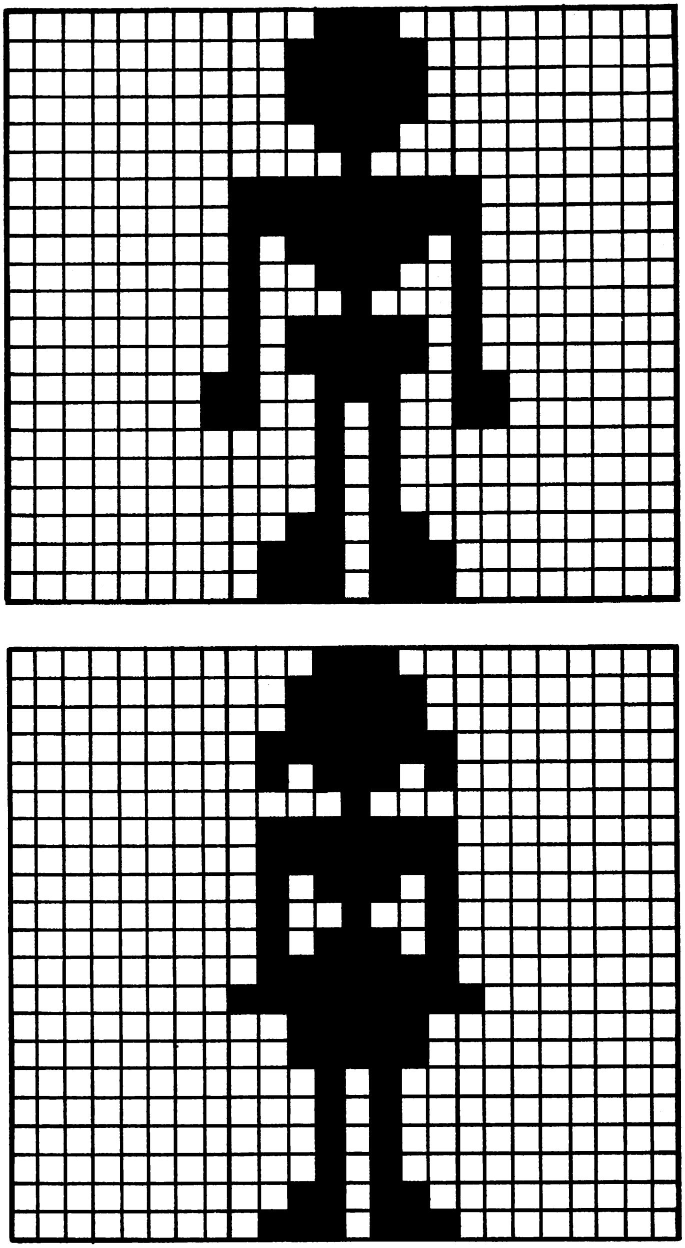


Fig. 2-12. Two new sprite designs are used in the program Spritely Couple.

```
1000 REM *** SPRITELY COUPLE ***
1010 :
1020 :
1030 REM ** SET UP SCREEN FEEDBACK
1040 :
1050 PRINT "♥QQQQQQQQQQTHINKING ";
1060 :
1070 :
1080 REM ** LOAD THE SPRITE DATA
1090 :
1100 FOR N = 896 TO 958    :REM 1ST ONE
1110 :    READ SPDTA
1120 :    POKE N, SPDTA
1130 NEXT N
1140 PRINT ". ";           :REM FEEDBACK
1150 :
1160 FOR N = 960 TO 1022   :REM 2ND ONE
1170 :    READ SPDTA
1180 :    POKE N, SPDTA
1190 NEXT N
1200 PRINT ". ";           :REM FEEDBACK
1210 :
1220 DATA   0,  28,   0,   0,  62,   0
1230 DATA   0,  62,   0,   0,  62,   0
1240 DATA   0,  28,   0,   0,   8,   0
1250 DATA   0, 255, 128,   0, 255, 128
1260 DATA   0, 190, 128,   0, 156, 128
1270 DATA   0, 136, 128,   0, 190, 128
1280 DATA   0, 190, 128,   1, 156, 192
1290 DATA   1, 148, 192,   0,  20,   0
1300 DATA   0,  20,   0,   0,  20,   0
1310 DATA   0,  54,   0,   0, 119,   0
1320 DATA   0, 119,   0
1330 :
1340 DATA   0,  28,   0,   0,  62,   0
1350 DATA   0,  62,   0,   0, 127,   0
1360 DATA   0,  93,   0,   0,   8,   0
1370 DATA   0, 127,   0,   0, 127,   0
1380 DATA   0,  93,   0,   0,  73,   0
1390 DATA   0,  93,   0,   0, 127,   0
1400 DATA   0, 255, 128,   0,  62,   0
1410 DATA   0,  62,   0,   0,  20,   0
1420 DATA   0,  20,   0,   0,  20,   0
1430 DATA   0,  20,   0,   0,  54,   0
1440 DATA   0, 119,   0
1450 :
1460 :
```

```
1470 REM ** SET UP THE SPRITE CONTROLS
1480 :
1490 PRINT "♥";          :REM CLEAR SCREEN
1500 VIC = 53248         :REM GRAPHICS CHIP
1510 :
1520 POKE 2040,14        :REM #0 DATA POINTR
1530 POKE 2041,15        :REM #1 DATA POINTR
1540 :
1550 POKE VIC,124        :REM #0 HORIZONTAL
1560 POKE VIC+2,173      :REM #1 HORIZONTAL
1570 POKE VIC+1,150      :REM #0 VERTICAL
1580 POKE VIC+3,150      :REM #1 VERTICAL
1590 :
1600 POKE VIC+39,3       :REM #0 IS CYAN
1610 POKE VIC+40,7       :REM #1 IS YELLOW
1620 :
1630 POKE VIC+23,3       :REM BOTH SPRITES
1640 POKE VIC+29,3       :REM DOUBLE-SIZED
1650 :
1660 POKE VIC+21,3       :REM TURN BOTH ON
1670 :
1680 :
1690 REM ** WAIT FOR KEYPRESS TO END
1700 :
1710 GET KP$
1720 IF KP$ = "" THEN 1710
1730 :
1740 :
1750 REM ** RESET THE SPRITE CONTROLS
1760 :
1770 POKE VIC+21,0       :REM SPRITES OFF
1780 POKE VIC+29,0       :REM AND SIZES
1790 POKE VIC+23,0       :REM BACK TO NORMAL
1800 :
1810 END
```

Fig. 2-13. Listing of the program Spritely Couple.

chase one another around the screen. You'll program this motion by making changes and additions to Spritely Couple. So load it into your machine, and type in the lines listed in Fig. 2-14. Save and run the resulting program.

### 2.6.1 Thinking about the Path

In this program, the two sprites race in a square path that's centered on the screen. The path is 100 pixels wide and 100 pixels high. So the corners of the path will be 50 pixels away from the center of the screen, both horizontally and vertically. To center a double-sized sprite on the screen, its horizontal position should be 160 and its vertical position should be 129 (see Fig. 1-16). To find the corner positions, just

```
1000 REM *** SPRITELY CHASE ***
1550 POKE VIC,110    :REM #0 HORIZONTAL
1560 POKE VIC+2,210  :REM #1 HORIZONTAL
1570 POKE VIC+1,79   :REM #0 VERTICAL
1580 POKE VIC+3,179  :REM #1 VERTICAL
1662 :
1664 :
1666 REM ** INITIALIZE SPRITE MOTION
1667 :
1668 D0 = 1 : D1 = -1
1670 :
1672 :
1674 REM ** MOVE VERTICALLY
1676 :
1678 FOR MOVE = 1 TO 100
1680 :    POKE VIC+1, PEEK(VIC+1) + D0
1682 :    POKE VIC+3, PEEK(VIC+3) + D1
1684 :    GET KP$
1686 :    IF KP$ = "" THEN 1690
1688 :       MOVE = 100 : KEYPRESS = -1
1690 NEXT MOVE
1692 :
1694 :
1696 REM ** IF KEY PRESSED, FINISH UP
1698 :
1700 IF KEYPRESS THEN 1750
1702 :
1704 :
1706 REM ** MOVE HORIZONTALLY
1708 :
1710 FOR MOVE = 1 TO 100
1712 :    POKE VIC, PEEK(VIC) + D0
1714 :    POKE VIC+2, PEEK(VIC+2) + D1
1716 :    GET KP$
1718 :    IF KP$ = "" THEN 1722
1720 :       MOVE = 100 : KEYPRESS = -1
1722 NEXT MOVE
1724 :
1726 :
1728 REM ** IF KEY PRESSED, FINISH UP
1730 :
1732 IF KEYPRESS THEN 1750
1734 :
1736 :
1738 REM ** REVERSE MOTION AND REPEAT
1740 :
1742 D0 = -D0 : D1 = -D1
1744 GOTO 1678
```

```
1746 :
1748 :
```

Fig. 2-14. Changes and additions that turn the program Spritely Oouple into the program Spritely Chase.

add and subtract 50 from the centering position. Figure 2-15 shows the resulting corner positions.

Start sprite #0 in the upper left corner, and sprite #1 in the lower right corner. Sprite #0 has to move down, right, up, and then left. Sprite #1 has to move up, left, down, and then right. Take a look at Fig. 2-16. It shows four views of the two sprites as they move about the path. View 1 shows the starting positions. The arrows indicate the direction each sprite is moving in. Notice that when one sprite moves vertically, the other also moves vertically, but in the opposite direction. When one moves horizontally, the other also moves horizontally, but again in the opposite direction. This symmetry of motion makes your programming job a lot easier.

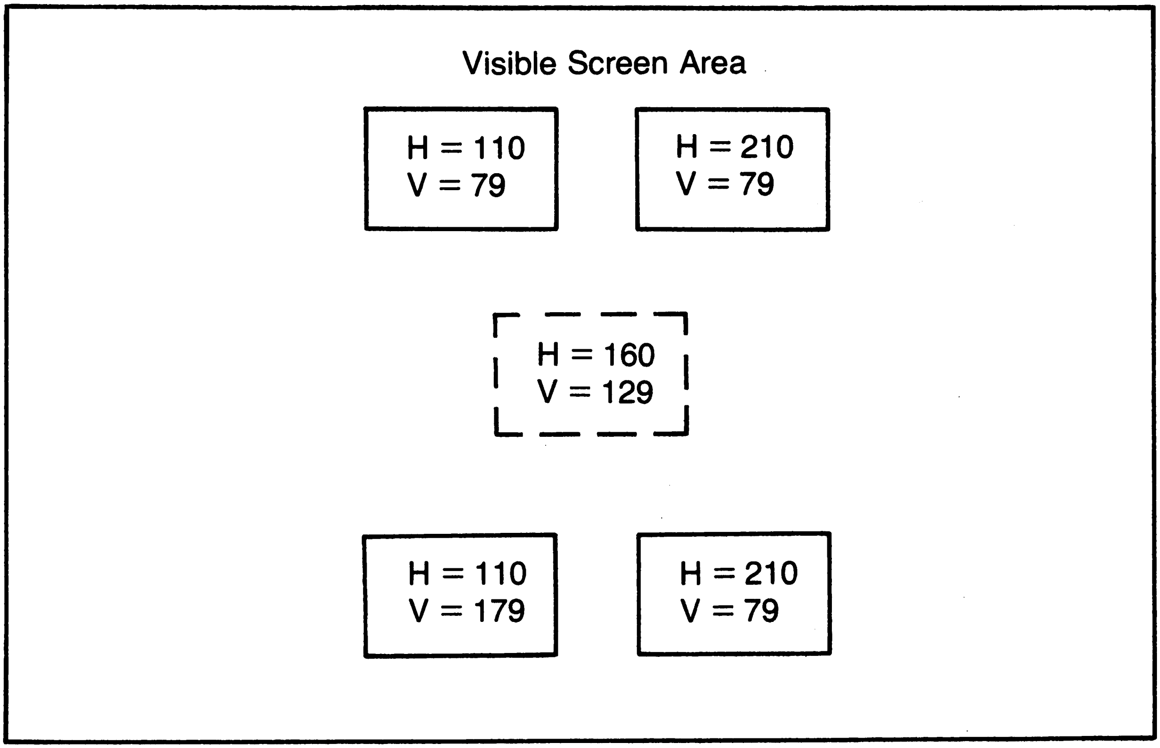



Fig. 2-15. Corner positions that are reached by a sprite following a square path. Each side of the square path is 100 pixels long; the square is centered on the screen.

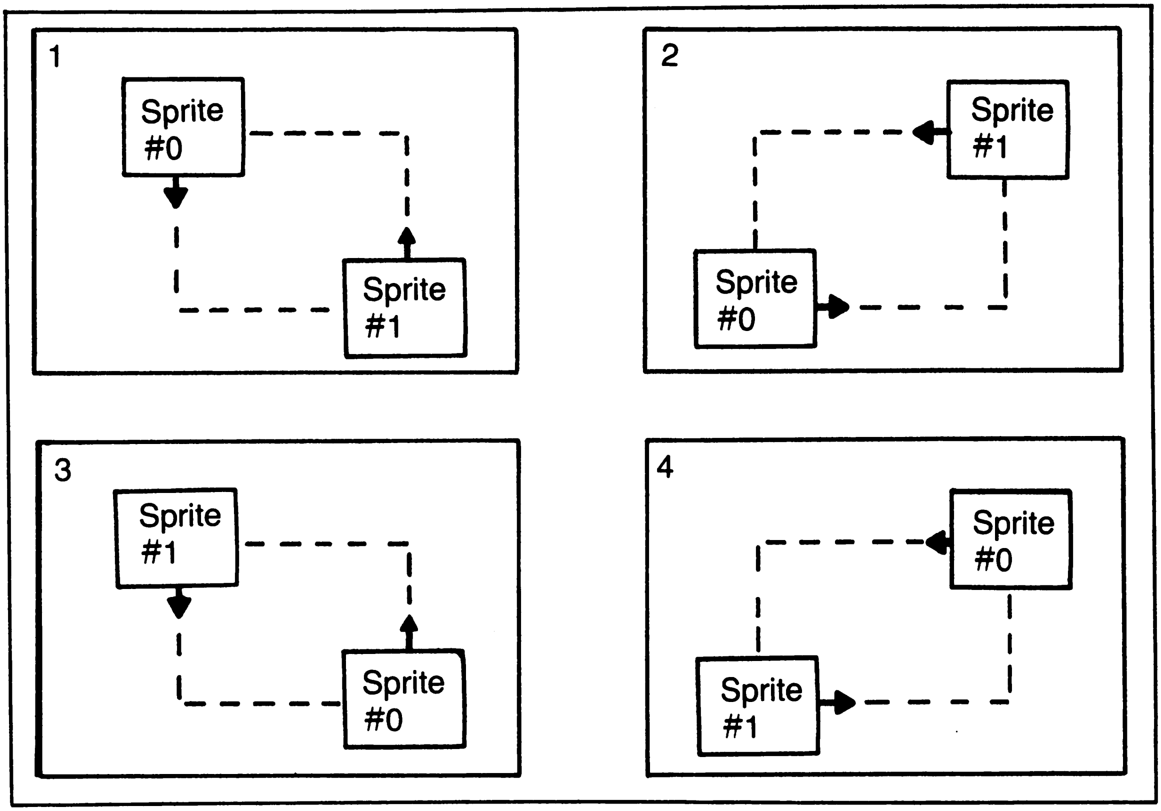



Fig. 2-16. Four pictures of two sprites as they move around the square path.

## 2.6.2 Establishing Sprite Positions and Motions

Lines 1550-1580 set the initial sprite positions:

```
1550 POKE VIC,110   :REM #0 HORIZONTAL
1560 POKE VIC+2,210 :REM #1 HORIZONTAL
1570 POKE VIC+1,79  :REM #0 VERTICAL
1580 POKE VIC+3,179 :REM #1 VERTICAL
```

As mentioned above, sprite #0 starts in the upper left corner of the path, and sprite #1 starts in the lower right corner.

The program uses two variables to produce the sprite's motions. D0 does the chore for sprite #0, and D1 does it for sprite #1. Line 1668 gives these two variables their starting values:

```
1668 D0 = 1 : D1 = -1
```

You'll be adding the values of these motion variables to the sprites' position registers. Let's think this out a bit.

If you add positive numbers to a sprite's vertical position, the number gets larger, and the sprite will move down the screen. Adding negative numbers will cause the vertical positions to have a smaller value, and the sprite will move up the screen. Horizontal positioning works in a similar way. Adding positive numbers to the horizontal position will move a sprite to the right, and adding negative numbers will move it to the left.

Lines 1678-1690 take care of all vertical path motions for both sprites:

```
1678 FOR MOVE = 1 TO 100
1680 :    POKE VIC+1, PEEK(VIC+1) + D0
1682 :    POKE VIC+3, PEEK(VIC+3) + D1
1684 :    GET KP$
1686 :    IF KP$ = "" THEN 1690
1688 :       MOVE = 100 : KEYPRESS = -1
1690 NEXT MOVE
```

Lines 1678 and 1690 set up a loop that will be carried out 100 times. That's because each side of the path is 100 pixels long, and you'll be moving one pixel each time you pass through the loop. Each time through, line 1680 will add the value of sprite #0's motion variable to that sprite's vertical position register. Similarly, line 1682 adds the value of sprite #1's motion variable to its vertical position.

Lines 1686-1688 represent an improvement over our previous moving sprite programs. Now you can check for a keypress after each sprite move, rather than waiting for a whole cycle to end. Line 1686 scans the keyboard. If a key hasn't been pressed, line 1688 is skipped, and the loop merrily goes about its business. If a key has been pressed, two things occur: the value of the loop-counting variable MOVE is jumped up to 100, and the variable KEYPRESS is set to −1. Setting MOVE to 100 will force a quick loop exit when line 1690 is hit. This is a clean way to leave a loop in a hurry. KEYPRESS is set to −1 because −1 represents TRUE. Refer back to Section 1.8 if this seems odd.

Line 1700 will either send us on to the horizontal motion loop or the end of the program based on the value of KEYPRESS:

```
1700 IF KEYPRESS THEN 1750
```

If KEYPRESS contains a 0, representing false, no key has been pressed, and the program goes on to the horizontal loop. But if KEYPRESS contains a −1, then a key has been pressed. KEYPRESS will be interpreted as true, and the program will go to the clean-up-shop-and-end segment that starts at line 1750.

By the way, these true/false tests are known as Boolean tests, and you can call KEYPRESS a Boolean variable. It's always a bit of fun to know some jargon.

Lines 1710-1722 form a loop that takes care of horizontal path motion:

```
1710 FOR MOVE = 1 TO 100
1712 :    POKE VIC, PEEK(VIC) + D0
1714 :    POKE VIC+2, PEEK(VIC+2) + D1
1716 :    GET KP$
1718 :    IF KP$ = "" THEN 1722
1720 :       MOVE = 100 : KEYPRESS = -1
1722 NEXT MOVE
```

This loop is almost exactly the same as the one for vertical motion. The only difference is that now the program will add the motion values to the horizontal position registers.

Line 1732 again tests to see if a key was pressed during the preceding loop:

```
1732 IF KEYPRESS THEN 1750
```

If a key has been pressed, the program will jump to line 1750 and end itself. If one hasn't been pressed, it's time to change directions.

### 2.6.3 A Cheap Path Trick: Changing Directions

Consider sprite #0 in this program. To complete one trip around the square path, it must go down, then right, then up, and then left. Or think of it another way: vertical motion, horizontal motion, vertical motion, and horizontal motion. You've covered two loops that took care of the first vertical and horizontal motions. Now you need another pair of these loops—or do you?

You don't. You can just switch the direction of sprite #0's motion, and then go back to the same two horizontal and vertical loops. The original value of the motion variable D0 was 1. If you multiply it by −1, it becomes −1. Now the vertical loop will send sprite #0 up, and the horizontal loop will send it to the left. Similarly, you can reverse the direction of sprite #1's motion. Its original motion value was −1; multiplying that by −1 gives a motion value of 1. It will now go down in the vertical loop, and to the right in the horizontal loop, which is just what you want it to do. Once both motions are reversed, you must leap back up to line 1678 and go through the motion loops again.

Lines 1742-1744 are the ones that pull off this reversal:

```
1742 D0 = -D0 : D1 = -D1
1744 GOTO 1678
```

One last bit of thinking: the next time the program gets to line 1742, the motions will again be reversed. This will set them back to their original values, which is perfect, because at that point each sprite will be back in its original position: #0 in the upper left corner of the path and #1 in the lower right corner of the path.

Okay, now it's your turn. Spend some time playing around with Spritely Chase. Can you get a triangular path? Or move four sprites around the square? Or have the sprites spiral in to the center of the screen, and then spiral out again? Remember to think first, and write program lines afterward.

## 2.7 CHAPTER SUMMARY

In this chapter you've seen a few techniques for dealing with more than one sprite at a time. You've learned:

* How to put several sprites on the screen, using the same 63 bytes of pixel data for each one
* About bits and bytes, and how they're used in some of the VIC-II registers to control individual sprites
* About storing more than one block of sprite data, and how to set the sprite pointers at 2040-2047
* One of the ways to get more than one sprite moving in an interesting pattern
* About using Boolean variables to quickly leave a program from deep inside a loop

## 2.8 EXERCISES

In the next chapter you'll discover more sprite magic. In the meantime, here are some exercises to sharpen your skills.

### 2.8.1 Self Test

Answers are given in Section 2.8.3. The numbers in parentheses tell you which section of the chapter to go to for help.

1. (2.1) Memory location 2045 usually serves as a sprite data pointer for sprite # ________.
2. (2.1.3) A group of eight bits is known as a ________.
3. (2.1.3) A byte can represent decimal numbers between 0 and ________.
4. (2.1.3) If you want sprites #2, #4, and #7 to appear, you just poke the decimal number ________ into location VIC+21.
5. (2.2) If you want sprites #0, #3, and #4 to be expanded vertically, you poke the number ________ into location VIC+23.
6. (2.3) If you're using eight blocks of sprite data, a good area of memory to store them starts at location ________.
7. (2.6.1) To set a double-sized sprite halfway down the screen, its vertical position register should be set to ________.
8. (2.6.2) As a sprite's horizontal position gets larger, it moves towards the ________ side of the screen.
9. (2.6.2) Variables that take on values representing true or false are known as ________ variables.
10. (2.6.3) To get a variable's value to switch back and forth from −1 to 1, we just repeatedly multiply the variable by ________.

### 2.8.2 Programming Exercises

1. Change the program Simple Clones so that four more clones appear, one in each corner of the screen.
2. Change the program Spritely Chase so that the sweet young couple moves in a clockwise direction.
3. Change the program Spritely Chase so that two females chase two males around the square.

### 2.8.3 Answers to the Self Test

Again, these are just my favorite answers. Other answers that you can justify to yourself are fine.

1. 5
2. byte
3. 255
4. 148
5. 25
6. 12288
7. 129
8. right
9. Boolean
10. −1

### 2.8.4 Possible Solutions to Programming Exercises

These solutions are based on adding or changing lines in the programs mentioned in the exercises. Remember, any solution that completes the task is fine.

1. Load in the program Simple Clones. Then type in these lines:

```
1000 REM *** EIGHT CLONES ***
1362 POKE 2044,14       :REM #4 DATA POINTR
1364 POKE 2045,14       :REM #5 DATA POINTR
1366 POKE 2046,14       :REM #6 DATA POINTR
1368 POKE 2047,14       :REM #7 DATA POINTR
1412 POKE VIC+8,24      :REM #4 HORZNTL POS
1414 POKE VIC+10,64     :REM #5 HORZNTL POS
```

```
1416 POKE VIC+12,24 :REM #6 HORZNTL POS
1418 POKE VIC+14,64 :REM #7 HORZNTL POS
1419 POKE VIC+16,160:REM 5&7 USE 2ND HR
1462 POKE VIC+9,50  :REM #4 VERTCAL POS
1464 POKE VIC+11,50 :REM #5 VERTCAL POS
1466 POKE VIC+13,229:REM #6 VERTCAL POS
1468 POKE VIC+15,229:REM #7 VERTCAL POS
1512 POKE VIC+43,7  :REM #4 IS YELLOW
1514 POKE VIC+44,5  :REM #5 IS GREEN
1516 POKE VIC+45,3  :REM #6 IS CYAN
1518 POKE VIC+46,1  :REM #7 IS WHITE
1530 POKE VIC+21,255:REM SPRITES 0-7 ON
```

2. Load in the program Spritely Chase. Then type in these lines:

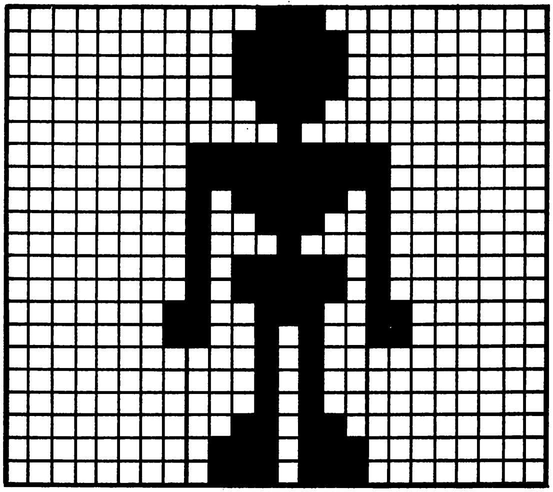
```
1000 REM *** CLOCKWISE CHASE ***
1674 REM ** MOVE HORIZONTALLY
1680 :    POKE VIC, PEEK(VIC) + D0
1682 :    POKE VIC+2, PEEK(VIC+2) + D1
1706 REM ** MOVE VERTICALLY
1712 :    POKE VIC+1, PEEK(VIC+1) + D0
1714 :    POKE VIC+3, PEEK(VIC+3) + D1
```

3. Load in the program Spritely Chase. Then type in these lines:

```
1000 REM *** COUPLES CHASE ***
1530 POKE 2041,14   :REM #1 DATA POINTR
1533 POKE 2042,15   :REM #2 DATA POINTR
1536 POKE 2043,15   :REM #3 DATA POINTR
1563 POKE VIC+4,110 :REM #2 HORIZONTAL
1566 POKE VIC+6,210 :REM #3 HORIZONTAL
1568 :
1583 POKE VIC+5,179 :REM #2 VERTICAL
1586 POKE VIC+7,79  :REM #3 VERTICAL
1613 POKE VIC+41,1  :REM #2 IS WHITE
1616 POKE VIC+42,5  :REM #3 IS GREEN
1630 POKE VIC+23,15 :REM ALL 4 SPRITES
1640 POKE VIC+29,15 :REM DOUBLE-SIZED
1660 POKE VIC+21,15 :REM TURN ALL 4 ON
1674 REM ** MOVE ONE SIDE OF PATH
1681 :    POKE VIC+4, PEEK(VIC+4) + D0
1683 :    POKE VIC+6, PEEK(VIC+6) + D1
1706 REM ** MOVE ANOTHER SIDE OF PATH
1713 :    POKE VIC+5, PEEK(VIC+5) + D1
1715 :    POKE VIC+7, PEEK(VIC+7) + D0
```

# Chapter 3

# Some More Sprite Tricks



Would you like to display some sprites that have more than one color? This chapter will show you how. You'll also learn about sliding sprites over and under one another. Finally, you'll use a set of sprite images to create some funny animation. Along the way, you'll pick up some more experience with bits, bytes, and spritely motion.

## 3.1 TRADING DETAIL FOR COLOR: SPRITE MULTICOLOR MODE

First, a little review. A normal sprite design is defined by storing 63 bytes of pixel information in the computer's memory. Each byte contains eight bits, and each bit turns one pixel on or off. Since 63 times 8 is 504, you're able to define sprites that contain 504 pixels.

If a pixel's bit is set to 1, that pixel will show up in the color you set in the sprite's color register. If a pixel's bit is set to 0, that pixel will show up as the color of the screen; in other words, it won't really show up at all.

Since normally there's just one bit to play with, a pixel has two choices: show up, or be invisible. However, Commodore has given us an alternative: the multicolor sprite mode. In this mode, you can use two bits to pick a color. The two bits are called a bit pair.

Two bits can hold four possible bit patterns, as shown in Fig. 3-1. And that means you can pick any one of four colors for the two dots set by a bit pair. Of course, both dots will have the same color. It's best to think of the two dots as one double-wide pixel. This brings up a tradeoff you must make: in multicolor sprite mode, each byte sets the colors for four double-wide pixels. So each row of the sprite image will have 12 double-wide pixels instead of 24 normally-sized pixels. The sprite will have more color, but less horizontal detail.

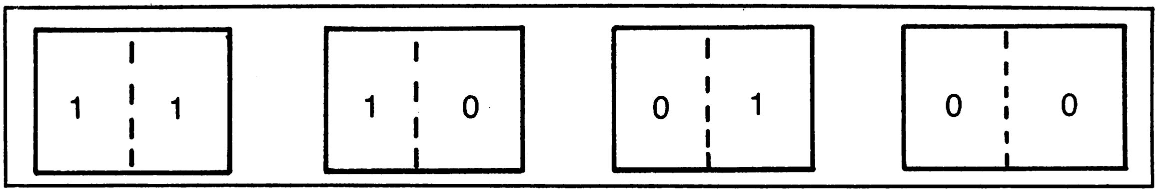



Fig. 3-1. Two bits can hold four possible bit patterns.

Let's go over that one more time. Since two bits will be needed to choose a color, each byte will only be able to control four double-wide pixels. See Fig. 3-2. With three bytes per row of the sprite design, that means the sprite will be 12 double-wide pixels across. It will show up the same size as a normal sprite, but with less horizontal detail. Since you still use 63 bytes to define the sprite design, it'll be composed of 252 double-wide pixels (63 × 4).

## 3.2 MORE ABOUT THE MULTICOLOR MODE

What colors will show up when you display a multicolor sprite? If the bit pair is 00, the double wide-pixel will be given the screen's color. By the way, the screen color is controlled by the number in the register at VIC+33 (53281). If the bit pair is 01, the color will come from sprite multicolor register #0 at VIC+37. If the bit pair is 10, the pixel will get its color from the sprite's regular color register. Remember, each sprite has its own color register in one of the locations VIC+39 through VIC+46. And if the bit pair is 11, the color will come from sprite multicolor register #1 at VIC+38. Figure 3-3 summarizes this.

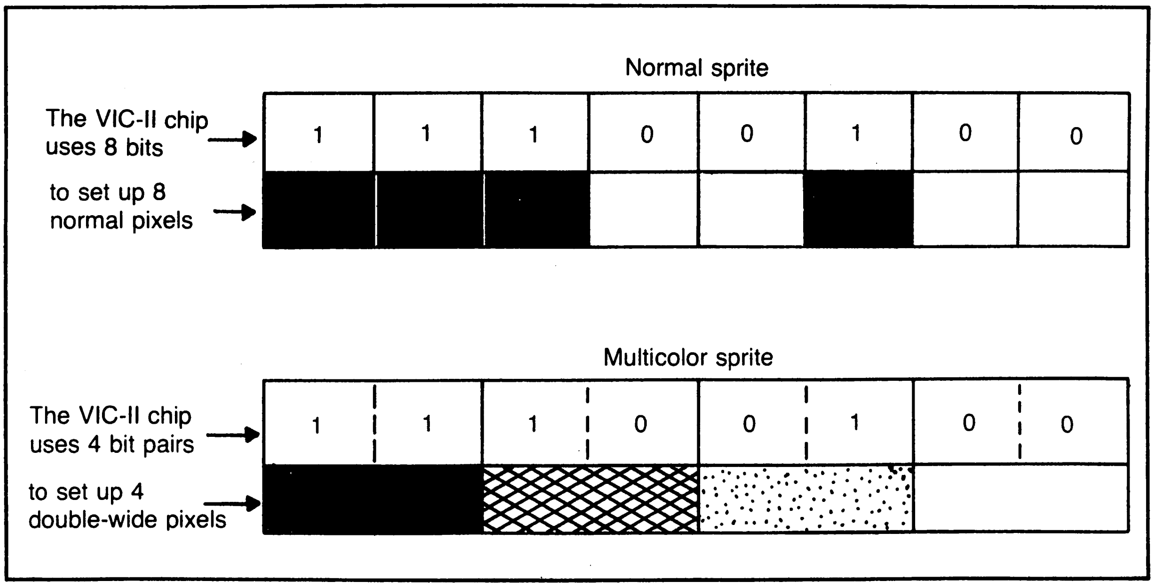



Fig. 3-2. In a multicolor sprite, each bit pair controls the color of one double-wide pixel.

| Bit pair | Description | Location |
|---|---|---|
| 0 0 | Screen color | Vic+33 (53281) |
| 0 1 | Sprite multicolor register #0 | VIC+37 (53285) |
| 1 0 | Sprite color register | One of registers VIC+39 – VIC+46 (53287–53294) |
| 1 1 | Sprite multicolor register #1 | VIC+38 (53286) |

Fig. 3-3. In a multicolor sprite, each bit pair gets its color from a particular VIC register.

| Column number | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | | 9 | | 10 | | 11 | | Number codes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Values | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | | | |
| Row 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 4 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 6 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 7 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 8 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 9 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 10 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 11 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 12 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 13 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 14 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 15 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 16 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 17 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 18 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 19 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 20 | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Transparent screen color | Multicolor register #0 | Sprite color | Multicolor register #1 |
|---|---|---|---|
| 0 0 | 0 1 | 1 0 | 1 1 |

Fig. 3-4. A special coding form for multicolor sprites.

And how do you tell the Commodore 64 that a particular sprite should be displayed in the multicolor mode? The register at VIC+28 is a sprite multicolor selector. Each bit controls one sprite, in the usual relationship: bit #0 controls sprite #1, and so forth. By setting a sprite's bit at VIC+28 to 1, you switch that sprite over to multicolor mode. Setting the bit to 0 puts the sprite back to normal mode.

## 3.3 DESIGNING A MULTICOLOR SPRITE

There may be a few of you who can design a multicolor sprite in your head. The rest of us need some help. Figure 3-4 is a sprite multicolor coding form. It's very similar to the regular sprite coding form of Fig. 1-6. There are still 21 rows, values over each bit position, and three columns for number codes over on the right. However, there are only 12 columns, since our pixels are double wide.

How do you use this form? Refer to Fig. 3-5, which shows a filled in multicolor coding form, as I describe the steps. First, fill in the color-key boxes at the bottom of the form.

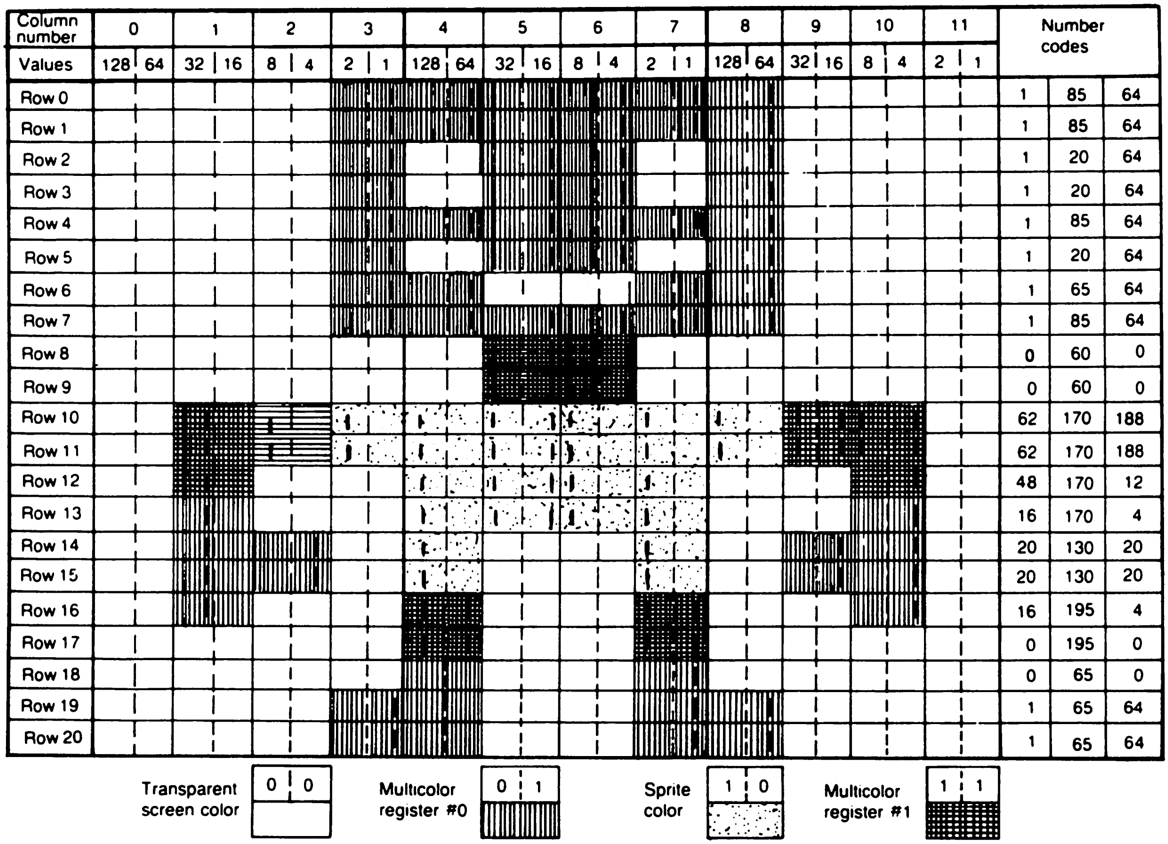



Fig. 3-5. Example of a filled-in multicolor sprite coding form.

```
1000 REM *** 4-COLOR SPRITE ***
1010 :
1020 :
1030 REM ** SET UP SCREEN FEEDBACK
1040 :
1050 PRINT "{CLR}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}THINKING ";
1060 :
1070 :
1080 REM ** LOAD THE SPRITE DATA
1090 :
1100 FOR N = 896 TO 958
1110 :   READ SPDTA
1120 :   POKE N, SPDTA
1130 NEXT N
1140 :
1150 DATA   1,  85,  64,   1,  85,  64
1160 DATA   1,  20,  64,   1,  20,  64
1170 DATA   1,  85,  64,   1,  20,  64
1180 DATA   1,  65,  64,   1,  85,  64
1190 DATA   0,  60,   0,   0,  60,   0
1200 DATA  62, 170, 188,  62, 170, 188
1210 DATA  48, 170,  12,  16, 170,   4
1220 DATA  20, 130,  20,  20, 130,  20
1230 DATA  16, 195,   4,   0, 195,   0
1240 DATA   0,  65,   0,   1,  65,  64
1250 DATA   1,  65,  64
1260 :
1270 :
1280 REM ** SET UP THE SPRITE CONTROLS
1290 :
1300 PRINT "{CLR}";
1310 POKE 2040,14    :REM POINT TO DATA
1320 VIC = 53248     :REM GRAPHICS CHIP
1330 :
1340 POKE VIC,160    :REM HORIZONTAL POS
1350 POKE VIC+1,129  :REM VERTICAL POS
1360 :
1370 POKE VIC+23,1   :REM EXPAND VERTCAL
1380 POKE VIC+29,1   :REM EXPAND HORZTAL
1390 :
1400 POKE VIC+28,1   :REM MULTICOLOR #0
1410 POKE VIC+33,0   :REM BKGRND BLACK
1420 POKE VIC+37,7   :REM MCR #0 YELLOW
1430 POKE VIC+39,5   :REM SPR #0 GREEN
1440 POKE VIC+38,6   :REM MCR #1 BLUE
1450 :
```

```
1460 POKE VIC+21,1   :REM SPRITE #0 ON
1470 :
1480 :
1490 REM ** WAIT FOR KEYPRESS
1500 :
1510 GET KP$
1520 IF KP$ = "" THEN 1510
1530 :
1540 :
1550 REM ** RESET THE SPRITE CONTROLS
1560 :
1570 POKE VIC+21,0   :REM SPRITE OFF
1580 POKE VIC+28,0   :REM MULTICOLOR OFF
1590 POKE VIC+29,0   :REM HORZ EXPND OFF
1600 POKE VIC+23,0   :REM VERT EXPND OFF
1610 :
1620 END
```

Fig. 3-6. Listing of the program 4-Color Sprite.

Give each of the four possible colors a different shade. It's usually simplest to let the screen color be represented by white.

Then fill in the double-wide pixel boxes with a design. Use the shades you've set up in the color-key boxes. When you've got something you like, it's time to fill in with 1's; don't bother with the 0's. Using the color-key boxes at the bottom as a guide, fill in all the bit positions that should have a 1 in them. You may find it easier to do all the pixels for one color before going on to the next color.

Finally, it's time to add up the bit values. For each byte, add all the values of the bits containing a 1. This step is no different than other bit value adding you've done. The sum goes in the appropriate number code box on the right side of the form.

Take another good look at Fig. 3-5. Make sure you understand how I got the 63 number codes. Then make some copies of the multicolor coding form and come up with your own design. You'll get to use it in the next section.

## 3.4 A PROGRAM TO DISPLAY YOUR TECHNICOLOR SPRITES

Figure 3-6 is a listing of the program 4-Color Sprite. The program puts the character designed in Fig. 3-5 onto the screen. A keypress ends the program.

This program is very much like our earlier sprite display programs. The big difference comes in Lines 1400-1440:

```
1400 POKE VIC+28,1   :REM MULTICOLOR #0
1410 POKE VIC+33,0   :REM BKGRND BLACK
1420 POKE VIC+37,7   :REM MCR #0 YELLOW
1430 POKE VIC+39,5   :REM SPR #0 GREEN
1440 POKE VIC+38,6   :REM MCR #1 BLUE
```

Line 1400 sets the sprite multicolor selection register so that sprite #0 will be displayed in multicolor mode. Lines 1410-1440 then set up the four colors that will be used: black, chosen by bit pair 00; yellow, chosen by bit pair 01; green, chosen by bit pair 10; and blue, chosen by bit pair 11.

There is one other difference: at the end of the program, you must reset the multicolor selection register:

```
1580 POKE VIC+28,0    :REM MULTICOLOR OFF
```

A sprite designed for normal display looks pretty strange if it's shown in multicolor mode. If you're wondering how strange, go back to some of our earlier programs and insert lines like 1400-1440 to turn on multicolor mode.

Type in the program 4-Color Sprite if you haven't done so already. Save it, and then run it. Fool around with the color choices in lines 1410-1440 to see if you can come up with a more pleasing combination.

When you're done with that experimentation, it's time to try out your coding. Replace the pixel data in lines 1150-1250 of 4-Color Sprite with the number codes you came up with in the last section. Then rerun the program. How does it look? It may take some tinkering to get the result you had in mind.

## 3.5 OVER AND UNDER

When a sprite travels around the screen, it may cover part of an area used by another sprite. When that happens, a fixed sprite-to-sprite priority determines which sprite shows up in front of the other. Sprite #0 has the highest priority, and sprite #7 has the lowest. Thus, if sprite #0 shares part of the display with sprite #7, sprite #0 will show up in front of sprite #7. Likewise, sprite #4 has priority over sprite #5. Figure 3-7 summarizes these priorities.

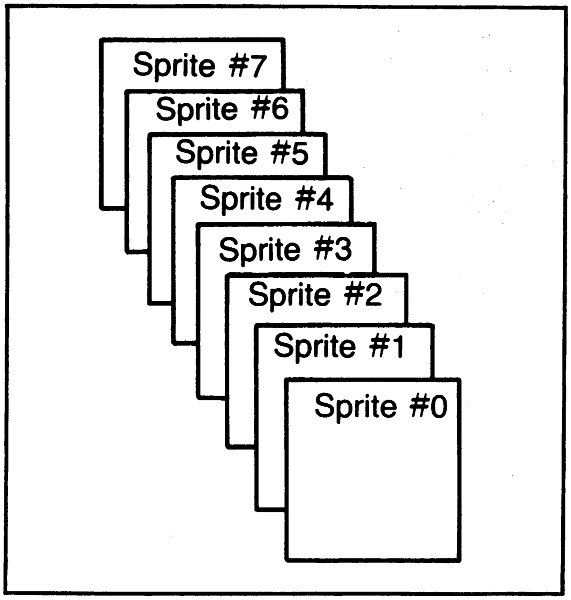



Fig. 3-7. When sprites meet, the highest priority goes to sprites with the lowest numbers, and they show up in front of higher-numbered sprites.

If one sprite is in front of another, it's possible to see parts of the sprite behind it. Those parts of the higher priority sprite that are transparent, that is, the pixels that are set to the screencolor, will act like a window. You'll be able to see parts of the lower priority sprite through this window.

Figure 3-8 is a listing of the program Sprite Overlap. Type it into your computer; save it; then run it. Watch it for a while.

Sprite Overlap puts four similar sprites on the screen and then sets up a never-ending (until you press a key) square dance. Notice how the transparent parts of sprites #1 and #0 let you see parts of the sprites that they're passing over.

This program has two interesting features, besides giving a demonstration of how sprites overlap. The first is the way the sprite shapes are defined. The second is the way the square dance is set up.

```
1000 REM *** SPRITE OVERLAP ***
1010 :
1020 :
1030 REM ** SET UP SCREEN FEEDBACK
1040 :
1050 PRINT "♥QQQQQQQQQQSETTING UP";
1060 :
1070 :
1080 REM ** LOAD THE SPRITE DATA
1090 :
1100 FOR N = 832 TO 894
1110 :    POKE N, 60
1120 NEXT N
1130 :
1140 :
1150 REM ** SET UP THE SPRITE CONTROLS
1160 :
1170 PRINT "♥"        :REM CLEAR SCREEN
1180 VIC = 53248      :REM GRAPHICS CHIP
1190 :
1200 POKE 2040,13     :REM #0 DATA POINTR
1210 POKE 2041,13     :REM #1 DATA POINTR
1220 POKE 2042,13     :REM #2 DATA POINTR
1230 POKE 2043,13     :REM #3 DATA POINTR
1240 :
1250 POKE VIC,226     :REM #0 HORZNTL POS
1260 POKE VIC+2,94    :REM #1 HORZNTL POS
1270 POKE VIC+4,144   :REM #2 HORZNTL POS
1280 POKE VIC+6,176   :REM #3 HORZNTL POS
1290 :
1300 POKE VIC+1,140   :REM #0 VERTCAL POS
1310 POKE VIC+3,118   :REM #1 VERTCAL POS
1320 POKE VIC+5,190   :REM #2 VERTCAL POS
1330 POKE VIC+7,68    :REM #3 VERTCAL POS
1340 :
1350 POKE VIC+39,7    :REM #0 IS YELLOW
1360 POKE VIC+40,5    :REM #1 IS GREEN
1370 POKE VIC+41,3    :REM #2 IS CYAN
1380 POKE VIC+42,1    :REM #3 IS WHITE
1390 :
1400 POKE VIC+23,15   :REM ALL SPRITES
1410 POKE VIC+29,15   :REM DOUBLE-SIZED
1420 :
1430 POKE VIC+21,15   :REM SPRITES 0-3 ON
1440 :
1450 :
1460 REM ** SET UP MOVING REGISTERS
1470 REM    AND INITIAL MOVES
```

```
1480 :
1490 MR(0) = VIC   : MR(1) = VIC+2
1500 MR(2) = VIC+5 : MR(3) = VIC+7
1510 :
1520 MV(0) = -1 : MV(1) = 1
1530 MV(2) = -1 : MV(3) = 1
1540 :
1550 DF = -1 :REM -1:INWARD, 0:OUTWARD
1560 :
1570 :
1580 REM ** MOVE THE SPRITES
1590 :
1600 FOR COUNT = 1 TO 200
1610 :    SPRNUM = INT((COUNT-1)/50)
1620 :    IF DF THEN SPRNUM = 3 - SPRNUM
1630 :    REG = MR(SPRNUM)
1640 :    MOVE = MV(SPRNUM)
1650 :    POKE REG, PEEK(REG) + MOVE
1660 :    GET KP$
1670 :    IF KP$ = "" THEN 1690
1680 :       COUNT = 200 : KEYPRESS = -1
1690 NEXT COUNT
1700 :
1710 :
1720 REM ** IF KEY PRESSED, FINISH UP
1730 :
1740 IF KEYPRESS THEN 1900
1750 :
1760 :
1770 REM ** PAUSE, THEN REVERSE
1780 REM    MOVEMENTS AND REPEAT
1790 :
1800 FOR DELAY = 1 TO 400 : NEXT DELAY
1810 FOR SPRNUM = 0 TO 3
1820 :    MV(SPRNUM) = -1 * MV(SPNUM)
1830 NEXT SPRNUM
1840 DF = -1 - DF
1850 GOTO 1600
1860 :
1870 :
1880 REM ** FINISH UP BY RESETTING
1890 :
1900 POKE VIC+21,0
1910 POKE VIC+29,0
1920 POKE VIC+23,0
1930 :
1940 END
```

Fig. 3-8. Listing of the program Sprite Overlap.

### 3.5.1 Loops That Generate Sprites

Lines 1100-1120 build up the block of sprite pixel data for Sprite Overlap:

```
1100 FOR N = 832 TO 894
1110 :    POKE N, 60
1120 NEXT N
```

You may remember that you used a similar technique in your first program, A Simple Sprite. In that case, though, you poked the number 255, which turned on every pixel in the sprite. In this case, you chose a number, 60, that turns on the middle four pixels of every group of eight. See Fig. 3-9. With three such patterns in each row, you end up with a sprite design made up of three vertical stripes.

You can make a lot of fascinating patterns by changing this loop around. Try typing in these two new lines:

```
1100 FOR N = 832 TO 894 STEP 2
1110 :    POKE N, 255 : POKE N+1, 0
```

Run the new version of the program. Try not to hypnotize yourself. It's an interesting puzzle to see how many complex sprites you can design just through the clever use of loops.

### 3.5.2 Ruminations Upon A Square Dance

At the start of the motion in Sprite Overlap, the four sprites are in the positions shown in Fig. 3-10. One at a time, the sprites will move towards the center of the screen. When all are gathered there, they'll pause and then go back to their original positions, again one at a time. After another brief pause, the motion will repeat itself.

Whenever you think about programming motion, it's useful to look for similarities and repetitive patterns. These patterns can simplify your programming. In this case, each sprite has to follow the same course of action: move inwards, and then move outwards. You can use a program segment that handles these motions for one sprite and then just change the sprite it works with. If you set up your motion variables as arrays, it will be easy to switch sprites: just vary the array subscripts in the motion segment.

There's another useful simplification to be made. Inwards and outwards motion will only differ in the direction a sprite travels. All you need to do is reverse the direction of a sprite's motion between repetitions of the motion segment. Thus the same program segment will be able to move all four sprites both inwards and outwards. Only the details need to be worked out (famous last words of many programmers).

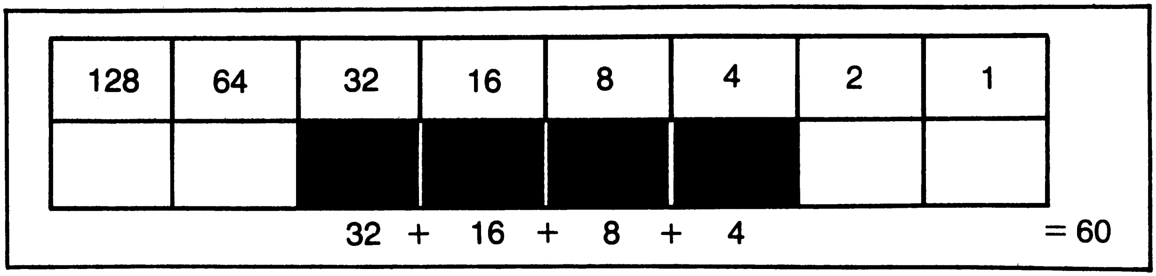



Fig. 3-9. Poking the number 60 as sprite data turns on the middle four pixels in each group of eight.

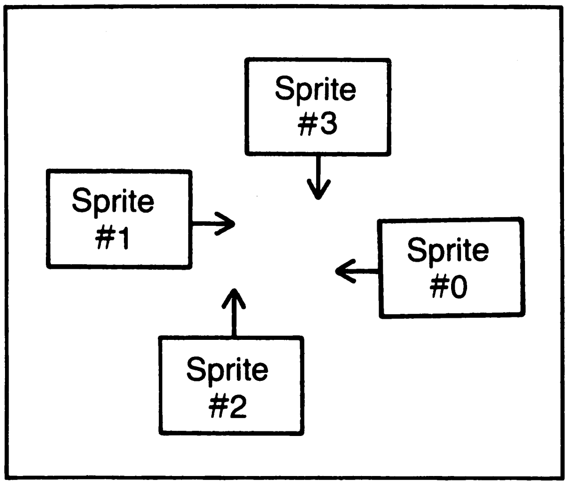



Fig. 3-10. Initial positions of the four sprites in Sprite Overlap, with arrows indicating the direction they'll first move in.

### 3.5.3 Setting Up Registers and Motions

Since any one sprite will only be moved vertically or horizontally, only one position register will be needed to move that sprite. Lines 1490-1500 set up the four registers that will be used for sprite moves:

```
1490 MR(0) = VIC   : MR(1) = VIC+2
1500 MR(2) = VIC+5 : MR(3) = VIC+7
```

Take another look at Fig. 3-10. Sprites #0 and #1 will be moving horizontally, and sprites #2 and #3 will be moving vertically. I used this information to figure out which position registers to use.

Lines 1520-1530 give each sprite an initial move:

```
1520 MV(0) = -1 : MV(1) = 1
1530 MV(2) = -1 : MV(3) = 1
```

The value of this move variable will be added to a sprite's current position to give it a new position. To check the logic behind these assignments, refer again to Fig. 3-10. The arrows indicate the direction of each sprite's initial motion. For example, sprite #3 will start out moving downwards. Each time it moves, its vertical position should increase, and that's what the move assigned to sprite #3 by line 1530 will do. When it comes time for sprite #3 to reverse its motion, you'll just multiply the value of MV(3) by −1. Then the sprite's vertical position will decrease by 1 each time, and it will move upwards.

You have one more item to consider: the order in which the sprites will move. When the motion is inwards, you want the order of moves to be #3, #2, #1, #0. When the sprites move outwards, you want to move #0 first, followed by #1, #2, and #3. The order will just reverse itself. Line 1550 sets up a variable that will keep track of inwards and outwards, so you get the correct order of sprite motions:

```
1550 DF = -1 :REM -1:INWARD, 0:OUTWARD
```

### 3.5.4 The All-Purpose Motion Loop

Lines 1600-1690 move the sprites:

```
1600 FOR COUNT = 1 TO 200
1610 :    SPRNUM = INT((COUNT-1)/50)
1620 :    IF DF THEN SPRNUM = 3 - SPRNUM
1630 :    REG = MR(SPRNUM)
1640 :    MOVE = MV(SPRNUM)
1650 :    POKE REG, PEEK(REG) + MOVE
1660 :    GET KP$
1670 :    IF KP$ = "" THEN 1690
1680 :       COUNT = 200 : KEYPRESS = -1
1690 NEXT COUNT
```

Lines 1600 and 1690 set up a loop that will be carried out 200 times, unless a keypress interrupts to end the program. Each sprite will move 50 times, 1 pixel at a time, and there are 4 sprites to move. 4 × 50 gives you 200.

Lines 1610 and 1620 figure out which

sprite should be moved, and store its number in the variable SPRNUM. If the sprites are moving inwards, DF will have the value −1. SPRNUM will take on the values 3, 2, 1, and then 0 as the loop progresses. If the sprites are moving outwards, DF will have the value 0. Now SPRNUM will take on the values 0, 1, 2, and then 3, just as you want.

Line 1630 picks the position register to adjust, based on SPRNUM, and line 1640 selects the sprite's move. Line 1650 does the actual work, taking the old position of the selected sprite and adding the appropriate move.

Line 1660 checks for any pressed keys. If there are any, line 1670 sets up a quick exit from the program. Take another look at Section 2.6.2 if you forget how this works.

If the sprites have just moved inwards, you want to set them to go outwards. And if they've gone outwards, you want to get them ready to go inwards again. Lines 1810-1850 prepare for the next round of the dance:

```
1810 FOR SPRNUM = 0 TO 3
1820 :    MU(SPRNUM) = -1 * MU(SPNUM)
1830 NEXT SPRNUM
1840 DF = -1 - DF
1850 GOTO 1600
```

First, each sprite's move is reversed by multiplying it by −1. Then the inward/outward variable is switched around in line 1840. If it was set to −1, it becomes 0, and if it was set to 0, it becomes −1. Then line 1850 sends the program back to the main dance loop, starting at line 1600. The program will run, with the sprites moving in and then out, until a key is pressed or the plug is pulled.

Here's a great opportunity to dive right in and play with motion loops. Make some changes to Sprite Overlap so you get other sprite dances. Here are some ideas if your imagination is out to lunch:

- Get two sprites to move at a time.
- Get the sprites to move to new starting positions when they move outwards.
- Have the sprites cover each other completely when they overlap.

## 3.6 BRING ON THE FANCY CARTOONS

Animation is a great form of magic. By quickly showing a series of still pictures, you can create the illusion of motion and life. So far, our sprites have had very limited animation. An image moves around the screen, but it doesn't change its form. It's like a cheap Saturday morning cartoon show.

Now you're going to try some fuller animation, where the image itself changes. This is easy to do with sprites. You start by loading several sprite images. Then, you set up a loop that cycles a sprite's data pointer through the images.

### 3.6.1 Developing the Images

Let's set up one of these animation cycles. Figure 3-11 shows three images of a juggler. Notice how the action progresses from image to image and how the last image leads back to the first. Setting up a cycle of images takes some tinkering. I'll usually come up with a preliminary set of images and then run a program to display them. Next, I fool around with the data until I get the effect I want. Ideas for additions and changes to the animation pop up,

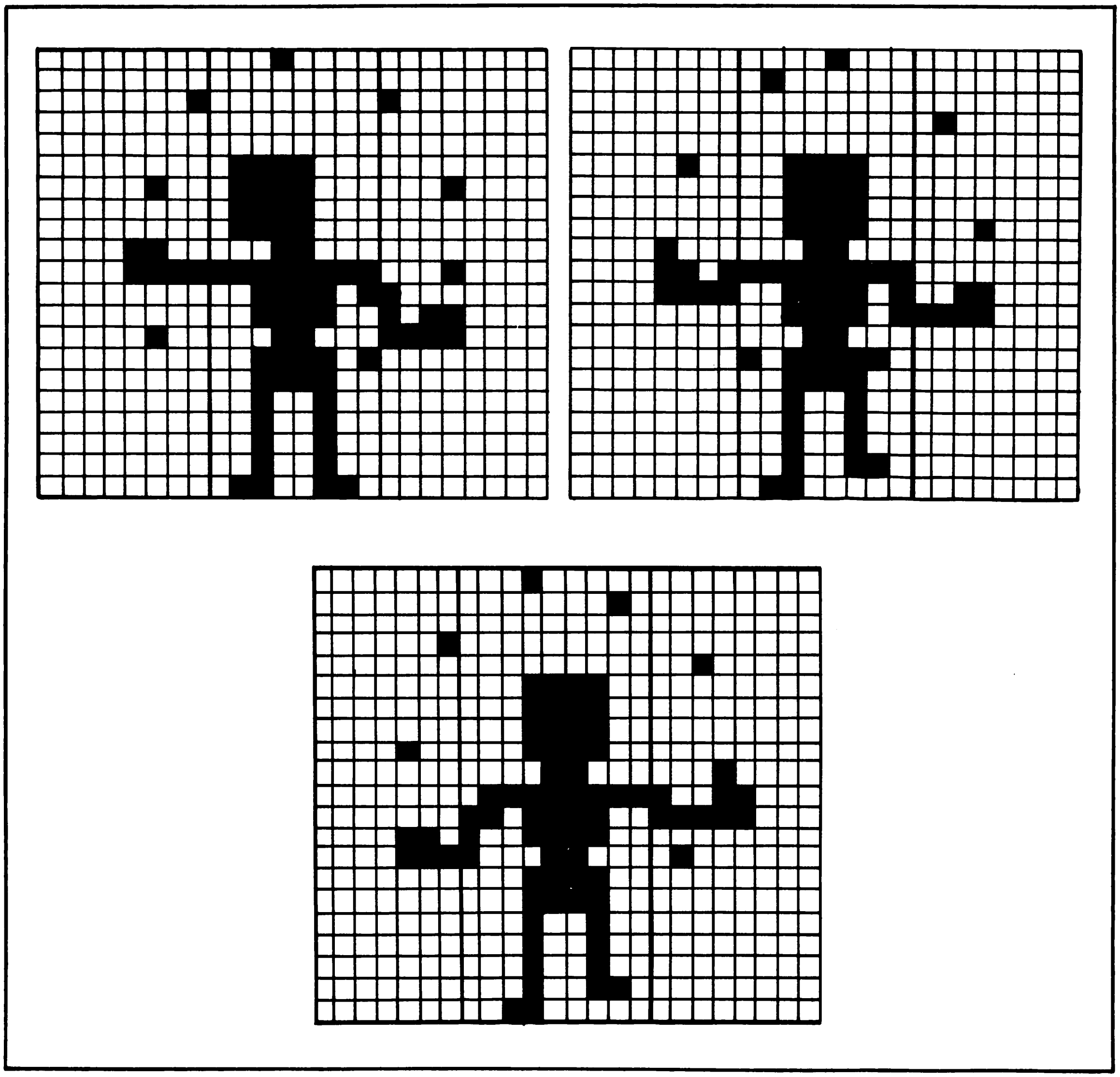


Fig. 3-11. The three sprite images used to animate a juggler in the program Juggling Fool.

get tried out, and then are kept or discarded. The images in Fig. 3-11 are the end result of such a process.

Once the images are developed, you can use the animation cycle in many different programs. After a while, you can develop a whole library of these animated image sets.

Now it's your turn. Using the sprite coding forms, develop a preliminary set of three images that form an animation cycle. The action in each image should lead to the next, and the last should lead to the first. If you're short on ideas, here are some suggestions for simple cycles: • a bouncing ball • an eye that opens • a line that grows and shrinks • a face that smiles • a star that twinkles • a blizzard. Figure out the number codes for each image. You'll use them in Section 3.6.3.

```
1000 REM *** JUGGLING FOOL ***
1010 :
1020 :
1030 REM ** SET UP SCREEN FEEDBACK
1040 :
1050 PRINT "♥QQQQQQQQQSETTING UP";
1060 :
1070 :
1080 REM ** LOAD THE SPRITE DATA
1090 :
1100 FOR N = 832 TO 1023
1110 :    READ SPDTA
1120 :    IF SPDTA = -1 THEN
            PRINT " ."; : GOTO 1140
1130 :    POKE N, SPDTA
1140 NEXT N
1150 :
1160 DATA   0,  16,   0,   0,   0,   0
1170 DATA   1,   0, 128,   0,   0,   0
1180 DATA   0,   0,   0,   0, 120,   0
1190 DATA   4, 120,  16,   0, 120,   0
1200 DATA   0, 120,   0,  12,  24,   0
1210 DATA  15, 255,  16,   0,  61, 128
1220 DATA   0,  60, 176,   4,  24, 240
1230 DATA   0,  61,   0,   0,  60,   0
1240 DATA   0,  36,   0,   0,  36,   0
1250 DATA   0,  36,   0,   0,  36,   0
1260 DATA   0, 102,   0,  -1
1270 :
1280 DATA   0,   8,   0,   0,  64,   0
1290 DATA   0,   0,   0,   0,   0,  64
1300 DATA   0,   0,   0,   4,  60,   0
1310 DATA   0,  60,   0,   0,  60,   0
1320 DATA   0,  60,  16,   8,  24,   0
1330 DATA  13, 255,   0,  15,  61,  48
1340 DATA   0,  61, 240,   0,  24,   0
1350 DATA   0, 190,   0,   0,  60,   0
1360 DATA   0,  36,   0,   0,  36,   0
1370 DATA   0,  36,   0,   0,  38,   0
1380 DATA   0,  96,   0,  -1
```

```
1390 :
1400 DATA   0,  32,   0,   0,   2,   0
1410 DATA   0,   0,   0,   2,   0,   0
1420 DATA   0,   0,  32,   0,  60,   0
1430 DATA   0,  60,   0,   0,  60,   0
1440 DATA   8,  60,   0,   0,  24,  16
1450 DATA   0, 255, 152,   1, 188, 248
1460 DATA  13,  60,   0,  15,  24,  64
1470 DATA   0,  60,   0,   0,  60,   0
1480 DATA   0,  36,   0,   0,  36,   0
1490 DATA   0,  36,   0,   0,  38,   0
1500 DATA   0,  96,   0,  -1
1510 :
1520 :
1530 REM ** SET UP THE SPRITE CONTROLS
1540 :
1550 PRINT "♥"        :REM CLEAR SCREEN
1560 VIC = 53248      :REM GRAPHICS CHIP
1570 :
1580 POKE 2040,13     :REM #0 DATA POINTR
1590 POKE VIC,160     :REM #0 HORZNTL POS
1600 POKE VIC+1,129   :REM #0 VERTCAL POS
1610 POKE VIC+39,1    :REM #0 IS WHITE
1620 POKE VIC+23,1    :REM SPRITE #0 IS
1630 POKE VIC+29,1    :REM DOUBLE-SIZED
1640 POKE VIC+21,1    :REM SPRITE #0 ON
1650 :
1660 :
1670 REM ** JUGGLE
1680 :
1690 IMAGE = PEEK (2040) + 1
1700 IF IMAGE = 16 THEN IMAGE = 13
1710 POKE 2040, IMAGE
1720 :
1730 FOR DELAY = 1 TO 30 : NEXT DELAY
1740 :
1750 :
1760 REM ** GET KEYPRESS TO END
1770 :
1780 GET KP$
1790 IF KP$ = "" THEN 1690
1800 :
1810 POKE VIC+21,0
1820 POKE VIC+29,0
1830 POKE VIC+23,0
1840 :
1850 END
```

Fig. 3-12. Listing of the program Juggling Fool.

### 3.6.2 The Juggling Fool

Figure 3-12 is a listing of the program Juggling Fool. It displays the images shown in Fig. 3-11. Let's do a brief analysis of some of its features.

Lines 1100-1130 load in the sprite definition data. Line 1120 is an interesting trick:

```
1120 :    IF SPDTA = -1 THEN
             PRINT " .";  : GOTO 1140
```

Sprite definitions fill 63 memory locations. But they're stored at intervals of 64 memory locations (check back to Section 2.3 and Fig. 2-11). If you're filling memory blocks that follow one another, you can keep the loading loop simple by just adding a 64th byte of dummy data to the data lists. That way, the data for all the sprite images can be loaded consecutively. And, if you choose the dummy byte to be a value that normally won't come up, you can recognize it and print out some loading feedback. In this program, the dummy value is −1; when it is read, the program will add a period ( . ) to the screen feedback display. The period tells us another image block has been read into memory.

The three sprite image data blocks are in memory locations 832-894, 896-958, and 960-1022. Dividing the starting address of each block by 64, you get sprite pointer values of 13, 14, and 15, respectively. The program will perform its animation by continually changing the pointer value for sprite #0, which is set at location 2040. The value will go from 13 to 14 to 15 and then back to 13 for another cycle.

Lines 1580-1640 set up initial values for the sprite controls. There is nothing new here. The data pointer for sprite #0 starts out with the value 13; the initial image will be the one stored at memory locations 832-894.

Lines 1690-1710 switch images:

```
1690 IMAGE = PEEK (2040) + 1
1700 IF IMAGE = 16 THEN IMAGE = 13
1710 POKE 2040, IMAGE
```

Line 1690 takes the current pointer value and adds 1 to it. If the new value is 16, Line 1700 sets it back to 13. Then line 1710 inserts the new value into the pointer location. Thus, the pointer will do what we want, going from 13 to 14 to 15 and then back to 13 again.

Line 1730 is a simple delay loop. By changing the length of the delay, the juggler will juggle at different rates of speed. And finally, lines 1780-1790 check for a keypress. If no key has been pressed, the program jumps back to line 1690 to display the next image. If there has been a keypress, the program cleans up the sprite settings and ends.

### 3.6.3 Now It's Your Turn

Pull out the coding sheets you created at the end of Section 3.6.1. Use the number codes to replace the data in lines 1160-1500 of Juggling Fool. Then run the new program. How does it look? Play with the program until you get an animation cycle you like. Change the timing, the data, and the order the images are shown in. You'll learn a lot about animation by such exploration.

## 3.7 CHAPTER SUMMARY

Let's recap what you've learned in this chapter:

* How to set up the VIC-II registers so a sprite is displayed in four colors
* How to design such a multicolor sprite
* What happens when sprites overlap one another
* More about setting up motions for many sprites
* How to set up an animation cycle by shifting a sprite's data pointer from one image block to another

Using a book this size, you can only begin to study sprite graphics techniques. Advanced knowledge will only come when you sit down and play with sprites for a while. In the next two chapters, you'll look at two other types of Commodore 64 picture magic: character and bit-mapped graphics.

## 3.8 EXERCISES

### 3.8.1 Self Test

Answers are supplied in Section 3.8.3. The numbers in parentheses tell you which chapter section to go to for help.

1. (3.1) In sprite multicolor mode, using two bits lets a double-wide pixel take on one of ________ possible colors.
2. (3.1) Since sprites in multicolor mode are only 12 double-wide pixels across, we say that they have less ________ resolution.
3. (3.2) If you poke the value 15 into the sprite multicolor selection register at VIC + 28, which sprites will be displayed in multicolor mode?
4. (3.5) When sprites cross paths, sprite # ________ has display priority over all the other sprites.
5. (3.5.1) In the program Sprite Overlap, describe the sprites that result if you type in these three lines:

```
1100 FOR N = 832 TO 894 STEP 3
1105 :    POKE N, 225
1110 :    POKE N+1, 195
1115 :    POKE N+2, 135
```

6. (3.5.4) Take a look at lines 1610-1620 of Sprite Overlap. If COUNT has the value 120, and DF has the value 0, what will lines 1610 and 1620 set SPRNUM to?
7. (3.6.2) How many periods (.) will get printed next to the words SETTING UP as the sprite data is loaded during the program Juggling Fool?
8. (3.6.2) What happens to the juggler in Juggling Fool if you change the delay time in line 1730 from 30 to 100?

### 3.8.2 Programming Exercises

1. Change the program 4-Color Sprite so that a second sprite, based on the same sprite data, is also displayed in multicolor mode.
2. Change the program Sprite Overlap so that the four sprites overlap completely at the center of the screen.
3. Change the program Juggling Fool so that the juggler juggles in a clockwise direction for a while, then switches to counter-clockwise, then goes back to clockwise, and so forth.

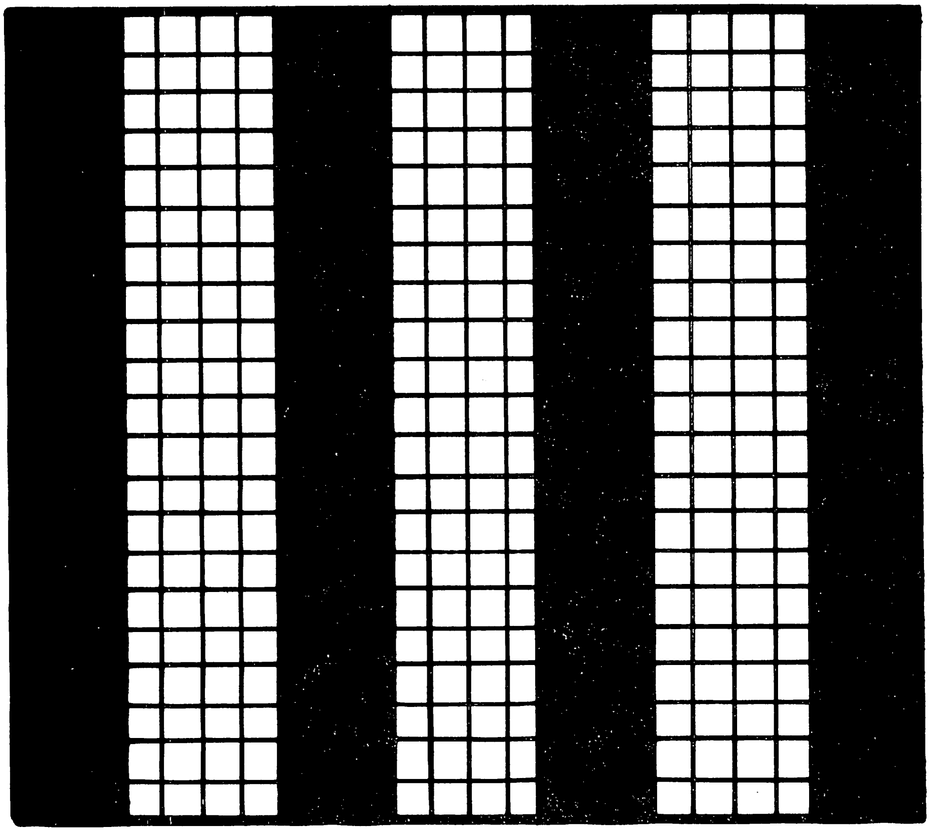


Fig. 3-13. Sprite that results from typing the changes to Sprite Overlap mentioned in Self Test, item 5.

### 3.8.3 Answers to Self Test

1. four
2. horizontal
3. #0, #1, #2, and #3
4. 0
5. each sprite will be made up of four vertical stripes - see Fig. 3-13
6. SPRNUM will be set to 2
7. three periods
8. the juggling will slow down

### 3.8.4 Possible Solutions To Programming Exercises

These solutions are based on adding and/or changing lines in the original programs.

1. Load in the program 4-Color Sprite. Then type in these lines:

```
1000 REM *** TWO 4-COLOR SPRITES ***
1315 POKE 2041,14   :REM SPRITE #1 PNTR
1353 POKE VIC+2,160 :REM SPRITE #1 HP
1356 POKE VIC+3,69  :REM SPRITE #1 VP
1370 POKE VIC+23,3  :REM EXPAND VERTCAL
```

```
1380 POKE VIC+29,3   :REM EXPAND HORZTAL
1400 POKE VIC+28,3   :REM MULTICOLOR 0&1
1435 POKE VIC+40,2   :REM SPR #1 RED
1460 POKE VIC+21,3   :REM SPRITE 0&1 ON
```

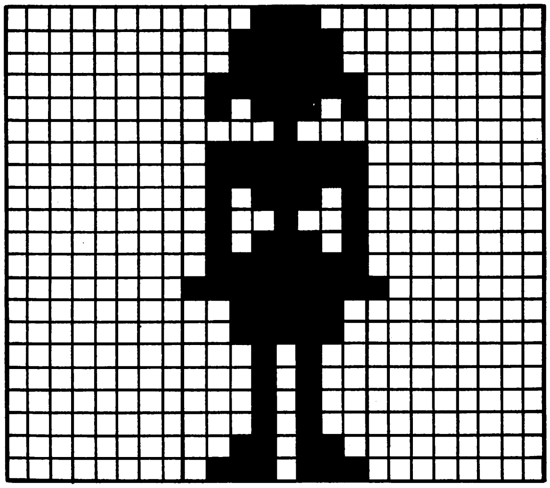
2. Load in the program Sprite Overlap. Then type in these lines:

```
1000 REM *** TOTAL OVERLAP ***
1250 POKE VIC,210    :REM #0 HORZNTL POS
1260 POKE VIC+2,110  :REM #1 HORZNTL POS
1270 POKE VIC+4,160  :REM #2 HORZNTL POS
1280 POKE VIC+6,160  :REM #3 HORZNTL POS
1300 POKE VIC+1,129  :REM #0 VERTCAL POS
1310 POKE VIC+3,129  :REM #1 VERTCAL POS
1320 POKE VIC+5,179  :REM #2 VERTCAL POS
1330 POKE VIC+7,79   :REM #3 VERTCAL POS
```

3. Load in the program Juggling Fool. Then type in these lines:

```
1000 REM *** SWITCH JUGGLER ***
1655 JUGDIR = 1      :REM CLOCKWISE JUGL
1690 IMAGE = PEEK (2040) + JUGDIR
1705 IF IMAGE = 12 THEN IMAGE = 15
1712 :
1715 COUNT = COUNT + 1
1718 IF INT (COUNT/27) = COUNT/27 THEN
        JUGDIR = -JUGDIR : COUNT = 0
```

# Chapter 4



# Character Graphics

The Commodore 64 has some powerful text display capabilities. In this chapter, you'll explore some of them. You'll learn about the built-in character sets and get to poke about in the screen and color memories. You'll build up strings of graphics characters and fly them around the screen. You'll learn how to modify the built-in character sets, and finally, you'll see how to design a character set for use in animation.

## 4.1 LET'S PLAY

It's time to do a little keyboard exploration. Sit down at your Commodore 64. Type in this command:

```
POKE 650, 128
```

In case you hadn't known, sticking a number greater than 127 into memory location 650 makes all the keys repeat when they're held down long enough. Repeating keys are fun to draw with. To go back to the normal situation, where only a few keys repeat, put a 0 into the same location.

Now, clear the screen. Pretend your TV screen is a blank artist's canvas. Using the various graphics characters, type some pretty designs. A few keys will come in especially handy: shift, the Commodore logo key, CTRL (control), the color keys, the RVS (reverse) ON and RVS OFF keys, and the cursor control keys. There are 512 different characters built into the Commodore 64's permanent memory; you can get some interesting designs with this simple drawing technique. Figure 4-1 is a screen printout of one such design.

## 4.2 SCREEN AND COLOR MEMORY

The 64 normally displays 25 text lines,

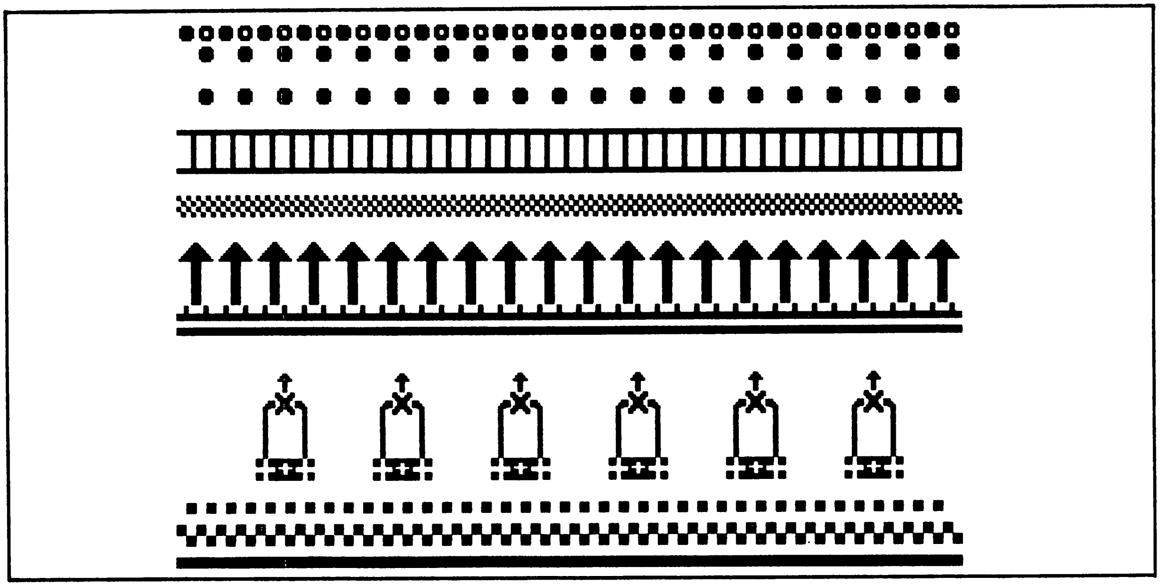


Fig. 4-1. Printout of a picture drawn on the screen by typing some of the Commodore 64's 512 built-in characters.

each containing 40 characters. That gives 1000 screen locations. Codes that determine which character is shown at a location are stored in what's called screen memory. The 64's wonderful flexibility lets you move this screen memory around if you want to. Normally, it occupies the thousand memory locations 1024-2023.

There's a second block of 1000 memory locations that control the color for each screen location. This area of memory, called color memory, occupies memory locations 55296-56295. This color memory is a bit stunted; each location can only hold four bits, which limits it to integers from 0 to 15. Since there are only 16 possible colors, this is okay.

So each location on the text screen normally has two memory locations associated with it. One, in screen memory, determines which one of 256 characters will show up. The second, in color memory, determines the color the character will take on. Appendices B and C map out the screen and color memory areas.

## 4.3 GETTING CHARACTERS ON THE SCREEN

The VIC-II chip controls the display of screen characters. It scans the screen memory locations many times each second. These locations contain values between 0 and 255. Based on the values found there, VIC goes to the section of memory where patterns for drawing all the different characters are stored. It uses those patterns and the information in the color memory locations to send the correct electrical signals to the TV set.

The Commodore 64 has patterns for two complete character sets stored in a part of its permanent memory. Each set contains the patterns for 256 characters. The device the sets are stored in is called a character generator ROM. Let's take a look at all of these built-in characters.

## 4.4 DISPLAYING ALL 512 BUILT-IN CHARACTERS

Figure 4-2 is a listing of the program Character ROM Display. Type it in, save it, and then run it. When the display starts, the first 256 characters appear. To see the second 256, just press the shift and Commodore logo keys at the same time. They operate as a toggle switch between the two character sets.

```
1000 REM *** CHAR ROM DISPLAY ***
1010 :
1020 :
1030 REM ** CLEAR SCREEN AND
1040 REM    SET UP CONSTANTS
1050 :
1060 PRINT "♥"
1070 :
1080 SCRMAP = 1024
1090 COLMAP = 55296
1100 :
1110 :
1120 REM ** THE BIG DISPLAY LOOP
1130 :
1140 FOR POCODE = 0 TO 255
1150 :    ROW = INT (POCODE / 20)
1160 :    CLM = POCODE - (20 * ROW)
1170 :
1180 :    EVROW = (ROW/2 = INT(ROW/2))
1190 :    CLM = (CLM * 2) - EVROW
1200 :    ROW = ROW * 2
1210 :
1220 :    SPOT = (ROW * 40) + CLM
1230 :
1240 :    POKE SCRMAP + SPOT, POCODE
1250 :    POKE COLMAP + SPOT, 1
1260 NEXT POCODE
1270 :
1280 :
1290 REM ** GET KEYPRESS TO END
1300 :
1310 GET KP$
1320 IF KP$ = "" THEN 1310
1330 :
1340 PRINT "♥";
1350 END
```

Fig. 4-2. Listing of the program Character ROM Display.

The operation of the program is simple in principle, but a bit complex in execution. You just want to poke each of the values between 0 and 255 into a screen memory location. That's the purpose of the loop in lines 1140-1260. The complexities come in when you figure the locations to poke to get a pleasing display. That's what all of the nuttiness in lines 1150-1220 does. Lines 1240-1250 do the poking work:

```
1240 :    POKE SCRMAP + SPOT, POCODE
1250 :    POKE COLMAP + SPOT, 1
```

```
1000 REM *** FLY THE FACE ***
1010 :
1020 :
1030 REM ** BUILD THE STRING
1040 :
1050 F$(1) = "          "
1060 F$(2) = "          "
1070 F$(3) = "          "
1080 F$(4) = "          "
1090 F$(5) = "          "
1100 F$(6) = "          "
1110 F$(7) = "          "
1120 :
1130 D1L9$ = "          "
1140 U7$ = "       "
1150 :
1160 FOR N = 1 TO 7
1170 :    F$ = F$ + F$(N) + D1L9$
1180 NEXT N
1190 F$ = F$ + U7$
1200 :
1210 :
1220 REM ** START OUT TIDY AT MIDSCREEN
1230 :
1240 PRINT "                              ";
1250 PRINT F$;       :REM PRINT FACE
1260 :
1270 :
1280 REM ** WAIT FOR A KEYPRESS
1290 :
1300 POKE 650,128  :REM ALL KEYS REPEAT
1310 GET KP$
1320 IF KP$ = "" THEN 1310
1330 :
1340 :
1350 REM ** DECIPHER KEYPRESS
1360 :
1370 IF KP$ = " " THEN 1440  :REM UP
1380 IF KP$ = " " THEN 1440  :REM DOWN
1390 IF KP$ = " " THEN 1440  :REM RIGHT
1400 IF KP$ = " " THEN 1440  :REM LEFT
1410 IF KP$ = " " THEN 1500  :REM SPACE
1420 GOTO 1310    :REM NO MATCH
1430 :
1440 PRINT KP$ ;  :REM MOVE CURSOR
1450 GOTO 1250    :REM PRINT FACE
1460 :
1470 :
```

```
1480 REM ** SPACE ENDS IT
1490 :
1500 PRINT "♥";    :REM CLEAN UP
1510 END
```

Fig. 4-3. Listing of the program Fly the Face.

Besides putting a character code into screen memory, you put the value 1 into the corresponding color memory location. That way, the character will show up in color 1, white.

You should create some variations on this program. Have it print characters in different colors, or have the characters displayed in different locations.

## 4.5 BUILD A CHARACTER STRING AND FLY IT

Figure 4-3 is a listing of the program Fly the Face. The program demonstrates a way to build moving pictures out of characters. Type the program in and run it. Pressing one of the cursor motion keys (up, down, left, or right) will move the smiling face, and pressing the spacebar will end the program.

By the way, there's a reason the lines in this program listing are closer together than usual. They're spaced the way they appear on the TV screen, so you can see how the graphics characters go together to form the face.

### 4.5.1 Building the String

The first part of the program builds a special string. This string, named F$, contains blank spaces, graphics characters, and cursor movement commands. When this string is printed, the smiling face will show up on the screen just as it looks in the listing.

Lines 1050-1140 set up the pieces that'll go into F$. Lines 1160-1190 put them together:

```
1160 FOR N = 1 TO 7
1170 :   F$ = F$ + F$(N) + D1L9$
1180 NEXT N
1190 F$ = F$ + U7$
```

After each graphics piece comes a cursor-movement piece. Some characters get printed, and then the cursor moves down a line and back to the left. Line 1190 adds a final cursor-movement piece to get the cursor back up to its starting position.

### 4.5.2 Flying the String

Line 1240 clears the screen, and then puts the cursor near the middle. Line 1250 draws the face string you built up:

```
1250 PRINT F$;     :REM PRINT FACE
```

Finally, the program enters the flying phase. Line 1300 sets the keyboard for auto-repeat. Then lines 1310-1320 wait for a keypress. When there is one, it's stored in KP$.

Lines 1370-1420 decipher KP$:

```
1370 IF KP$ = "□" THEN 1440   :REM UP
1380 IF KP$ = "◙" THEN 1440   :REM DOWN
1390 IF KP$ = "▙" THEN 1440   :REM RIGHT
1400 IF KP$ = "▌▌" THEN 1440   :REM LEFT
1410 IF KP$ = " " THEN 1500   :REM SPACE
1420 GOTO 1310    :REM NO MATCH
```

If the keypress is one of the four cursor moves, up, down, left, or right, the program jumps to line 1440. If the spacebar was pressed, the program jumps to line 1500 to end itself. If the key pressed was not one of the above, the program just loops back to read the keyboard at line 1310.

What happens if one of the cursor motion keys was pressed?

```
1440 PRINT KP$ ;  :REM MOVE CURSOR
1450 GOTO 1250    :REM PRINT FACE
```

You just print the keypress, which moves the cursor. Then the program jumps back to line 1250, prints the face in its new position, and goes on to get another keypress.

### 4.5.3 Carrying Your Own Eraser

You may be wondering why the flying face was drawn surrounded by a ring of spaces. This is what I call the carry-your-own eraser technique. The face can only move one position at a time. You don't bother to erase the old face when you move it to a new position. When the face is drawn in a new position, it covers up most of the old face. The outer ring of spaces covers up any remaining parts. If you wanted the face to move two positions at a time, the ring of spaces would have to be two spaces wide.

If you didn't use this technique, you'd have to completely erase the face at its old position before drawing the new face. That would eat up precious time. In animation, you're always trying to move and draw objects as quickly as possible.

### 4.5.4 Flying Your Own Face

It's time to apply some of the knowledge you picked up playing with the keyboard in Section 4.1. Change lines 1050-1190 so a different image flies around the screen. If you want to get especially fancy, imbed some color-setting characters in your string. Try adding some other functions chosen by keypresses. For fun, create an image that's not surrounded by a ring of self-erasing spaces.

## 4.6 MORE ABOUT THE CHARACTER MEMORY

When you crank up your Commodore 64, it gets its character patterns from the built-in character generator ROM. A ROM is a memory device that can only be read from. The character patterns are put into it when it's manufactured. You can't put new information into a ROM.

However, you can tell the VIC-II chip to get its patterns from other areas of memory. Those areas can be RAM memory, which can be written to and read from. So you can insert your own character patterns for the VIC chip to use.

The VIC-II chip looks at 16K, 16384 bytes, of memory at a time. A complete set of patterns for 256 characters takes up 2K, 2048 bytes, of memory. Thus, there are eight possible locations for the 2K character memory block in a 16K bank.

Bits 1, 2, and 3 of the register located at VIC+24 (53272) tell VIC where to find the character patterns. When the machine is first turned on, it looks at the 2K block that begins at location 4096 and finds the first 256 patterns stored in the character generator ROM. If you press the shift key and the Commodore logo key together, new values get stored in VIC+

24. VIC now looks at the 2K block of character patterns that begin at location 6144 and displays characters from the second set of 256 characters stored in the ROM.

If you want to use other characters, you need to fill a 2K block of RAM with the patterns and then set the pointers in bits 1, 2, and 3 of VIC+24. The pattern for each character uses up eight bytes; it's a large job to figure out patterns for a full set of 256 characters. There is a shortcut, however.

In many cases, you only want to change a few character patterns. So you can copy a set of patterns from the character generator ROM into RAM memory and then just change a few of them.

### 4.7 MOVING THE CHARACTER ROM INTO RAM

There are a few complications involved in moving the patterns from the character ROM into RAM. First, the character ROM is a bit of a trickster. It spends a lot of time appearing to be at different memory locations. Now it's at one place, now it's at another. You need to tie it down to one area long enough to copy its contents.

That brings up the second complication. When you manage to tie the ROM down, it lands in the memory area normally used by the Commodore's input/output devices. With the ROM brought into memory, the computer can't communicate with the outside world. If it tries to do some I/O (input/output) operation, it'll go to never-never land.

Now there's one I/O operation that your Commodore tries to do 60 times each second: scan the keyboard. You'll need to turn that operation off while you transfer ROM to RAM. It's like clamping arteries shut during an operation.

```
1000 REM *** CHAR ROM TO RAM ***
1010 :
1020 :
1030 REM ** SET UP FEEDBACK
1040 :
1050 PRINT "{CLR}{10 DOWN}MOVING";
1060 :
1070 :
1080 REM ** SET UP FOR TRANSFER
1090 :
1100 POKE 56334, PEEK (56334) AND 254
1110 REM ** KEYSCAN INTERRUPT OFF
1120 :
1130 POKE 1, PEEK (1) AND 251
1140 REM ** BRING ROM INTO MEMORY
1150 :
1160 ROM = 53248 :REM START OF CHAR ROM
1170 RAM = 12288 :REM WHERE IT'LL GO TO
1180 :
1190 :
```

```
1200 REM ** TRANSFER, WITH FEEDBACK
1210 :
1220 FOR CHAR = 0 TO 255
1230 :    SR = ROM + (CHAR * 8)
1240 :    DS = RAM + (CHAR * 8)
1250 :
1260 :    FOR BYTE = 0 TO 7
1270 :        POKE DS + BYTE,
                   PEEK (SR + BYTE)
1280 :    NEXT BYTE
1290 :
1300 :    POKE 1, PEEK(1) OR 4
1310 :    PRINT ".";
1320 :    POKE 1, PEEK(1) AND 251
1330 NEXT CHAR
1340 :
1350 :
1360 REM ** CLEAN UP
1370 :
1380 POKE 1, PEEK (1) OR 4
1390 POKE 56334, PEEK (56334) OR 1
1400 :
1410 VIC = 53248 : CPTR = VIC+24
1420 PTR = PEEK (CPTR) AND 241
1430 PTR = PTR OR 12
1440 POKE CPTR, PTR
1450 :
1460 PRINT : PRINT "DONE."
1470 END
```

Fig. 4-4. Listing of the program Character ROM to RAM.

## 4.8 A PRACTICAL EXAMPLE

Figure 4-4 is a listing of the program Character ROM to RAM. Let's see how it handles the transfer. Line 1100 turns off the keyboard scanning:

```
1100 POKE 56334, PEEK (56334) AND 254
```

This statement puts a 0 into bit 0 of location 56334, and leaves the other bits alone. That stops the keyboard scanning operation. Refer to Appendix N for more information about the workings of the AND statement.

Line 1130 ties the ROM down in memory so you can copy it:

```
1130 POKE 1, PEEK (1) AND 251
```

This statement puts a 0 into bit 2 of location 1, again leaving the other bits untouched. That bit is a switch that causes the character ROM to be brought solidly into memory. In the process, the I/O functions of the machine are put aside. Again, more curious readers can turn to Appendix N for details of how ANDing works.

Lines 1220-1330 transfer the first set of

256 character patterns from the ROM to RAM. That's 2048 bytes. It takes a while, so the program gives some feedback as the transfer progresses. The block is transferred in 256 pieces, eight bytes at a time. Line 1270 performs the actual transfer:

```
1270 :        POKE DS + BYTE,
                   PEEK (SR + BYTE)
```

It peeks at a ROM memory location and then pokes the value it finds there into a RAM memory location.

After each group of eight bytes is transferred, the program prints a period (.) on the screen. To do that, it's necessary to bring the I/O functions back for a moment:

```
1300 :    POKE 1, PEEK(1) OR 4
1310 :    PRINT ".";
1320 :    POKE 1, PEEK(1) AND 251
```

Line 1300 puts a 1 into bit 2 of memory location 1. That switches I/O functions back in. Appendix N also goes into the workings of OR statements. Line 1310 prints the period. Then line 1320 switches I/O back out and the character ROM back in.

When all 2048 bytes have been copied to RAM memory, line 1380 brings I/O back in for keeps. Line 1390 restarts the keyboard scan by putting a 1 into bit 0 of memory location 56334. Finally, lines 1410-1440 tell VIC-II to start using the newly-established RAM memory locations for character patterns:

```
1410 VIC = 53248 : CPTR = VIC+24
1420 PTR = PEEK (CPTR) AND 241
1430 PTR = PTR OR 12
1440 POKE CPTR, PTR
```

These lines may seem a bit cryptic. Let's look into how they work.

Three bits of the register at VIC+24 control the location of the character patterns: bits 1, 2, and 3. Bit 0 of that register does nothing. When you want to change the location of the character patterns, you first clear bits 1, 2, and 3, and then set them to new values.

Line 1420 clears the three bits in question with an ANDing operation. It sets bits 1, 2, and 3 to 0, leaving the other bits unscathed. Then line 1430 sets the bits to new values with an ORing operation.

Blocks of memory containing character patterns must begin at memory locations that are multiples of 2048. In this case, the patterns start at 12288, which is 6 × 2048. When you want to point VIC at a character pattern block, you divide the starting address by 1024 and then use that number to set the bits at VIC+24. 12288 divided by 1024 is 12, so that's the number you use to set the bits.

## 4.9 A LITTLE MODIFICATION

If you haven't done so already, enter and run the program Character ROM to RAM. Nothing seems to happen when the program ends. Press the shift and Commodore logo keys to switch to the second character set—surprise!

You only moved one set of character patterns to RAM. When you switch sets, VIC looks at the next 2K block of RAM for patterns. Since you didn't put patterns into that block, the letters come up as random blotches. Press the shift and Commodore logo keys to get back to the first set.

Let's do some pattern changing. Type in these commands, one by one, and watch how the word READY changes on your screen:

```
POKE 12296, 238
POKE 12297, 204
POKE 12298, 204
POKE 12299, 252
POKE 12300, 204
POKE 12301, 216
POKE 12302, 112
POKE 12303, 0
```

You've changed the pattern used by VIC to put the letter A on the screen. Whenever the code for A appears in screen memory, VIC will use this new pattern to draw the letter.

This command will tell VIC to use the patterns in the built-in character ROM again:

```
POKE 53272, 21
```

Type it in, and watch your A's return to normal. To get them wacky again, use this shortcut command that tells VIC to use the patterns you put into RAM starting at location 12288:

```
POKE 53272, 29
```

| Bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Number codes |
|---|---|---|---|---|---|---|---|---|---|
| Bit value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
| Byte 0 | | | | | | | | | |
| Byte 1 | | | | | | | | | |
| Byte 2 | | | | | | | | | |
| Byte 3 | | | | | | | | | |
| Byte 4 | | | | | | | | | |
| Byte 5 | | | | | | | | | |
| Byte 6 | | | | | | | | | |
| Byte 7 | | | | | | | | | |

Fig. 4-5. A coding form you can use to design characters.

| Bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Number codes |
|---|---|---|---|---|---|---|---|---|---|
| Bit value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
| Byte 0 | | | | ■ | ■ | | | | 24 |
| Byte 1 | | | ■ | ■ | ■ | ■ | | | 60 |
| Byte 2 | | ■ | ■ | | | ■ | ■ | | 102 |
| Byte 3 | | ■ | ■ | ■ | ■ | ■ | ■ | | 126 |
| Byte 4 | | ■ | ■ | | | ■ | ■ | | 102 |
| Byte 5 | | ■ | ■ | | | ■ | ■ | | 102 |
| Byte 6 | | ■ | ■ | | | ■ | ■ | | 102 |
| Byte 7 | | | | | | | | | 0 |

Fig. 4-6. Example of a filled-in character coding form.

## 4.10 DESIGNING CHARACTERS

Figure 4-5 is a coding form you can use to design a character. It's very similar to the coding forms you used with sprites. Eight bytes are used to code a character. Each byte codes the pixel pattern for a row of the character. Each bit in a byte represents a pixel. In any row, the bit values of the pixels to show up are added together to get a number code.

Figures 4-6 and 4-7 are examples that show this coding form in use. In Fig. 4-6, a normal letter A is coded. Figure 4-7 gives codes for an elaborate upside-down A. These codes are the numbers you poked in Section 4.9.

Make some copies of the form in Fig. 4-5. Then design an upside-down version of the letter E. You'll use it in the next section.

| Bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Number codes |
|---|---|---|---|---|---|---|---|---|---|
| Bit value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
| Byte 0 | ■ | ■ | ■ | | ■ | ■ | ■ | | 238 |
| Byte 1 | ■ | ■ | | | ■ | ■ | | | 204 |
| Byte 2 | ■ | ■ | | | ■ | ■ | | | 204 |
| Byte 3 | ■ | ■ | ■ | ■ | ■ | ■ | | | 252 |
| Byte 4 | ■ | ■ | | | ■ | ■ | | | 204 |
| Byte 5 | ■ | ■ | | ■ | ■ | | | | 216 |
| Byte 6 | ■ | ■ | ■ | ■ | | | | | 112 |
| Byte 7 | | | | | | | | | 0 |

Fig. 4-7. Another example of a filled-in character coding form.

## 4.11 PUTTING YOUR MODIFICATIONS INTO POSITION

Appendix D is a list of screen display codes. These are the numbers that are poked into screen memory to tell VIC which character pattern to look up. For example, the screen display code for @ is 0, and the screen display code for A is 1.

Each character pattern uses eight bytes. The patterns are stored in the order of the display codes. First come the eight bytes for @, then the eight bytes for A, and so on. To find the memory location of the first byte of a character's eight pattern bytes, just multiply the character's display code by 8 and add the result to the start of the character memory block.

Here's an example. In the program

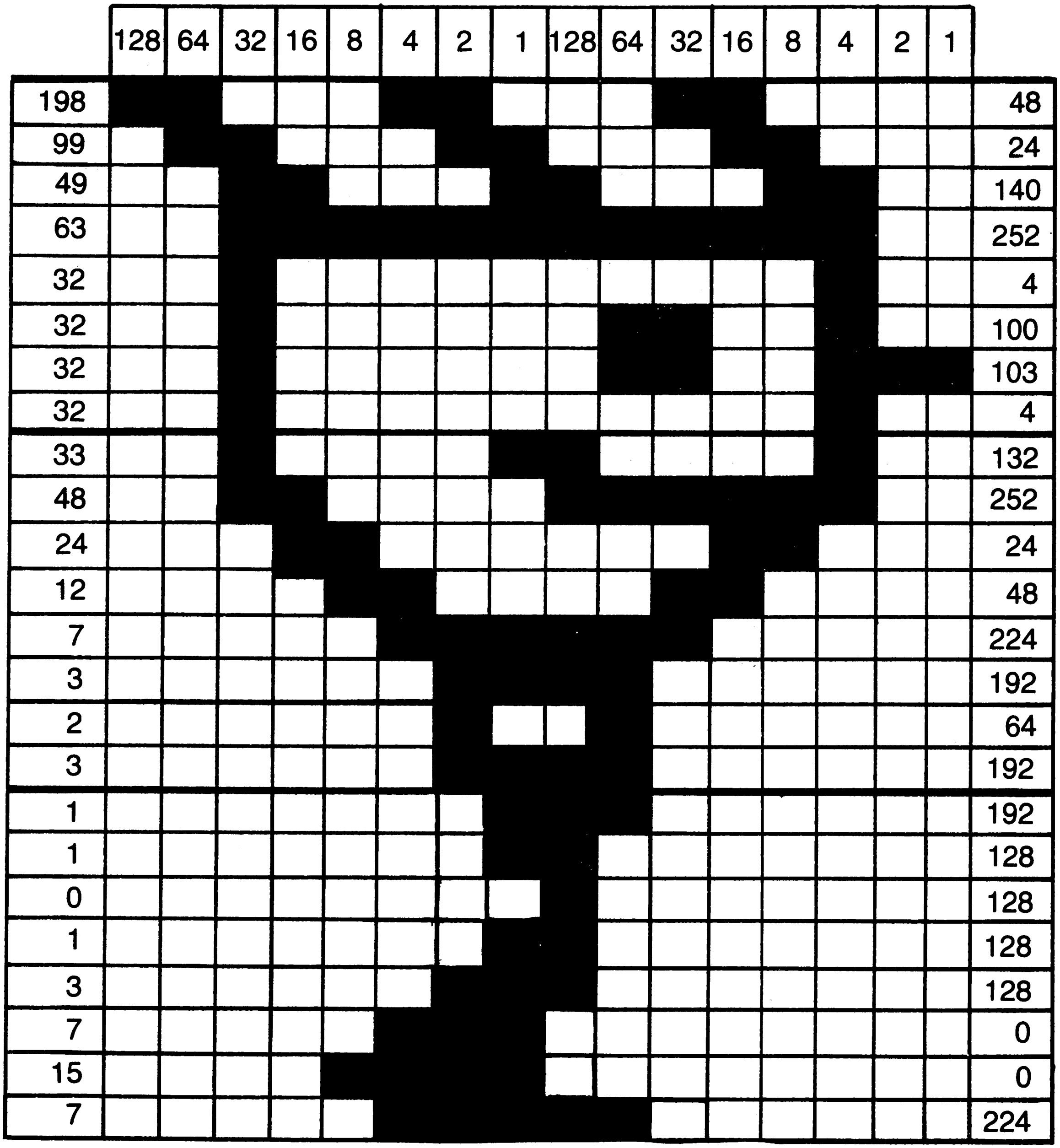



Fig. 4-8. An alien creature drawn on a grid that's two characters wide and three characters high.

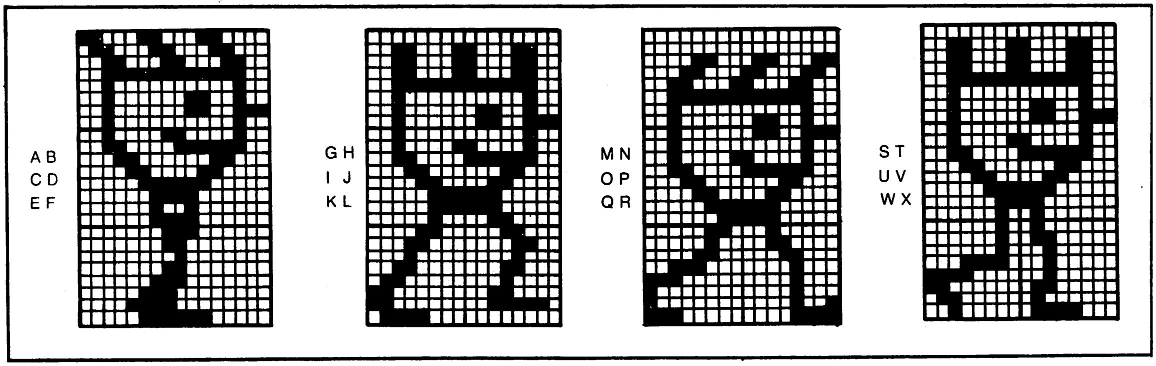



Fig. 4-9. The alien drawn in four positions on 2-by-3 grids, with letters ripe for replacement shown beside each image.

Character ROM to RAM, you moved character memory to a 2K block starting at 12288. To find the first pattern byte for the letter A, you multiply its display code by 8 and add the result to 12288. 1 × 8 is 8, and 12288 + 8 is 12296. So memory locations 12296-12303 (8 bytes) hold the patterns for A. If you look back at Section 4.9, you see that those are the eight locations you poked to change the looks of A.

Let's try this out again. The display code for E is 5. 5 × 8 is 40, and 12288 + 40 is 12328. Run Character ROM to RAM and then poke the eight memory locations beginning at 12328 with the upside-down E codes that you figured out in the last section. Watch the ready prompt as you make each poke.

## 4-12 DESIGNING A SET OF CHARACTERS FOR ANIMATION

You've seen how to change text characters. This gives you the ability to develop all kinds of symbols for games, business applications, foreign languages, and practical jokes. Let's see how you can develop some characters that'll help you pull off some slick animation.

Why would you use characters for animation, when sprites are so easy to use? There are a number of situations where custom character animation has some uses. In some cases, all eight sprites may already be in use. Also, character animation allows some types of color variation without losing horizontal resolution. Finally, you have more leeway in terms of shape and size, since you can put almost any combination of characters together into an image.

Figure 4-8 shows an alien creature drawn on a grid that's two characters wide and three characters high. Along the top, values are shown for each bit position. Along the sides, number codes for the byte rows have been figured. For example, the codes for the character used in the lower right corner of the design are 192, 128, 128, 128, 128, 0, 0, and 224.

Figure 4-9 shows our alien in four positions. Each position is drawn on a 2-character by 3-character grid. Beside each image is a clue to the technique you'll use to get this alien onto the TV screen. You'll insert number codes developed from the images in place of letters A - X. Then you'll just print strings

```
1000 REM *** ALIEN WALKER ***
1010 :
1020 :
1030 REM ** SET UP FEEDBACK DISPLAY
1040 :
1050 PRINT "{CLR}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}LOADING";
1060 :
1070 :
1080 REM ** READ IN THE DATA
1090 :
1100 BASE = 12 * 1024
1110 FOR CHAR = 1 TO 24
1120 :   FOR BYTE = 0 TO 7
1130 :       READ INFO
1140 :       SPOT = BASE + (CHAR * 8)
                    + BYTE
1150 :       POKE SPOT, INFO
1160 :   NEXT BYTE
1170 :   PRINT ".";      :REM FEEDBACK
1180 NEXT CHAR
1190 :
1200 DATA 198,99,49,63,32,32,32,32
1210 DATA 48,24,140,252,4,100,103,4
1220 DATA 33,48,24,12,7,3,2,3
1230 DATA 132,252,24,48,224,192,64,192
1240 DATA 1,1,0,1,3,7,15,7
1250 DATA 192,128,128,128,128,0,0,224
1260 :
1270 DATA 0,49,49,49,63,32,32,32
1280 DATA 0,140,140,140,252,4,100,103
1290 DATA 32,33,48,24,12,7,7,4
1300 DATA 4,132,252,24,48,224,224,48
1310 DATA 12,24,16,48,96,192,192,120
1320 DATA 24,12,8,24,16,48,62,0
1330 :
1340 DATA 0,0,12,24,49,63,32,32
1350 DATA 0,0,99,198,140,252,4,100
1360 DATA 32,32,33,48,24,12,7,3
1370 DATA 103,4,132,252,24,48,224,224
1380 DATA 6,4,12,56,224,128,128,224
1390 DATA 32,48,24,8,8,8,9,15
1400 :
1410 DATA 0,49,49,49,63,32,32,32
1420 DATA 0,140,140,140,252,4,100,103
1430 DATA 32,33,48,24,12,7,3,2
1440 DATA 4,132,252,24,48,224,192,64
1450 DATA 2,2,2,30,240,192,96,32
1460 DATA 64,96,32,32,32,96,64,120
```

```
1470 :
1480 :
1490 REM ** SET UP IMAGE STRINGS
1500 :
1510 IMAGE$(0) = "AB▌█▙CD▌█▙EF▌█◘◘"
1520 IMAGE$(1) = "GH▌█▙IJ▌█▙KL▌█◘◘"
1530 IMAGE$(2) = "MN▌█▙OP▌█▙QR▌█◘◘"
1540 IMAGE$(3) = "ST▌█▙UV▌█▙WX▌█◘◘"
1550 :
1560 :
1570 REM ** CLEAN SCREEN, CENTER, AND
1580 REM    CHANGE THE CHAR DATA PNTRS
1590 :
1600 PRINT "▙▙▙▙▙▙▙▙▙▙▙▙▙▙▙▙▙▙▙▙";
1610 PRINT "▛▛▛▛▛▛▛▛▛▛▛▛▛▛▛▛▛▛▛▛▛";
1620 :
1630 VIC = 53248
1640 POKE (VIC+24),29
1650 :
1660 :
1670 REM ** WALK
1680 :
1690 FOR N = 0 TO 3
1700 :    PRINT IMAGE$(N);
1710 :    FOR DLY = 1 TO 70 : NEXT DLY
1720 :    GET KP$
1730 :    IF KP$ = "" THEN 1750
1740 :       KEY = -1 : N = 3
1750 NEXT N
1760 :
1770 IF (NOT KEY) THEN 1690
1780 :
1790 :
1800 REM ** CLEAN UP SHOP
1810 :
1820 PRINT "▙";
1830 POKE (VIC+24),21
1840 :
1850 END
```

Fig. 4-10. Listing of the program Alien Walker.

made from those letters in combination with some cursor moves, as you did in Fly the Face. Rather than printing 2-by-3 blocks of the real letters, VIC will show 2-by-3 blocks that portray our alien walker.

## 4.13 THE ALIEN WALKER

The program Alien Walker is listed in Fig. 4-10. Let's look at some of its features. You've got four images, each one composed of six redefined character patterns. With eight bytes per pattern, that gives us 24 × 8, or 192, bytes of data to load in. Lines 1100-1180 do the loading:

```
1100 BASE = 12 * 1024
1110 FOR CHAR = 1 TO 24
1120 :    FOR BYTE = 0 TO 7
1130 :         READ INFO
1140 :         SPOT = BASE + (CHAR * 8)
                         + BYTE
1150 :         POKE SPOT, INFO
1160 :    NEXT BYTE
1170 :    PRINT ".";        :REM FEEDBACK
1180 NEXT CHAR
```

Line 1100 sets the base of our character memory at the same convenient location used previously, 12288. Lines 1110 and 1180 set up a loop that will run from character code 1, which stands for A, through character code 24, which stands for X. An inner loop, set up in lines 1120 and 1160, reads in the eight bytes of data for each character and then pokes them into the proper position. Line 1140 figures the proper position by using a formula similar to that used in Section 4.11.

Lines 1200 through 1460 contain pattern codes based on the images from Fig. 4-9. Each line of data contains the codes for one new character definition.

Lines 1510-1540 set up four image strings. Each one is composed of six of our new characters, combined with the cursor moves necessary to display the six characters in a 2-by-3-block. If you don't recognize the graphics icons that represent the various cursor moves in the strings, refer back to the Introduction. Notice that the cursor commands are used in such a way that, after the pieces of the image are drawn, the cursor ends up where it started.

Lines 1600-1610 clear the screen and move the cursor to midscreen. Then Line 1640 tells VIC-II to start getting its character patterns from the 2K block starting at 12288. The line uses the same shortcut seen at the end of Section 4.9. As long as you don't move the location of screen memory, which is coded in bits 4, 5, 6, and 7 of VIC+24, you can use the following formula to set VIC+24 to point at a new character memory block: divide the new starting address by 1024, add that number to 17, and poke it in.

The loop in lines 1690-1750 simply prints the image strings in succession, with a pause between image changes. Lines 1730-1740 are our familiar keypress test. If a key is pressed, the program will end by clearing the screen and resetting the character memory pointer at VIC+24 to point to the built-in character generator ROM.

## 4.14 CHAPTER SUMMARY

Here are some of the topics that have been covered in this chapter:

* The Commodore 64's ability to display 512 built-in characters
* The 1000 screen locations, 1000 bytes of screen memory, and 1000 bytes of color memory
* Poking character codes and colors into screen and color memory
* Putting characters and cursor movements together into strings that can be moved around the screen

* How VIC-II knows where to look for character patterns
* Moving the character ROM patterns into RAM memory
* Designing and installing modifications to the built-in character sets
* Designing and installing a set of characters to be used in an animation cycle

You've been able to scratch the surface of the Commodore 64's wide range of character display abilities. Playful experimentation will help you learn more.

## 4.15 EXERCISES

### 4.15.1 Self Test

Answers will be found in Section 4.15.3.

1. (4.1) There are ____________ different characters built into the Commodore 64's character generator ROM.
2. (4.2) The Commodore 64 normally displays ______________ text lines, each with ____________ characters, which gives __________ screen locations.
3. (4.3) The 64 has __________ complete character sets in ROM.
4. (4.4) Pressing the shift and Commodore logo keys at the same time switches you between the ______________.
5. (4.5.3) Why is the face in Fly the Face drawn surrounded by a ring of spaces?
6. (4.6) Bits 1, 2, and 3 of the register located at VIC+24 tell VIC the location of __________.
7. (4.7) What are two complications involved in copying the contents of the character generator ROM to RAM?
8. (4.10) What would a character pattern look like if its eight number codes were all 255?

### 4.15.2 Programming Exercises

1. Change the program Fly the Face so another design flies around the screen.
2. Change the program Character ROM to RAM so the characters come out upside-down.
3. Change the program Alien Walker so that three aliens, all alike, are walking across the screen.

### 4.15.3 Answers to Self Test

1. 512
2. 25; 40; 1000
3. two
4. two character sets
5. so it'll erase any traces of itself as it moves
6. the character patterns
7. (1) the ROM floats around at different memory addresses
   (2) when it's tied down, input/output operations are disabled
8. a solid square

### 4.15.4 Possible Solutions to Programming Exercises

These solutions are based on adding or changing lines in the programs mentioned in the exercises.

1. Load in the program Fly the Face. Then type in these lines:

```
1000 REM *** FLY THE FIGURE ***
1060 F$(2) = "               "
1070 F$(3) = "               "
1080 F$(4) = "               "
1090 F$(5) = "               "
1100 F$(6) = "               "
```

2. Load in the program Char ROM to RAM. Then type in these lines:

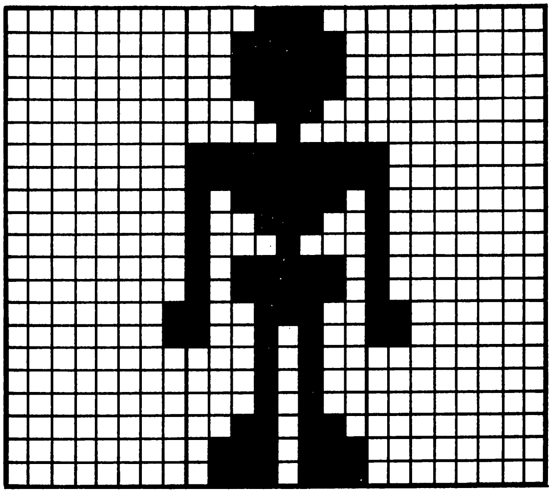
```
1000 REM *** UPSIDE-DOWN ROM ***
1270 :        POKE DS + (7 - BYTE),
                   PEEK (SR + BYTE)
```

3. Load in the program Alien Walker. Then type in these lines:

```
1000 REM *** 3 ALIEN WALKERS ***
1610 PRINT "";
1701 :     PRINT "";
1702 :    PRINT IMAGES$(N);
1703 :     PRINT "";
1704 :    PRINT IMAGES$(N);
1705 :     PRINT "";
1706 :
1710 :    FOR DLY = 1 TO 60 : NEXT DLY
```

# Chapter 5

# Bit Mapped Graphics



So far, you've explored two aspects of Commodore 64 graphics: sprites and characters. Both these graphics entities let you play with collections of pixels. Is there a way to draw large, detailed pictures by controlling individual pixels? You bet. It's called bit mapped graphics.

In this chapter, you'll learn how to set up bit map mode. You'll turn individual pixels on and off, and see how to set their color. I'll give you a machine-language routine that will speed up one tedious aspect of bit mapping. Finally, you'll build a simple electronic doodling program.

## 5.1 SIXTY FOUR THOUSAND PIXELS

Time to do a little arithmetic. Consider the Commodore 64's text display. There are 25 lines, each with 40 characters. Each character is 8 pixels wide, and 8 pixels high. That gives 8 × 40, or 320, pixels across the screen and 8 × 25, or 200, pixels from top to bottom. 320 pixels across the screen multiplied by 200 from top to bottom gives a grand total of 64,000 pixels.

In bit map mode, you control each one of these pixels with a bit. That's where the name bit mapping comes from. Since there are 8 bits stored in a byte, you can divide 64,000 by 8 and find you need 8,000 bytes to control a screen filled with 64,000 pixels. Those 8,000 bytes form the bit map. Where can you store such a large bit map?

## 5.2 STORING THE BIT MAP

Back in Section 4.6, I mentioned that the VIC-II graphics chip looks at 16K of memory at a time. 8000 bytes is almost 8K, or half of a

16K block of memory. An 8000-byte bit map can live in either the first or second half of the current VIC-II 16K bank.

When you're working with BASIC, VIC normally looks at the 16K memory block from locations 0 through 16383. The first few thousand memory locations in that block are vital real estate for BASIC; it won't give them up easily. So the bit map goes in the second half of the block, starting at memory location 8192. Bit 3 of the register at VIC+24 (memory location 53272) controls the location of the bit map. If there's a 0 stored there, it goes in the first half of the current 16K VIC-II bank. Storing a 1 at bit 3 of VIC+24 puts the bit map in the second half of the 16K bank, which is what is normally done when using a bit map from BASIC.

This BASIC command will store a 0 at bit 3 of VIC+24 (53272):

```
POKE 53272, PEEK(53272) AND 247
```

And this command will store a 1 at that position:

```
POKE 53272, PEEK(53272) OR 8
```

## 5.3 TURNING BIT MAP MODE ON AND OFF

Bit 5 of the register at VIC+17 (memory location 53265) controls bit map mode. Storing a 1 at that location turns bit map mode on and storing a 0 turns it off. Here's the BASIC command to turn bit mapping on:

```
POKE 53265, PEEK(53265) OR 32
```

And here's the command that turns it off, bringing back a normal text display:

```
POKE 53265, PEEK(53265) AND 223
```

## 5.4 A SHORT DISCLAIMER

BASIC is a fine computer language, with advantages and disadvantages. Programs can be put together and debugged fairly quickly, but they run slowly when compared to programs in many other languages. Of course, in many applications, BASIC's speed problems aren't noticeable, and its ease of use is a welcome relief.

The speed problem shows up in programs where there's a lot of fairly repetitive activities. Bit mapped graphics, where 64,000 bits are waiting for instructions, is one of the areas where BASIC's lethargy shows.

How can you speed up bit mapped programs written in BASIC? The best technique is intelligent program design. For example, many calculations can be done just once, with the results stored in data tables, rather than being repeated over and over. Skills you pick up trying to apply intelligent design techniques carry over to other computer languages.

A popular technique, yet one I'm not too fond of, involves squashing code together, with as many statements on a line as space permits. I find that the time savings from this technique are minimal, and the problems of debugging such programs are depressing.

A third alternative involves taking critical operations and coding them in machine language. Short of rewriting an entire program in machine language, this technique leads to some of the biggest time savings possible. You'll see an example of it later in this chapter.

## 5.5 ONE LAST DETAIL: COLOR

Before we get to an example program,

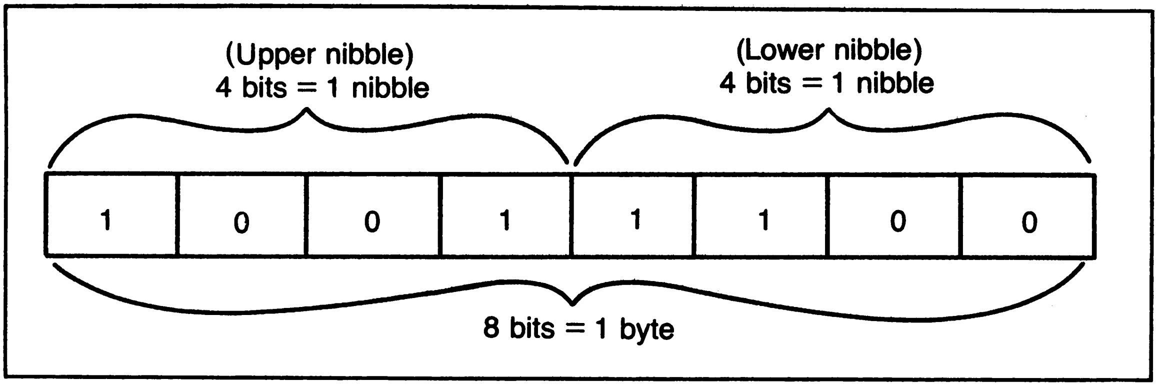



Fig. 5-1. The relationships between bits, bytes, and nibbles.

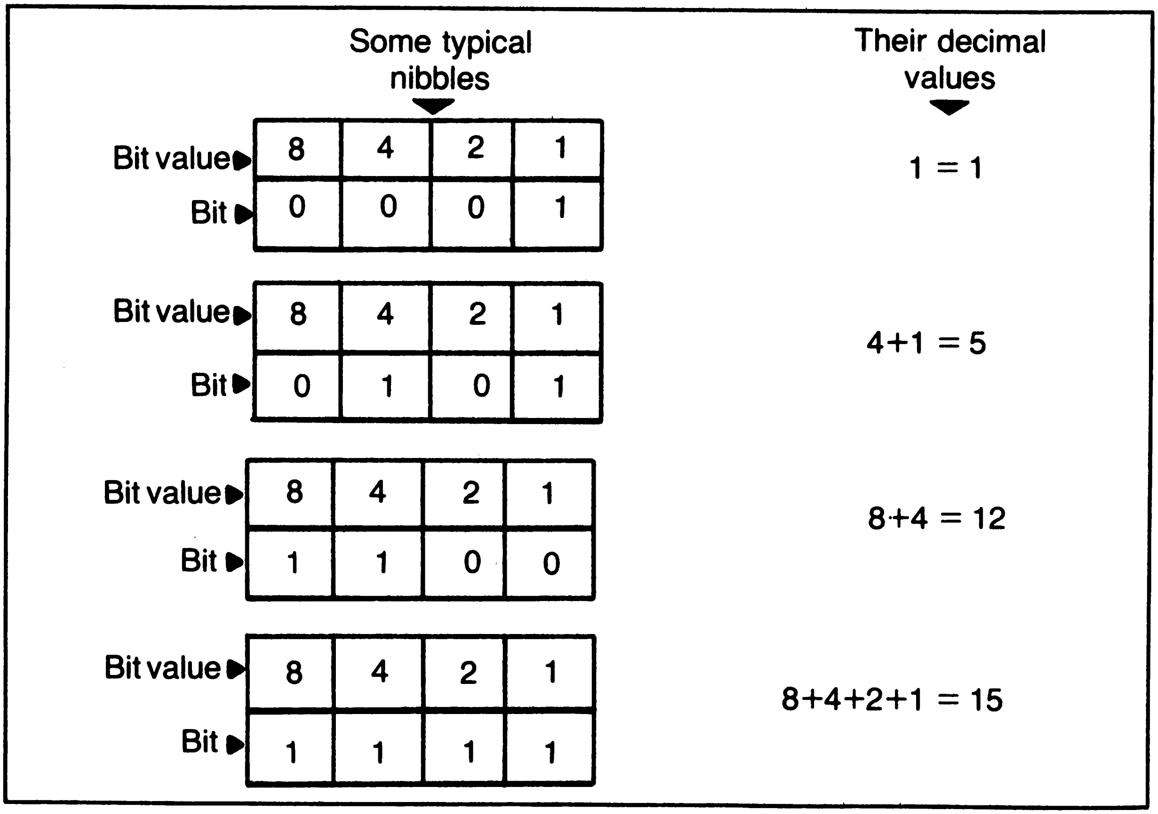



Fig. 5-2. Some typical nibbles, with the corresponding base 10 values.

there's one last detail to discuss: color. How does VIC-II decide on a color for each of the 64,000 pixels?

With normal bit mapped graphics, pixels in each 8-by-8-pixel section of the screen, an area the size of a character, have a choice of two colors. The fact that these areas are the same size as a character in text display mode leads to a clever storage idea. The two color codes for each 8-by-8 area are stored in the 1,000 locations of screen memory. That's the same area used in text display mode to hold screen display codes.

Computer people like cute names. 8 bits is known as a *byte*, and 4 bits is called a ***nibble***. See Fig. 5-1. A nibble can store values between 0 and 15. See Fig. 5-2. In bit map mode, the upper 4 bits, or nibble, of each screen

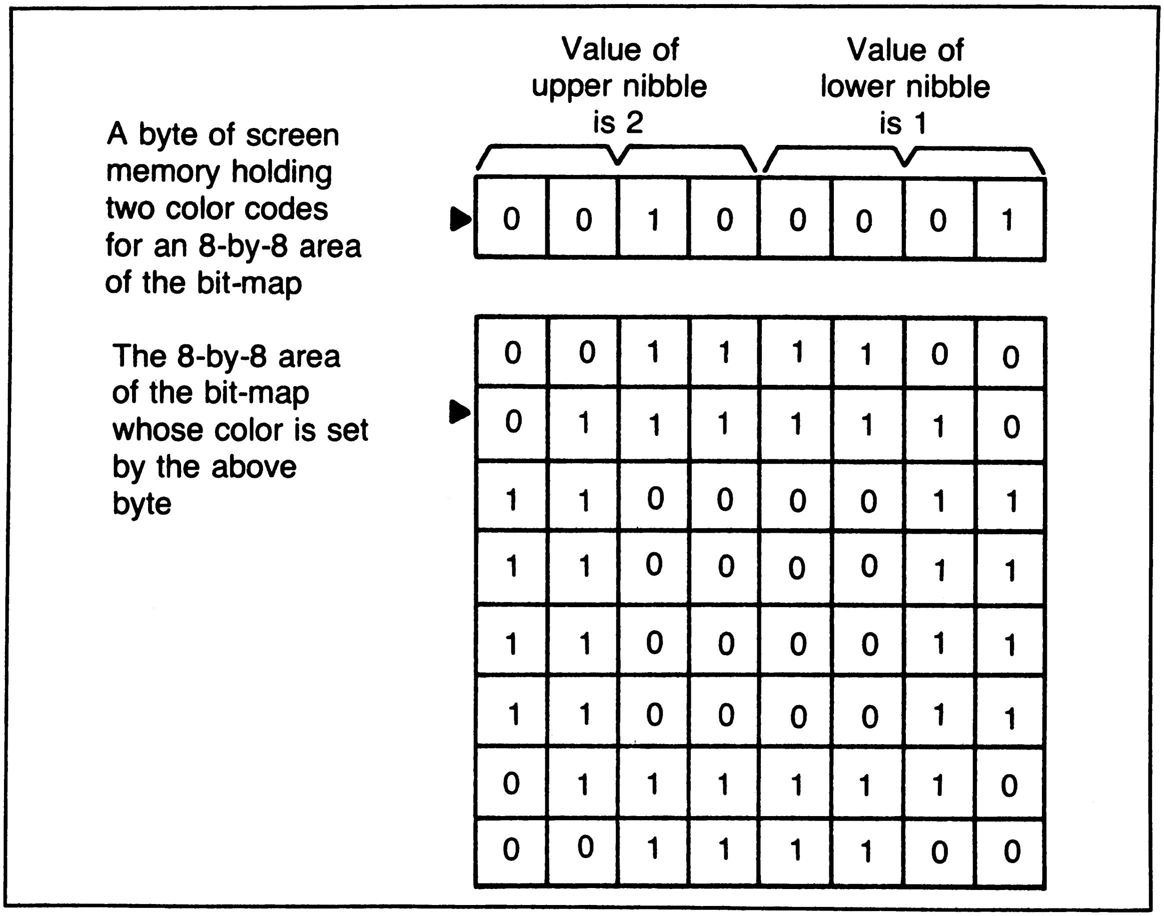



Fig. 5-3. An example of an 8-by-8-bit area of the bit map whose color is controlled by a byte of screen memory. The value in the byte's upper nibble codes the color for bits in the map set to 1, while the value in the lower nibble codes for bits in the map set to 0.

memory location hold the color code for any bit set to 1 in the 8-by-8-bit area controlled by that memory location. The lower nibble of the screen memory location holds the color code for bits set to 0. Take a look at Fig. 5-3 for an example. There's a little formula to help you figure out what number to poke into this screen memory for a given pair of colors: take the color code for the 1 bits, multiply it by 16, and then add the color code for the 0 bits. For example, if you wanted 1 bits to come out red (color code 2), and 0 bits to come out black (color code 0), you would calculate that $(2 \times 16) + 0 = 32$, and you'd poke into screen memory.

## 5.6 AN EXAMPLE OF BIT MAPPED GRAPHICS

So much for your preliminary dose of bit mapping theory. It's time for some action. Type in the program listed in Fig. 5-4, Random Draw. Save it to tape or disk and then run it. Watch it for a couple of minutes, and then let it run unattended for 5 or 10 minutes. Take a last good look, and press the spacebar to end it.

### 5.6.1 Setting Up for the Bit Map Mode

Let's examine the program, and see if you can understand what you saw happen on the screen. Line 1100 uses the command dis-

```
1000 REM *** RANDOM DRAW ***
1010 :
1020 :
1030 REM ** SET UP FOR BIT-MAP MODE
1040 :
1050 VIC = 53248
1060 BASE = 8192    :REM BIT MAP START
1070 BLOC = VIC+24 :REM LOCATES BIT MAP
1080 BSET = VIC+17 :REM TURNS ON BMM
1090 :
1100 POKE BLOC, PEEK(BLOC) OR 8
1110 POKE BSET, PEEK(BSET) OR 32
1120 :
1130 :
1140 REM ** CLEAR THE BIT MAP
1150 :
1160 FOR SPOT = BASE TO BASE + 7999
1170 :    POKE SPOT, 0
1180 NEXT SPOT
1190 :
1200 :
1210 REM ** SEED THE RANDOM FUNCTION
1220 REM    WITH A RANDOM NUMBER
1230 :
1240 DUMMY = RND (-RND(0))
1250 :
1260 :
```

the new version, and then run it. Whoosh! You can see why hot programmers eventually turn to machine language whenever real speed's needed.

A brief explanation of the new lines: lines 1146-1156 poke the machine language subroutine into a portion of memory that most BASIC programs won't bump into. Lines 1163-1176 contain the 26 bytes of data that make up the little whizzer. Finally, line 1193 calls the newly installed machine language subroutine into action with a SYS command. It's like jumping to a BASIC subroutine. When the machine language routine finishes, it pops control back to BASIC, and BASIC just carries on with the next statement.

You can use this routine in any bit map program that uses locations 8192-16191 as the bit map area. If you want to clear a bit map that starts at another area, just divide the starting address of the bit map by 256 and type the new value in place of the 32 at the end of line 1163.

## 5.8 LOCATING A PIXEL'S BYTE AND BIT

Let's learn how to gain more control over individual pixels in bit map mode. You need to find a way to locate the byte and bit that control an individual pixel.

First, you need a model of the screen display. Take a look at Fig. 5-6. Each pixel has a horizontal position, H, with values from 0 through 319. Each pixel also has a vertical

```
1000 REM *** FAST RANDOM DRAW ***
1140 REM ** LOAD FAST M/L BIT MAP CLEAR
1143 :
1146 FOR N = 21248 TO 21273
1150 :   READ MLDTA
1153 :   POKE N, MLDTA
1156 NEXT N
1160 :
1163 DATA 169, 0, 133, 251, 169, 32
1166 DATA 133, 252, 162, 32, 160, 0
1170 DATA 152, 145, 251, 200, 208, 251
1173 DATA 202, 240, 4, 230, 252, 208
1176 DATA 244, 96
1180 :
1183 :
1186 REM ** CLEAR THE BIT MAP
1190 :
1193 SYS 21248
1196 :
```

Fig. 5-5. Changes and additions that turn Random Draw into Fast Random Draw.

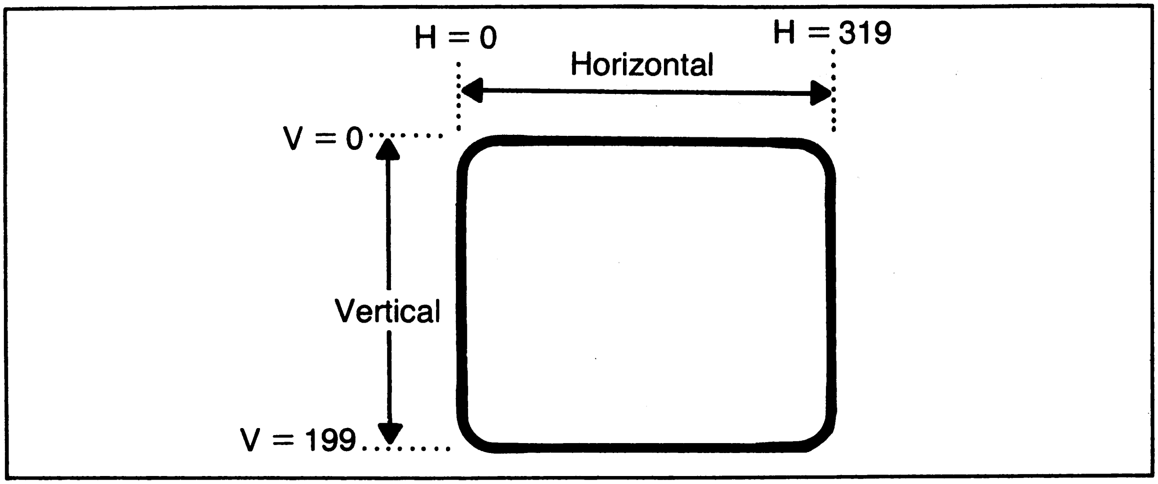



Fig. 5-6. You can give each pixel on the bit map a horizontal position from 0 through 319 and a vertical position from 0 through 199.

position, V, with values from 0 through 199. For example, a pixel in the upper left corner has H = 0 and V = 0. A pixel in the lower right corner has H = 319 and V = 199.

It would be wonderful if the bytes in the bit map had a simple correspondence to Fig. 5-6. Unfortunately, that's not the case. The bytes in the bit map correspond to the screen in a pattern that suggests bit mapping's close kinship to text display.

Take a look at Fig. 5-7. It shows how the bit map bytes are set up. Groups of 8 consecutive bytes form a block the size of a character. Similar to the text screen, these 8-byte-high areas are arranged in 40 columns and 25 rows. Trying to determine which bit of which byte controls a pixel, given that pixel's horizontal and vertical position, looks like an arduous task.

It's actually not too tough. If you go slowly, and keep referring back to Figs. 5-6 and 5-7, the following formula derivations may make sense. Remember, H and V refer to a pixel's horizontal and vertical positions respectively.

Let's start with vertical information. Since a row is 8 vertical positions high, this formula gives us the row a pixel's in:

```
ROW = INT(V/8)
```

There are 320 bytes per row, so a row's offset in bytes from the base of the bit map is:

```
RBF = ROW * 320
```

The AND function is a convenient way of finding remainders when you're dividing by a power of 2: Simply AND the original number with the divisor minus 1. Finding the remainder of the vertical position divided by 8 will tell you which of the 8 lines in a row you want:

```
LINE = (V AND 7)
```

You can combine these results and form a total vertical byte offset for your pixel:

```
VBF = INT(V/8) * 320 + (V AND 7)
```

Now you need to work with the pixel's horizontal position. There are 8 horizontal positions per column, so the column can be figured this way:

```
COLUMN = INT(H/8)
```

| | Column 0 | Column 1 | Column 2 | | Column 39 |
|---|---|---|---|---|---|
| Row 0 | Byte 0 | Byte 8 | Byte 16 | | Byte 312 |
| | Byte 1 | Byte 9 | Byte 17 | | Byte 313 |
| | Byte 2 | Byte 10 | Byte 18 | | Byte 314 |
| | Byte 3 | Byte 11 | Byte 19 | | Byte 315 |
| | Byte 4 | Byte 12 | Byte 20 | • • • | Byte 316 |
| | Byte 5 | Byte 13 | Byte 21 | | Byte 317 |
| | Byte 6 | Byte 14 | Byte 22 | | Byte 318 |
| | Byte 7 | Byte 15 | Byte 23 | | Byte 319 |
| Row 1 | Byte 320 | Byte 328 | Byte 336 | | Byte 632 |
| | Byte 321 | Byte 329 | Byte 337 | | Byte 633 |
| | Byte 322 | Byte 330 | Byte 338 | | Byte 634 |
| | Byte 323 | Byte 331 | Byte 339 | • • • | Byte 635 |
| | Byte 324 | Byte 332 | Byte 340 | | Byte 636 |
| | Byte 325 | Byte 333 | Byte 341 | | Byte 637 |
| | Byte 326 | Byte 334 | Byte 342 | | Byte 638 |
| | Byte 327 | Byte 335 | Byte 343 | | Byte 639 |
| | ⋮ | ⋮ | ⋮ | | ⋮ |
| Row 24 | Byte 7680 | Byte 7688 | Byte 7696 | | Byte 7992 |
| | Byte 7681 | Byte 7689 | Byte 7697 | | Byte 7993 |
| | Byte 7682 | Byte 7690 | Byte 7698 | | Byte 7994 |
| | Byte 7683 | Byte 7691 | Byte 7699 | • • • | Byte 7995 |
| | Byte 7684 | Byte 7692 | Byte 7700 | | Byte 7996 |
| | Byte 7685 | Byte 7693 | Byte 7701 | | Byte 7997 |
| | Byte 7686 | Byte 7694 | Byte 7702 | | Byte 7998 |
| | Byte 7687 | Byte 7695 | Byte 7703 | | Byte 7999 |

Fig. 5-7. How the bit map bytes are set up. Notice the close relationship to the Commodore 64's text display.

Notice how there's a jump of 8 bytes as you move from column to column. Now figure your total horizontal byte offset factor:

```
HBF = INT(H/8) * 8
```

Now you can add the vertical and horizontal byte offsets to the start of the bit map to get to your target byte:

```
BYTE = BASE + VBF + HBF
```

You've got the byte. You need to find the bit. There are 8 pixels to a column. You need to know how many pixels are left after you've gone through all the full columns. Again, you use an AND operation to find a remainder:

```
PXL = (H AND 7)
```

Since bits in a byte are numbered from right to left, and your horizontal pixel positions go from left to right, you have to adjust this with a little reversal operation:

```
BIT = 7 - (H AND 7)
```

So now you've got formulas to find a bit mapped pixel's byte and bit. Let's do something with them.

## 5.9 TURNING PIXELS ON AND OFF

Once you've found a pixel's byte and bit with the formulas developed in Section 5.8, the following statement will set the bit to 1:

```
POKE BYTE, PEEK(BYTE) OR (2↑BIT)
```

Remember, that will tell the pixel to take on the color whose code is in the upper nibble of a byte of screen memory.

This command will set a pixel's bit to 0:

```
POKE BYTE, PEEK(BYTE) AND (255 - 2↑BIT)
```

The pixel will then take on the color whose code is in the lower nibble of the appropriate screen memory byte.

## 5.10 THE ELECTRONIC DOODLER

Now that you can turn individual pixels on and off, let's play with an electronic doodling program. Figure 5-8 is a listing of the program Sketch. Type it in, save it, and then run it.

A dot-sized pen will appear in the center of the screen. You can move the pen in any of

```
1000 REM *** SKETCH ***
1010 :
1020 :
1030 REM ** INITIAL SET-UP
1040 :
1050 PRINT "♥";      :REM CLEAR SCREEN
1060 POKE 650, 128  :REM ALL KEYS REPEAT
1070 :
1080 BASE = 8192    :REM BIT MAP START
1090 VIC = 53248    :REM GRAPHICS CHIP
1100 BLOC = VIC+24  :REM SETS BASE
1110 BSET = VIC+17  :REM SETS BMM
1120 :
1130 :
```

```
1140 REM ** LOAD SPEEDY M/L CLEAR
1150 :
1160 FOR N = 21248 TO 21273
1170 :    READ MLDTA
1180 :    POKE N, MLDTA
1190 NEXT N
1200 :
1210 DATA 169, 0, 133, 251, 169, 32
1220 DATA 133, 252, 162, 32, 160, 0
1230 DATA 152, 145, 251, 200, 208, 251
1240 DATA 202, 240, 4, 230, 252, 208
1250 DATA 244, 96
1260 :
1270 :
1280 REM ** SET FOR BIT-MAP MODE, CLEAR
1290 REM    BIT MAP, SET COLOR COMBO
1300 :
1310 POKE BLOC, PEEK(BLOC) OR 8
1320 POKE BSET, PEEK(BSET) OR 32
1330 :
1340 SYS 21248   :REM M/L BIT MAP CLEAR
1350 :
1360 FOR HUEMAP = 1024 TO 2023
1370 :    POKE HUEMAP, 3
1380 NEXT HUEMAP
1390 :
1400 :
1410 REM ** INITIALIZE H AND V
1420 :
1430 H = 160 : V = 100
1440 :
1450 :
1460 REM ** DRAW THE DOT AT H,V
1470 :
1480 VBF = INT (V/8) * 320 + (V AND 7)
1490 HBF = INT (H/8) * 8
1500 BIT = 7 - (H AND 7)
1510 BYTE = BASE + VBF + HBF
1520 POKE BYTE, PEEK(BYTE) OR (2↑BIT)
1530 :
1540 :
1550 REM ** GET KEYPRESS COMMAND
1560 :
1570 GET KP$
1580 IF KP$ = "" THEN 1570
1590 :
1600 :
1610 REM ** DEAL WITH KEYPRESS
```

```
1620 :
1630 IF KP$ = " " THEN 1850
1640 IF KP$ = "S" THEN SYS 21248 :
                       GOTO 1430
1650 :
1660 IF KP$ = "W" THEN             V=V-1
1670 IF KP$ = "E" THEN H=H+1  :  V=V-1
1680 IF KP$ = "D" THEN H=H+1
1690 IF KP$ = "C" THEN H=H+1  :  V=V+1
1700 IF KP$ = "X" THEN             V=V+1
1710 IF KP$ = "Z" THEN H=H-1  :  V=V+1
1720 IF KP$ = "A" THEN H=H-1
1730 IF KP$ = "Q" THEN H=H-1  :  V=V-1
1740 :
1750 IF V < 0   THEN V = 0
1760 IF V > 199 THEN V = 199
1770 IF H < 0   THEN H = 0
1780 IF H > 319 THEN H = 319
1790 :
1800 GOTO 1480
1810 :
1820 :
1830 REM ** WRAP IT UP
1840 :
1850 POKE BSET, PEEK(BSET) AND 223
1860 POKE BLOC, 21
1870 :
1880 PRINT "♥";
1890 :
1900 END
```

Fig. 5-8. Listing of the program Sketch.

the eight compass directions by pressing W, E, D, C, X, Z, A, or Q. Figure 5-9 shows the layout of these keys, and the direction each one will send the pen. Pressing the S key erases your drawing and places the pen back in the center of the screen—there's no need to turn your TV set upside down and shake it.

When you finish playing, press the spacebar to stop the program. Then settle down for a little explanation of how it works.

### 5.10.1 Setting Up the Sketch Pad

Lines 1000-1340 should look pretty familiar. You clear the screen and then set the keyboard so all the keys will repeat when held down long enough. Lines 1160-1190 load the fast machine language routine to clear the bit map. Then lines 1310-1340 set up bit mapping and use the machine language clearing routine.

Lines 1360-1380 fill screen memory with a color scheme for the bit map. Bits set to 0 will be cyan, and bits set to 1 will be black. Since line 1340 filled the bit map with 0's, the screen turns cyan.

You'll store the pen's current horizontal and vertical positions in the variables H and V. Line 1430 sets these variables so the pen is centered on the screen. Whenever the S key gets pressed, the program will pop back up to this line.

### 5.10.2 Drawing

Lines 1480-1520 use the formulas developed in Sections 5.8 and 5.9 to turn on the bit corresponding to the current pen position. Putting a 1 in that bit causes the pixel at the pen position to turn black.

### 5.10.3 Getting and Following Orders

Lines 1570-1580 wait for the sketcher to press a key. Then lines 1630-1800 figure out what to do with the keypress. A space sends the program to line 1850, where it cleans up shop and ends. Pressing S clears the bit map and then puts the pen back in the center by jumping back to line 1430.

Lines 1660-1730 change the pen's posi-

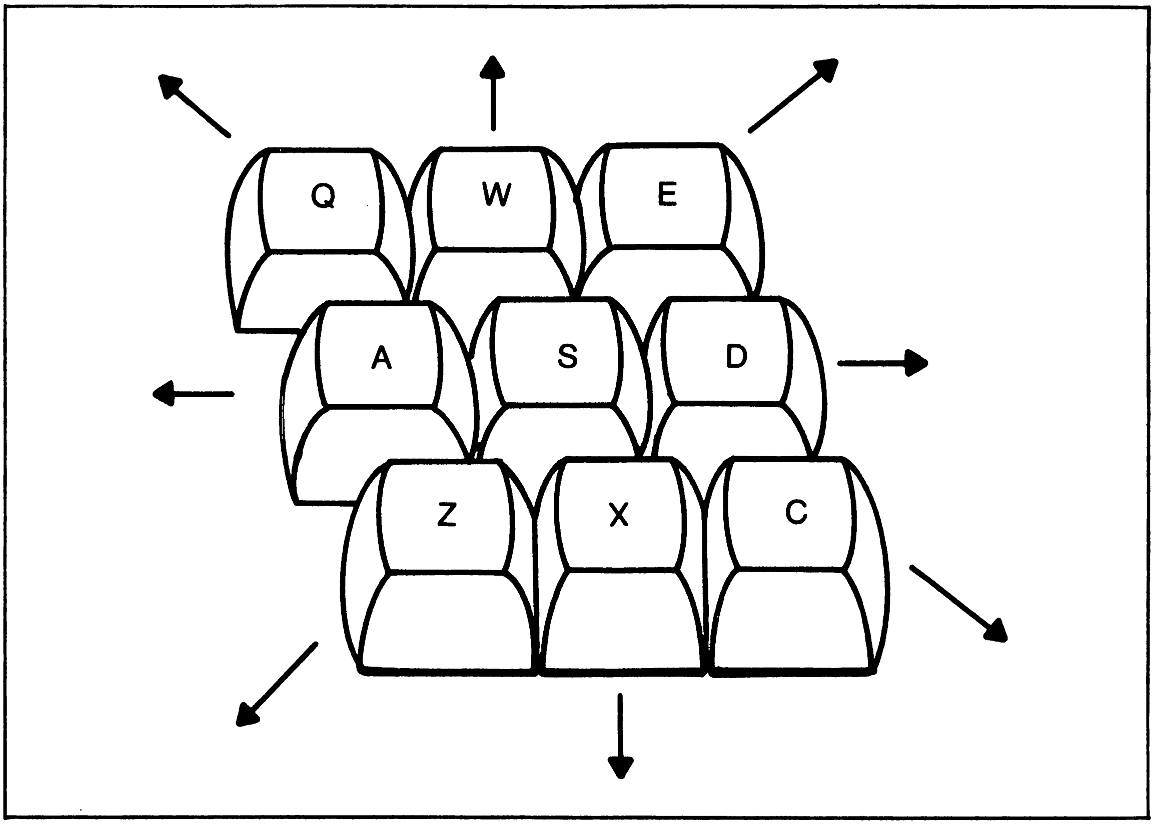



Fig. 5-9. Layout of the control keys used in Sketch, and the direction each one will send the pen.

tion if one of the eight movement keys has been pressed. Referring to Figs. 5-6 and 5-9 should help you understand these lines.

Lines 1750-1780 check to make sure the pen doesn't fall off the screen. If a keypress tries to push the pen off, these four lines pull it back on. Finally, line 1800 loops on back to draw the pen's dot on the screen.

Notice that any keys not included in the program's command set will be ignored. Also, the clean structure of this section makes it easy to add new commands.

Lines 1850-1900 are a straightforward end to the program. They reset the display to text mode, and clear the screen. It's the same way you ended Random Draw.

Take some time to play with Sketch. See what interesting features you can add to it.

## 5.11 CHAPTER SUMMARY

This chapter has introduced some of the techniques of bit mapped graphics. More specifically, you should now know:

* How to represent 64,000 screen pixels in an 8,000-byte bit map
* Where you usually store the bit map when working in BASIC, and how to tell VIC-II the location
* How to turn bit map mode on and off via the register at VIC+17
* Why really fast bit mapped graphics work often requires the use of machine language routines
* How the screen memory is used to provide color information for pixels in bit mapped mode
* Some of the ways random numbers can be used to create bit mapped designs
* How to find the byte and bit that control an individual pixel in bit map mode
* How to set an individual pixel to either of the two colors available in its block

At this point, you've been introduced to the Commodore's three main graphics capabilities: sprites, character graphics, and bit mapping. In the next chapter, you'll look at some odds and ends from the Commodore 64's set of graphics tricks.

## 5.12 EXERCISES

### 5.12.1 Self Test

Answers can be found in Section 5.12.3

1. (5.1) Bit mapping lets you control __________ screen pixels with an __________ -byte bit map.
2. (5.2) When using BASIC, the bit map is usually located in the __________ half of the first 16K of memory.
3. (5.3) Bit 5 of the register at __________ (memory location 53265) turns bit map mode on and off.
4. (5.4) Why are machine language routines often used with bit mapped graphics?
5. (5.5) In bit map mode, the two nibbles of a byte of screen memory are used to __________.
6. (5.6.3) Which lines of Random Draw set the colors for the bit map?
7. (5.7) The __________ command lets you jump to a machine language subroutine from BASIC.
8. (5.8) The relationship between bytes in

the bit map and pixels on the screen is __________.

9. (5.9) Setting a bit in the bit map to 1 gives the related pixel the color that's in the __________ nibble of a byte in screen memory.
10. (5.10) What would happen to the program Sketch if line 1640 jumped to line 1480 rather than to line 1430?

### 5.12.2 Programming Exercises

1. Change the program Random Draw so it draws colored vertical lines at random on a black screen.
2. Change Sketch so that it makes lines that are twice as wide. Warning: the program will probably run slowly. This is a case where a new program design and/or machine language routines would be warranted after you get the slow version running.
3. This one may seem tough, but it's really not too bad. You can use sprites with bit map mode. Design a sprite that looks like a pen, pencil, or brush. Then change the program Sketch so it looks as if your sprite is drawing the lines.

### 5.12.3 Answers to Self Test

Answers may vary, especially with questions #4 and #8.

1. 64,000; 8,000
2. Second
3. VIC+17
4. Speed
5. Set colors for an 8-by-8 pixel area of the screen display
6. Lines 1300-1320
7. SYS
8. Arcane and strange, yet often useful
9. Upper
10. When a drawing was erased, the pen would start up where it left off, rather than at the center of the screen

### 5.12.4 Possible Solutions to Programming Exercises

Once again, these solutions are based on adding or changing lines in the programs mentioned in the exercises.

1. Load in the program Random Draw. Then type in these lines:

```
1000 REM *** VERTICAL RANDOM DRAW ***
1310 :  POKE SPOT, INT (RND(1)*16) * 16
1380 SPOT = INT(RND(1)*1000) * 8 + BASE
1385 PATTERN = 56
1390 :
1395 FOR BYTE = 0 TO 7
1400 :    POKE SPOT + BYTE, PATTERN
1405 NEXT BYTE
```

2. Load in the program Sketch. Then type in these lines:

```
1000 REM *** FAT SKETCH ***
1473 FOR X = H TO (H + 1)
1476 :    FOR Y = V TO (V + 1)
1480 :    VBF = INT (Y/8) * 320 +
                            (Y AND 7)
1490 :    HBF = INT (X/8) * 8
1500 :    BIT = 7 - (X AND 7)
1510 :    BYTE = BASE + VBF + HBF
1520 :    POKE BYTE, PEEK(BYTE) OR
                               (2↑BIT)
1523 :    NEXT Y
1526 NEXT X
```

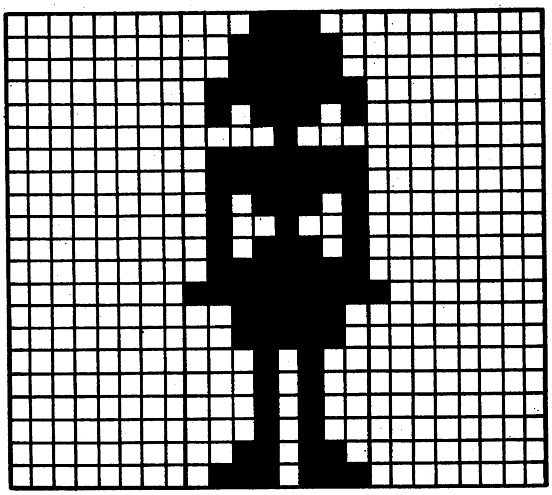
```
1760 IF V > 198 THEN V = 198
1780 IF H > 318 THEN H = 318
1800 GOTO 1473
```

3. Load in the program Sketch. Then type in these lines:

```
1000 REM *** PENCIL SKETCH ***
1251 :
1252 :
1253 REM ** LOAD THE SPRITE DATA
1254 :
1255 FOR N = 896 TO 958
1256 :   READ SPDTA
1257 :   POKE N, SPDTA
1258 NEXT N
1259 :
1260 DATA   0,   1, 224,   0,   3,  48
1261 DATA   0,   6,  24,   0,  12,  12
1262 DATA   0,  24,   6,   0,  48,   2
1263 DATA   0,  96,   6,   0, 192,  12
1264 DATA   1, 128,  24,   3,   0,  48
1265 DATA   6,   0,  96,   7,   0, 192
1266 DATA  13, 129, 128,  24, 195,   0
1267 DATA  16, 102,   0,  16,  60,   0
1268 DATA  48,  48,   0,  56, 240,   0
1269 DATA 127, 128,   0, 120,   0,   0
1270 DATA 192,   0,   0
1271 :
1272 :
1391 :
1392 REM ** SET THE SPRITE CONTROLS
1393 :
1394 POKE 2040, 14  :REM SET #0'S PNTR
1395 POKE VIC+39, 0 :REM PAINT IT BLACK
1396 POKE VIC+29, 1 :REM EXPAND HORZNTL
1397 POKE VIC+23, 1 :REM EXPAND VERTCAL
1398 POKE VIC, 184  :REM INIT HORZ POS
1399 POKE VIC+1,109 :REM INIT VERT POS
1400 POKE VIC+21,1  :REM SPRITE #0 ON
1401 :
1402 :
1531 REM ** MOVE THE SPRITE
1532 :
1533 SH = H + 24 : SV = V + 9
1534 RS = (SH > 255)
1535 POKE VIC, SH + (RS * 256)
1536 POKE VIC+16, -RS
1537 POKE VIC+1, SV
1538 :
1539 :
1871 POKE VIC+21, 0  :REM SPRITE #0 OFF
1872 POKE VIC+23, 0  :REM EXPANSION OFF
1873 POKE VIC+29, 0
```

## Chapter 6

# More Graphics Tricks



This chapter will be a little different from the previous five. I'll touch lightly on a larger number of graphics features. The program discussions will be slimmed down so more topics can be covered.

Here are the areas you'll be looking at: sliding sprites over and under background graphics, putting text onto a bit mapped display, flying a sprite with a joystick, detecting collisions between sprites and other graphics objects, two more color modes for character graphics, and multicolor bit mapping. There's lots to deal with, so let's dive right in . . .

### 6.1 SPRITE TO BACKGROUND PRIORITY

Back in Chapter 3, Section 3.5 sprite to sprite display priorities were discussed. When two or more sprites overlap on the screen, sprites with lower numbers have higher display priorities. For example, sprite #3 will appear in front of sprite #5.

There is a register at VIC+27 (memory location 53275) that controls sprite to background priorities. Background means any display that's not part of a sprite: characters and bit mapped images. Each sprite has a bit allocated to it in the register at VIC+27. Bit 0 controls sprite #0; bit 1 controls sprite #1, and so on.

If a sprite's bit is set to 1, that sprite has lower priority than any background it runs into. The sprite will appear to go behind the background. If a sprite's bit is set to 0, the sprite has higher priority than the background. It will pass in front of the background.

Take a look at Fig. 6-1. It shows one setting of the sprite to background control register. To set sprites to background priori-

Value stored at VIC+27=128+16+8+1=153

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit value → | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Bit number → | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |

Sprites #1, #2, #5, & #6 will appear in front of background images.
Sprites #0, #3, #4, & #7 will appear behind background images.

Fig. 6-1. This setting of the sprite-to-background control register means that sprites #1, #2, #5, and #6 will appear in front of background images, while the other sprites will appear behind background images.

ties, start by putting 1's in the bit positions that correspond to sprites you want to have lower priorities. Then add up the bit values of those bits, and poke the resulting number into VIC+27.

Figure 6-2 is a listing of the program Over and Under. It uses changing priorities to show a sprite orbitting a block of text. Type it in,

```
1000 REM *** OVER AND UNDER ***
1010 :
1020 :
1030 REM ** DRAW THE CENTRAL SHAPE
1040 :
1050 PRINT "▮▮";     :REM CLEAR AND CENTER
1060 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮";
1070 PRINT "▮▮▮▮▮▮▮▮▮▮▮▮";
1080 :
1090 PRINT "▮";     :REM DRAW IT IN CYAN
1100 FOR N = 1 TO 6
1110 :   PRINT "▮          ▮▮▮▮▮▮▮▮▮▮";
1120 NEXT N
1130 PRINT "▮";     :REM BACK TO WHITE
1140 :
1150 :
1160 REM ** SET UP THE SPRITE
1170 :
```

```
1180 FOR N = 832 TO 894 :REM LOAD DATA
1190 :    POKE N, 255
1200 NEXT N
1210 :
1220 VIC = 53248        :REM GRAFIX CHIP
1230 POKE VIC+33, 0     :REM BLACK BKGRND
1240 POKE 2040, 13      :REM #0 DATA PNTR
1250 POKE VIC+39, 12    :REM #0 MDM GRAY
1260 POKE VIC, 104      :REM #0 HORZ POS
1270 POKE VIC+1, 136    :REM #0 VERT POS
1280 POKE VIC+21, 1     :REM SPRITE #0 ON
1290 :
1300 :
1310 REM ** FLY THE SPRITE
1320 :
1330 DR = 1 :REM SPRITE PATH DIRECTION
1340 PR = 0 :REM SPRITE/BKGRND PRIORITY
1350 :
1360 FOR MOVE = 1 TO 136     :REM FLY IT
1370 :    POKE VIC, PEEK(VIC) + DR
1380 :    GET KP$
1390 :    IF KP$ = "" THEN 1410
1400 :        MOVE = 136 : BYEBYE = -1
1410 NEXT MOVE
1420 :
1430 IF BYEBYE THEN 1540  :REM DONE ?
1440 :
1450 DR = -DR       :REM CHANGE DIRECTION
1460 PR = 1 - PR    :REM CHANGE PRIORITY
1470 :
1480 POKE VIC+27,PR    :REM POKE PRIORITY
1490 GOTO 1360         :REM MORE FLYING
1500 :
1510 :
1520 REM ** CLEAN UP AND END
1530 :
1540 POKE VIC+21, 0 :REM SPRITE OFF
1550 POKE VIC+27, 0 :REM RESET PRIORITY
1560 PRINT "♥";      :REM CLEAR SCREEN
1570 :
1580 END
```

Fig. 6-2. Listing of the program Over and Under.

save it, and then run it. Pressing the spacebar will end the program.

As mentioned at the outset, this chapter will have shortened program explanations. That way, more topics and programs will fit. Let's take a brief look at Over and Under.

The first module draws a large square, using reversed cyan spaces and cursor motion commands. The next module sets up a simple medium gray sprite.

Next comes the main program module. Lines 1360-1410 move the sprite to the right or the left, depending on the current value of a direction variable, DR. A keypress during the motion ends the program by setting the flag BYEBYE to True.

After a set of moves, the direction and sprite to background priority are changed. It's amazing how the simple priority switch can change our perception of the sprite's motion. It looks as if the sprite is orbitting the central square, rather than just moving from side to side.

## 6.2 USING TEXT WITH A BIT MAPPED DISPLAY

Back in the last chapter, in Section 5.8, you got to see the strange way bytes in a bit map correspond to the screen display. The setup doesn't make much sense when you're trying to draw lines. It does come in handy when you want to add text characters to bit mapped material. Let's do a little review to see why.

In bit mapped mode, eight consecutive bytes of memory control an area on the screen eight pixels wide and eight pixels high. Each byte controls a row of this image block: the first byte controls the topmost row, the second byte the next row down, and so on.

Character information is stored in the same format. Eight consecutive bytes of memory form a character that's eight pixels wide and eight pixels high. The first byte controls the topmost row of the character, the second byte the next row down, and so on. Patterns for 512 characters are provided in the built-in character ROM, and you can also design your own.

In order to place a character on a bit mapped screen, you just transfer its eight bytes to an eight byte section of the bit map. Figure 6-3 is a listing of a program that does just that. The imaginatively named Bit Mapped Text takes character patterns from the built-in ROM and puts them onto a bit-mapped display. Let's take a brief look at it.

The first section of the program initializes a number of constants and variables. It also sets the keyboard up so all keys will repeat. The next section, lines 1170-1180, switches the display over to bit map mode.

The next two segments create a Jackson Pollack painting. Lines 1230-1250 set the colors for the bit map. Colors for 0 bits are chosen at random, while all bits set to 1 will be black. Then lines 1300-1320 fill the bit map itself with random values.

Lines 1370-1380 wait for a keypress. If the key pressed is a space, the program jumps to its last module and ends. Lines 1400-1410 make sure the key is a letter, number, or punctuation mark.

The next program module figures out the display code for the pressed key. Then the built-in character ROM is brought into mem-

```
1000 REM *** BIT MAPPED TEXT ***
1010 :
1020 :
1030 REM ** INITIALIZE VARIOUS STUFF
1040 :
1050 PRINT "♥";      :REM CLEAR SCREEN
1060 POKE 650, 128 :REM ALL KEYS REPEAT
1070 ROM = 53248   :REM CHARACTER ROM
1080 BASE = 8192   :REM BIT MAP BASE
1090 CURSR = BASE  :REM BIT MAP CURSOR
1100 VIC = 53248   :REM GRAFIX CHIP
1110 BLOC = VIC+24 :REM LOCATES BM
1120 BSET = VIC+17 :REM SETS BMM
1130 :
1140 :
1150 REM ** TURN ON BIT MAP MODE
1160 :
1170 POKE BLOC, PEEK(BLOC) OR 8
1180 POKE BSET, PEEK(BSET) OR 32
1190 :
1200 :
1210 REM ** SET BIT MAP COLORS RANDOMLY
1220 :
1230 FOR SL = 1024 TO 2023
1240 :    POKE SL, INT(RND(1) * 15) + 1
1250 NEXT SL
1260 :
1270 :
1280 REM ** FILL BIT MAP WITH GARBAGE
1290 :
1300 FOR BMLOC = BASE TO BASE + 7999
1310 :    POKE BMLOC, INT(RND(1) * 256)
1320 NEXT BMLOC
1330 :
1340 :
1350 REM ** GET A LETTER, NUMBER, OR
            PUNCTUATION MARK
1360 :
1370 GET KP$
1380 IF KP$ = "" THEN 1370
1390 IF KP$ = " " THEN 1790
1400 IF ASC(KP$) < 32 THEN 1370
1410 IF ASC(KP$) > 95 THEN 1370
1420 :
```

```
1430 :
1440 REM ** FIGURE OUT THE DISPLAY CODE
1450 :
1460 ADJFAC = (ASC(KP$) > 63)
1470 DSCODE = ASC(KP$) + (ADJFAC * 64)
1480 SA = ROM + (DSCODE * 8)
1490 :
1500 :
1510 REM ** BRING CHAR ROM INTO MEMORY
1520 :
1530 POKE 56334, PEEK(56334) AND 254
1540 POKE 1, PEEK(1) AND 251
1550 :
1560 :
1570 REM ** CHAR PATTERNS TO BIT MAP
1580 :
1590 FOR BYTE = 0 TO 7
1600 :    POKE CURSR + BYTE,
                         PEEK (SA + BYTE)
1610 NEXT BYTE
1620 :
1630 :
1640 REM ** LET CHAR ROM GO
1650 :
1660 POKE 1, PEEK(1) OR 4
1670 POKE 56334, PEEK(56334) OR 1
1680 :
1690 :
1700 REM ** ADJUST CURSOR AND LOOP BACK
1710 :
1720 CURSR = CURSR + 8
1730 IF CURSR = BASE + 8000 THEN
                              CURSR = BASE
1740 GOTO 1370
1750 :
1760 :
1770 REM ** BACK TO TEXT DISPLAY & END
1780 :
1790 POKE BSET, PEEK(BSET) AND 223
1800 POKE BLOC, 21
1810 :
1820 PRINT "[CLR]";
1830 END
```

Fig. 6-3. Listing of the program Bit Mapped Text.

ory. Lines 1590-1610 copy the eight character pattern bytes into the bit map, and then the character ROM is let go. The next section updates the cursor variable, which keeps track of our position in the bit map, and then loops back to get another keypress. So much for explanation. If you haven't done so already, type the program in, save it, run it, and experiment with it.

## 6.3 JOYSTICKS

You can plug two standard video game joysticks into your Commodore 64. Let's see how you can get at the information that comes from a joystick. Then you'll use that information to fly a sprite.

A joystick has four direction switches, which you can label with compass directions as shown in Fig. 6-4. At any time, none, one, or two switches may be activated. For example, if you push the joystick north, switch 0 is activated. If you push it southwest, switches 1 and 2 are activated. If you don't push it at all, no switches are activated. There's also a fifth switch on the joystick, and it's used as a fire button.

Each switch is connected to a bit in a special input/output location in the computer.

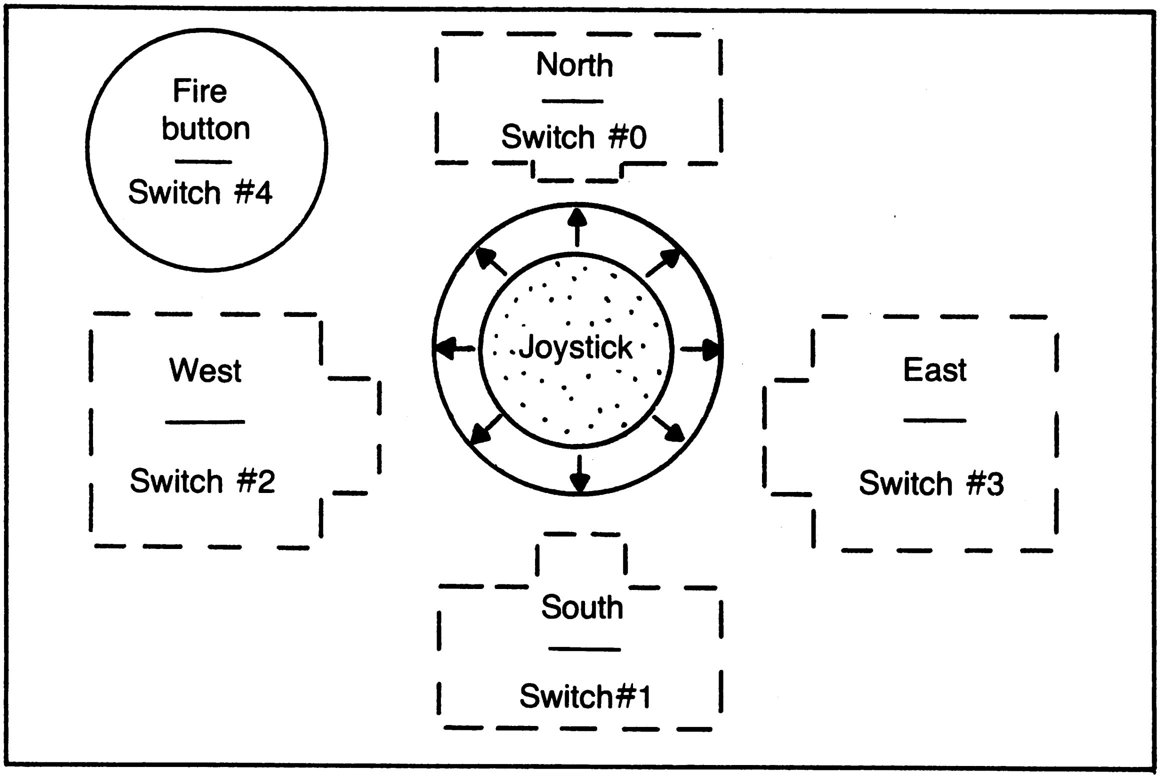



Fig. 6-4. A joystick and its five switches, as seen from above with limited x-ray vision.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit value→ | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Bit number→ | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | — | — | — | Switch #4 — Fire button | Switch #3 — East | Switch #2 — West | Switch #1 — South | Switch #0 — North |

Bits 5, 6, 7 used for other purposes

**Fig. 6-5. How the five joystick switches connect to the lower five bits of the input/output register at memory location 56321 or 56320.**

The five switches of the joystick plugged into control port 1 are connected to the lower five bits of the input/output register at memory location 56321. Likewise, the five switches of the joystick plugged into control port 2 are connected to the lower five bits of the input/output register at memory location 56320. See Fig. 6-5.

By the way, these input/output locations are also used by the computer's operating system to scan the keyboard. Because of some complications caused by this keyboard scanning, strange things can happen with a joystick plugged into control port 1. So, if you're just using one joystick, plug it into control port 2.

You can tell what's happening to a joystick by reading the data from the corresponding input/output register. When a switch is not activated, the corresponding bit will be set to 1. When the switch is activated, the bit will be set to 0. For example, if you push the joystick to the east, it will activate switch 3, so bit 3 of the input/output byte will be set to 0. If you press the fire button, that activates switch 5, so bit 5 will be set to 0. Figure 6-6 gives some more examples of this.

By using the AND function, you can isolate the bits you're interested in checking. Based on the results, you can figure out new values for a sprite's position and move it around the screen. Programmers are always looking for the quickest, cleverest way to read a joystick. Just remember, no matter how weird the joystick-reading code looks, it's just trying to translate the bit values into horizontal and vertical movement information. In the next section, a program that uses one of these quick and clever techniques will be discussed. But first, you'll take a short course in collision detection.

## 6.4 THINGS THAT GO BUMP ON THE SCREEN

It's useful to know when objects collide with one another on the screen. With previous small computers, this wasn't easy. The Commodore 64 has special built-in hardware to detect collisions.

Sprite to sprite collisions are recorded in a register at VIC+30 (memory location 53278). Each bit of the register corresponds to a sprite. Any sprite involved in a collision gets its bit set to 1. For example, if sprite #2 bumps into sprite #7, bits 2 and 7 of VIC+30 will be set to 1. The bits will stay set until you read information from the register with a peek statement.

Sprite to data collisions are recorded in a register at VIC+31 (memory location 53279). Data means parts of characters or bit mapped images. Again, each bit of the register corresponds to a sprite, and that bit is set to 1 if its sprite is in a collision. For example, if sprite #5 bumps into parts of a character, bit 5 of VIC+31 will be set to 1. The bits stay set until the contents of the register are read.

Figure 6-7 lists the program Joyous Collision. It gives examples of joystick reading and sprite to sprite collision detection. Type it in, save it, and then run it. Two sprites will appear, as shown in Fig. 6-8. Use a joystick plugged into control port 2 to fly the face into the weather vane. Notice what happens when they collide. Pressing the fire button will end the program.

Let's review this program. Lines 1050-1090 load the data for both sprites. Lines 1380-1540 then set the necessary VIC registers and turn both sprites on.

Now comes the program's main segment. Line 1590 reads the value of the input/output location at 56320. Remember, that's the register that talks to the joystick plugged into control port 2. Line 1600 uses an ANDing operation to see if the fire button's been pressed. If it has, the program exits via the cleanup

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| — | — | — | 1 | 1 | 1 | 1 | 0 | Joystick pushed north |
| — | — | — | 1 | 1 | 0 | 0 | 1 | Joystick pushed southwest |
| — | — | — | 0 | 1 | 0 | 1 | 1 | Joystick pushed west & fire button pressed |
| — | — | — | 0 | 1 | 1 | 1 | 1 | Fire button pressed |

Fig. 6-6. Examples of what the lower five bits of memory location 56321 look like when an attached joystick is manipulated in various ways.

```
1000 REM *** JOYOUS COLLISION ***
1010 :
1020 :
1030 REM ** LOAD THE SPRITE DATA
1040 :
1050 PRINT "♥"
1060 FOR N = 832 TO 958
1070 :    READ SPDTA
1080 :    POKE N, SPDTA
1090 NEXT N
1100 :
1110 DATA   0, 255,   0,   1, 129, 128
1120 DATA   3,   0, 192,   3,   0, 192
1130 DATA   3,   0, 192,   6, 102,  96
1140 DATA  60, 102,  60,  96,   0,   6
1150 DATA 192,   0,   3, 192, 102,   3
1160 DATA 198,  60,  99,  99,   0, 198
1170 DATA 113, 129, 142,  28, 195,  56
1180 DATA  12, 195,  48,  12, 102,  48
1190 DATA   6,  60,  96,   6,   0,  96
1200 DATA   3, 129, 192,   0, 195,   0
1210 DATA   0, 126,   0,   0
1220 :
1230 DATA   0,   0,   0,   0,  16,   0
1240 DATA   0,  56,   0,   0,  84,   0
1250 DATA   0,  16,   0,   2,  16, 128
1260 DATA   1,  17,   0,   0, 146,   0
1270 DATA  16,  84,  16,  32,  56,   8
1280 DATA 127, 255, 252,  32,  56,   8
1290 DATA  16,  84,  16,   0, 146,   0
1300 DATA   1,  17,   0,   2,  16, 128
1310 DATA   0,  16,   0,   0,  84,   0
1320 DATA   0,  56,   0,   0,  16,   0
1330 DATA   0,   0,   0
1340 :
1350 :
1360 REM ** SET SPRITES UP AND TURN ON
1370 :
1380 VIC = 53248       :REM GRAPHICS CHIP
1390 POKE VIC+33,0     :REM BKGROUND BLACK
1400 :
1410 POKE 2040,13      :REM #0 DATA POINTR
1420 POKE 2041,14      :REM #1 DATA POINTR
```

```
1430 :
1440 POKE VIC,120    :REM #0 HORIZONTAL
1450 POKE VIC+2,160  :REM #1 HORIZONTAL
1460 POKE VIC+1,138  :REM #1 VERTICAL
1470 POKE VIC+3,126  :REM #1 VERTICAL
1480 :
1490 POKE VIC+39,3   :REM #0 IS CYAN
1500 POKE VIC+40,7   :REM #1 IS YELLOW
1510 POKE VIC+29,2   :REM ONLY #1 IS
1520 POKE VIC+23,2   :REM DOUBLE-SIZED
1530 :
1540 POKE VIC+21,3   :REM TURN BOTH ON
1550 :
1560 :
1570 REM ** FLY SPRITE #0
1580 :
1590 JR = PEEK (56320) :REM CTRL PORT 2
1600 IF (JR AND 16) = 0 THEN 1870
1610 HD = SGN(JR AND 4) - SGN(JR AND 8)
1620 VD = SGN(JR AND 1) - SGN(JR AND 2)
1630 :
1640 POKE VIC, PEEK(VIC) + HD
1650 POKE VIC+1, PEEK(VIC+1) + VD
1660 :
1670 :
1680 REM ** IF NO COLLISIONS LOOP BACK
1690 :
1700 IF PEEK(VIC+30) = 0 THEN 1490
1710 :
1720 :
1730 REM ** COLLISION : #1 GOES WHITE
            AND #0 VIBRATES RAINBOWS
1740 :
1750 POKE VIC+40, 1
1760 :
1770 HUE = PEEK(VIC+39) AND 15
1780 HUE = HUE + 1
1790 IF HUE = 8 THEN HUE = 1
1800 POKE VIC+39, HUE
1810 :
1820 GOTO 1590
1830 :
1840 :
1850 REM ** CLEAN UP AND END
```

```
1860 :
1870 POKE VIC+21,0
1880 POKE VIC+29,0
1890 POKE VIC+23,0
1900 :
1910 END
```

Fig. 6-7. Listing of the program Joyous Collision.

routine that begins at line 1870.

Lines 1610 and 1620 take the value of location 56320 and figure out the net horizontal and vertical motion. They do it with a quick, tricky technique. ANDing isolates individual bits corresponding to individual switches in the joystick. The SGN function returns values of 0 or 1, depending on whether the expression in parentheses comes out to be 0 or greater than 0. Depending on how the joystick is moved, HD will be given one of the values −1, 0, or 1. The same goes for VD, the variable that holds values for vertical motion. These motion values are then used to update sprite #0's position.

Line 1700 then checks the sprite to sprite collision register. If the sprites aren't bumping

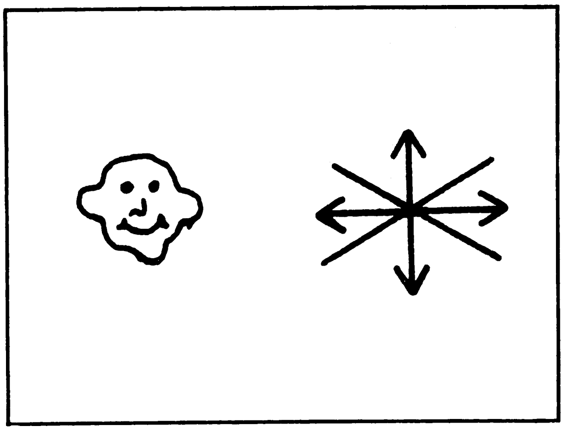


Fig. 6-8. Initial image shown by the program Joyous Collision.

into one another, the program loops back to reset the original sprite colors and look at the joystick again. If there is a collision, lines 1750-1800 change the sprites' colors before going back to read the joystick.

## 6.5 MULTICOLOR CHARACTER MODE

Back in Chapter 3, Sections 3.1 through 3.4, you learned how to create multicolor sprites. By trading off a little horizontal resolution, you were able to get more colors into a sprite design.

There's also a multicolor mode for character displays. Again, you trade off a little horizontal resolution for a wider range of colors. You can use this multicolor mode with either the built-in ROM characters or characters you design from scratch.

As with multicolor sprites, multicolor characters use two bits to choose a color. Thus, four double-wide pixels will make up each row of the character. You may remember that two bits can take on four possible values: 00, 01, 10, and 11. That lets you use four colors in a multicolor character.

Setting bit 4 of the register at VIC+22 (memory location 53270) to 1 turns on multicolor character mode. Resetting the same bit to 0 turns it off. To add even more control (and complication), each location on the screen has the option of going with multicolor mode or not. If a screen location's corresponding color map location has bit 3 set to 1, the character will show up in multicolor mode. If bit 3 of color memory is set to 0, the character will show up in its normal (two color) fashion.

Confusing? Here's another way to look at. Assume that you've turned on multicolor character mode by setting bit 4 of VIC+22 to 1. If you put a number from 0-7 in a color memory location, the corresponding screen location will show its character normally. But, if you put a number from 8-15 into the color memory location, the character will show up in multicolor mode.

Next detail: if multicolor character mode is on, and a character's color memory location

| Bit value ▶ | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | Number codes ▼ |
|---|---|---|---|---|---|---|---|---|---|
| Byte 0 | | | | | | | | | |
| Byte 1 | | | | | | | | | |
| Byte 2 | | | | | | | | | |
| Byte 3 | | | | | | | | | |
| Byte 4 | | | | | | | | | |
| Byte 5 | | | | | | | | | |
| Byte 6 | | | | | | | | | |
| Byte 7 | | | | | | | | | |

| 0 0 | 0 1 | 1 0 | 1 1 |
|---|---|---|---|
| | | | |
| Background #0 color (screen color) | Background #1 color | Background #2 color | Lower 3 bits of color memory color |

Fig. 6-9. A coding form you can use to design multicolor characters.

| Bit value ▶ | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | Number codes |
|---|---|---|---|---|---|---|---|---|---|
| Byte 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 107 |
| Byte 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 107 |
| Byte 2 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 107 |
| Byte 3 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 67 |
| Byte 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 193 |
| Byte 5 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 233 |
| Byte 6 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 233 |
| Byte 7 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 233 |

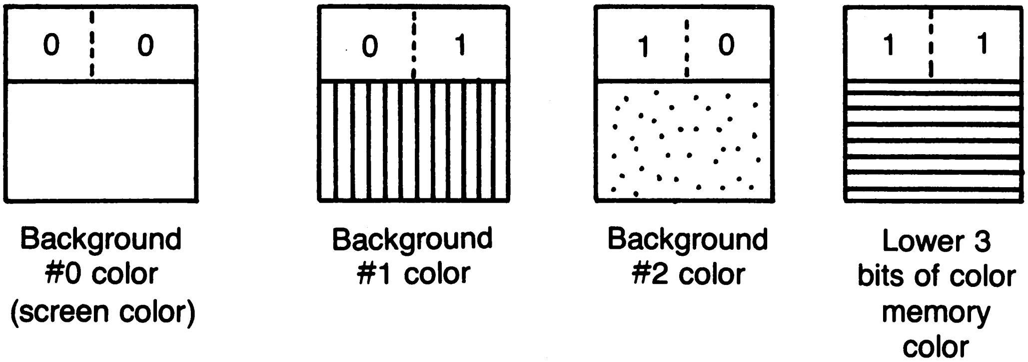



Fig. 6-10. An example showing how the multicolor character coding form can be used.

is set to a number from 8-15, where do the four colors come from? If the bit pair is 00, the color comes from the value stored at VIC+33, the screen color register, also called background register 0. If the bit pair is 01, the color comes from VIC+34, background register 1. If the bit pair is 10, the color comes from VIC+35, background register 2. Finally, if the bit pair is 11, the color comes from the lower 3 bits of the character's color memory location.

If you stop and think for a moment, you'll realize that all characters displayed in multicolor mode will share three colors. Poking new values into the three background registers will quickly change a whole screen of multicolor characters.

You can use multicolor mode with the built-in characters, but the results aren't very interesting. It's more fun to design your own multicolor characters. Figure 6-9 is a coding form you can use for this task. Figure 6-10 is an example of how this form can be used. I recommend using colored markers to represent the four colors, but in a black-and-white book, I have to resort to shading.

Figure 6-11 lists a program that demonstrates multicolor characters. Type it in, save it, and then run it. Pressing any of the keys 1, 2, 3, or 4 will change one of the four colors used in the display. Holding one of those keys down will cause continuous color change. Notice how quickly the picture shifts when a new value is poked into one of the background registers.

Playing around with this program will teach you a lot about multicolor character

```
1000 REM *** CUSTOM MULTICOLOR ***
1010 :
1020 :
1030 REM ** LOAD IN NEW A, B, & SPACE
1040 :
1050 CBASE = 12288 :REM NEW CHARS START
1060 :
1070 FOR CHAR = 1 TO 2
1080 :   FOR BYTE = 0 TO 7
1090 :     SPOT = CBASE + CHAR*8 + BYTE
1100 :     READ CDTA
1110 :     POKE SPOT, CDTA
1120 :   NEXT BYTE
1130 NEXT CHAR
1140 :
1150 FOR BYTE = 0 TO 7
1160 :   SPOT = CBASE + 32*8 + BYTE
1170 :   POKE SPOT, 0
1180 NEXT BYTE
1190 :
1200 DATA 107, 107, 107,  67
1210 DATA  67, 107, 107, 107
1220 DATA 233, 233, 233, 193
1230 DATA 193, 233, 233, 233
1240 :
1250 :
1260 REM ** CLEAR SCREEN, BRING IN NEW
        CHAR SET, TURN MULTICOLOR ON
1270 :
1280 PRINT "♥";        :REM CLEAR SCREEN
```

```
1290 UIC = 53248        :REM GRAFIX CHIP
1300 POKE UIC+24, 29   :REM NEW SET IN
1310 POKE UIC+22, PEEK(UIC+22) OR 16
1320 :
1330 :
1340 REM ** SET UP DISPLAY
1350 :
1360 PRINT "QQQQQQQQQQ"; :REM DOWN 10
1370 PRINT "]]]]]]]";     :REM OVER 7
1380 PRINT "[illegible]"; :REM START WITH COLOR 9
1390 :
1400 FOR N = 1 TO 26
1410 :  PRINT "AB";
1420 :  IF N <> 13 THEN 1440
1430 :     PRINT:PRINT:PRINT "]]]]]]]";
1440 NEXT N
1450 :
1460 :
1470 REM ** PLAY BUTTON PUSH
1480 :
1490 COLMAP = 55296
1500 BG = COLMAP + (10 * 40) + 7
1510 POKE 650, 128 :REM ALL KEYS REPEAT
1520 :
1530 GET KP$
1540 IF KP$ = "" THEN 1530
1550 IF KP$ = " " THEN 1830
1560 :
1570 BKREG = 0
1580 IF KP$ = "1" THEN BKREG = UIC+33
1590 IF KP$ = "2" THEN BKREG = UIC+34
1600 IF KP$ = "3" THEN BKREG = UIC+35
1610 IF KP$ = "4" THEN GOSUB 1720
1620 IF BKREG = 0 THEN 1530
1630 :
1640 HUE = (PEEK(BKREG) AND 15) + 1
1650 IF HUE = 16 THEN HUE = 0
1660 POKE BKREG, HUE
1670 GOTO 1530
1680 :
1690 :
1700 REM ** SUBROUTINE TO CHANGE ALL
             LETTERS' COLOR MAP COLORS
1710 :
```

```
1720 HUE = (PEEK(BG) AND 15) + 1
1730 IF HUE > 15 THEN HUE = 0
1740 :
1750 FOR SPOT = BG TO (BG + 106)
1760 :    POKE SPOT, HUE
1770 NEXT SPOT
1780 RETURN
1790 :
1800 :
1810 REM ** CLEAN UP AND END
1820 :
1830 PRINT "♥";
1840 POKE VIC+22, PEEK(VIC+22) AND 239
1850 POKE VIC+24, 21
1860 PRINT "■"      :REM WHITE TEXT
1870 POKE VIC+33,0 :REM ON BLACK BKGRND
1880 :
1890 END
```

Fig. 6-11. Listing of the program Custom Multicolor.

mode. The program is pretty simple. The first segment loads in two custom character patterns and the pattern for a space. Then the screen clears; VIC is set to point to the new character set; and the multicolor mode comes on. Lines 1360-1440 print two lines full of the new characters.

Now comes the workhorse section. The program gets a keypress. If it's a space, the program ends. If it's a 1, 2, 3, or 4, the appropriate color storage location(s) is (are) changed. Then the program loops back for another keypress.

One technique you might make note of: when reading a color from memory, an AND operation is used to screen out unwanted bits. This happens in lines 1640 and 1720.

## 6.6 EXTENDED BACKGROUND CHARACTER MODE

There is one more way you can display characters: extended background mode. In this mode, you can use any one of the 16 colors for a character's background. As usual, the character itself can take on any of the 16 colors.

There are four memory locations used with extended background mode: background registers 0-3, located at VIC+33, VIC+34, VIC+35, and VIC+36 respectively. That's memory locations 53281 through 53284. Each of these locations can be set to any one of the 16 colors.

As you've seen, getting more colorful displays usually means cutting down on something else. Extended background mode is no exception. Only 64 different characters can be displayed, rather than 256. This is because bits 6 and 7 of each character code are used to select one of the four background registers. That leaves just six bits to code the character, and the laws of binary arithmetic say that six bits produce 64 different values.

Let's look at some practical details. Putting a 1 into bit 6 of memory location 53265, VIC+17, turns on extended color mode. Placing a 0 into the same bit position turns the mode off. The character's color is stored in color memory, as in the normal character mode. The character code is stored in screen memory, also as usual. However, only the first 64 character patterns are used. If the first two bits of a character code are 00, the background color comes from background register 0, at VIC+33. If the first two bits of the code are 01, 10, or 11, the background color comes from background register 1, 2, or 3, respectively.

For example: if extended background color mode is in effect, poking a 5 into a screen memory location will put an E on the screen. The character's background color will come from background register 0, at VIC+33. Since that register sets the background color for the whole screen, the E will appear quite ordinary. Poking a 69 into a screen memory lcoation will also put an E on the screen, but the character's 8-by-8 area will fill with a background color based on the contents of VIC+34. Likewise, poking a 133 will produce an E with local background color based on the contents of VIC+35. Poking 197 into screen memory will produce an E with a background color based on the contents of VIC+36.

Figure 6-12 lists the program Extended Background, which gives a demonstration of this mode. Type it in, save it, and then run it. Each column of dashes shares the same background register. Pressing one of the keys 1-4 will change the contents of one of the background registers. Pressing 5 will change the color of the character itself. Once again, if you really want to understand a new mode, spend some time modifying the program.

Here's a brief explanation of Extended Background: lines 1050-1070 clear the screen and turn on extended background mode. Lines 1120-1250 set up four columns of the same character, a dash (display code 45). However,

```
1000 REM *** EXTENDED BACKGROUND ***
1010 :
1020 :
1030 REM ** TURN ON EXTENDED BKGRD MODE
1040 :
1050 PRINT "♥";          :REM CLEAR SCREEN
1060 VIC = 53248         :REM GRAFIX CHIP
1070 POKE VIC+17, PEEK(VIC+17) OR 64
1080 :
1090 :
1100 REM ** SET UP DISPLAY
1110 :
1120 SCREEN = 1024
1130 COLMAP = 55296
1140 SS = SCREEN + (10 * 40) + 16
1150 CS = COLMAP + (10 * 40) + 16
```

```
1160 :
1170 HUE = PEEK(CS) + 1
1180 IF HUE = 16 THEN HUE = 0
1190 :
1200 FOR RW = 0 TO 3
1210 :FOR N = 0 TO 3
1220 : POKE SS + RW*40 + N*2, 45 + 64*N
1230 : POKE CS + RW*40 + N*2, HUE
1240 : NEXT N
1250 NEXT RW
1260 :
1270 :
1280 REM ** PLAY BUTTON PUSH
1290 :
1300 POKE 650, 128 :REM ALL KEYS REPEAT
1310 :
1320 GET KP$
1330 IF KP$ = "" THEN 1320
1340 IF KP$ = " " THEN 1520
1350 :
1360 BKREG = 0
1370 IF KP$ = "1" THEN BKREG = VIC+33
1380 IF KP$ = "2" THEN BKREG = VIC+34
1390 IF KP$ = "3" THEN BKREG = VIC+35
1400 IF KP$ = "4" THEN BKREG = VIC+36
1410 IF KP$ = "5" THEN 1170
1420 IF BKREG = 0 THEN 1320
1430 :
1440 HUE = (PEEK(BKREG) AND 15) + 1
1450 IF HUE = 16 THEN HUE = 0
1460 POKE BKREG, HUE
1470 GOTO 1320
1480 :
1490 :
1500 REM ** CLEAN UP AND END
1510 :
1520 PRINT "[illegible]";
1530 POKE VIC+17, PEEK(VIC+17) AND 191
1540 PRINT "[illegible]"      :REM WHITE TEXT
1550 POKE VIC+33,0 :REM ON BLACK BKGRND
1560 :
1570 END
```

Fig. 6-12. Listing of the program Extended Background.

each column differs in bits 6 and 7, so the columns of dashes will look to different registers for background colors.

The next section is another big keyboard polling loop. A space ends things, the numbers 1-5 change colors as noted above, and anything else is ignored. Finally, the last module cleans things up by turning extended background mode off, clearing the screen, and setting the character color to white.

### 6.7 MULTICOLOR BIT MAP MODE

There is one last Commodore 64 display option: multicolor bit map mode. As you may have guessed, this graphic mode lets you use 4 colors in an 8-by-8 block of the bit map display. You've probably also guessed the cost: horizontal resolution cut in half.

How do you set this mode up? First, you put a 1 into bit 5 of VIC+17 to turn on bit map mode. Then you tell VIC where the 8K bit map is located by setting bit 3 of VIC+24. In most cases, that bit will be set to 1. So far, these are just the steps you used to set up standard bit mapping. Finally, you set bit 4 of VIC+22 to 1, which turns on multicolor mode.

The correspondence between bytes in the bit map and the dots on the screen display is the same as in standard bit map mode. However, two bits are used to choose a color for a double-wide pixel. As you've learned, two bits

```
1000 REM *** COLOR BIT MAPPED TEXT ***
1010 :
1020 :
1030 REM ** INITIALIZE VARIOUS STUFF
1040 :
1050 PRINT "J";       :REM CLEAR SCREEN
1060 POKE 650, 128 :REM ALL KEYS REPEAT
1070 ROM = 53248   :REM CHARACTER ROM
1080 BASE = 8192   :REM BIT MAP BASE
1090 CURSR = BASE  :REM BIT MAP CURSOR
1100 VIC = 53248   :REM GRAFIX CHIP
1110 BLOC = VIC+24 :REM LOCATES BM
1120 BSET = VIC+17 :REM SETS BMM
1130 :
1140 :
1150 REM ** TURN ON BIT MAP MODE
1160 :
1170 POKE BLOC, PEEK(BLOC) OR 8
1180 POKE BSET, PEEK(BSET) OR 32
1190 :
1200 :
1210 REM ** SET BIT MAP COLORS RANDOMLY
1220 :
```

```
1230 FOR SL = 1024 TO 2023
1240 :    POKE SL, (INT(RND(1) * 15) + 1)         * 16
1250 NEXT SL
1260 :
1270 :
1280 REM ** FILL BIT MAP WITH GARBAGE
1290 :
1300 FOR BMLOC = BASE TO BASE + 7999
1310 :    POKE BMLOC, INT(RND(1) * 256)
1320 NEXT BMLOC
1330 :
1340 :
1350 REM ** GET A LETTER, NUMBER, OR  PUNCTUATION MARK
1360 :
1370 GET KP$
1380 IF KP$ = "" THEN 1370
1390 IF KP$ = " " THEN 1790
1400 IF ASC(KP$) < 32 THEN 1370
1410 IF ASC(KP$) > 95 THEN 1370
1420 :
1430 :
1440 REM ** FIGURE OUT THE DISPLAY CODE
1450 :
1460 ADJFAC = (ASC(KP$) > 63)
1470 DSCODE = ASC(KP$) + (ADJFAC * 64)
1480 SA = ROM + (DSCODE * 8)
1490 :
1500 :
1510 REM ** BRING CHAR ROM INTO MEMORY
1520 :
1530 POKE 56334, PEEK(56334) AND 254
1540 POKE 1, PEEK(1) AND 251
1550 :
1560 :
1570 REM ** CHAR PATTERNS TO BIT MAP
1580 :
1590 FOR BYTE = 0 TO 7
1600 :    POKE CURSR + BYTE, PEEK (SA + 7 - BYTE)
1610 NEXT BYTE
1620 :
1630 :
```

```
1640 REM ** LET CHAR ROM GO
1650 :
1660 POKE 1, PEEK(1) OR 4
1670 POKE 56334, PEEK(56334) OR 1
1680 :
1690 :
1700 REM ** ADJUST CURSOR AND LOOP BACK
1710 :
1720 CURSR = CURSR + 8
1730 IF CURSR = BASE + 8000 THEN CURSR = BASE
1740 GOTO 1370
1750 :
1760 :
1770 REM ** BACK TO TEXT DISPLAY & END
1780 :
1790 POKE BSET, PEEK(BSET) AND 223
1800 POKE BLOC, 21
1810 :
1820 PRINT "J";
1830 END
```

Fig. 6-13. Multicolor Bit Mapped Mode.

can code 4 values. Depending on the value of a bit pair, color information for a given 8-by-8 area can come from one of four locations.

If the bit pair is 00, color comes from background register 0 at VIC+33. That's the screen background color. If the bit pair is 01, color comes from the upper nibble of the corresponding screen memory location. If the bit pair is 10, color comes from the lower nibble of the same byte of screen memory. And if the bit pair is 11, color comes from the corresponding color memory location.

To return to a standard text display from this mode, just reverse the setup steps. That is, put a 0 into bit 5 of VIC+17, put a 0 into bit 4 of VIC+22, and reset VIC+24 with the value 21.

## 6.8 CHAPTER SUMMARY

Whew, this has been a packed chapter. I wanted to wrap up a number of loose ends before going on to the next major topic: sounds. Here's an overview of what's been covered:

* Moving sprites in front of and behind other images by setting sprite to background priorities
* Placing characters on a bit mapped display by transferring eight bytes from character memory
* Reading a joystick by looking at the lower five bits of memory locations 56320 and 56321
* Using joystick information to move a sprite around

* Detecting collisions between sprites and between sprites and other images
* Displaying characters in multicolor mode, where four colors can be used in each character
* Displaying characters in extended background mode, where all 16 colors are available for local background duty
* Setting up multicolor bit map mode, where 4 colors can be used in each 8-by-8 block of the bit map, although horizontal resolution gets cut in half

## 6.9 EXERCISES

### 6.9.1 Self Test

Answers are in Self Test Section 6.9.3.

1. (6.1) Which sprites will move behind background images if the value 85 is poked into the register at VIC+27?
2. (6.2) Give an instance when the strange layout of bytes in the bit map comes in handy.
3. (6.3) Which direction is the joystick being pushed if the input/output register at 56321 holds the value 26?
4. (6.4) If the sprite to sprite collision register contains the value 170, which sprites have collided?
5. (6.5) Setting bit ________ of the register at VIC+22 to ________ turns on multicolor character mode.
6. (6.6) In extended background mode, bits ________ and ________ of a character's display code select one of four background registers.
7. (6.7) Which 3 bits need to be dealt with to set up multicolor bit map mode?

### 6.9.2 Programming Exercises

These should be quick and easy to code. Possible solutions are shown in Section 6.9.4. Question 2 will test your ability to examine printed programs critically.

1. Change the program Over and Under so that the sprite moves in a vertical, rather than horizontal, orbit.
2. Study the program shown in Fig. 6-13. It is a revision of the Bit Mapped Text program that makes the text characters come out in color, upside down, on a black background. Identify the lines that make these cause these changes.
3. Change the program Joyous Collision so the joystick operates in reverse. That is, moving it west moves the sprite to the east, moving it north moves the sprite south, and so on.

### 6.9.3 Answers to Self Test

1. sprites #0, #2, #4, and #6
2. when you want to put characters onto a bit mapped display
3. northwest
4. sprites #1, #3, #5, and #7
5. 4 ; 1
6. 6 ; 7
7. bit 5 of VIC+17 (53265); bit 3 of VIC+24 (53272); bit 4 of VIC+22 (53270)

### 6.9.4 Possible Solutions to Programming Exercises

1. Load in the program Over and Under. Then type in these lines:

```
1000 REM *** VERTICAL OVER & UNDER ***
1260 POKE VIC, 172      :REM #0 HORZ POS
1270 POKE VIC+1, 68     :REM #0 VERT POS
1370 :    POKE VIC+1, PEEK(VIC+1) + DR
```

2. The following lines are the ones that caused the appropriate changes:

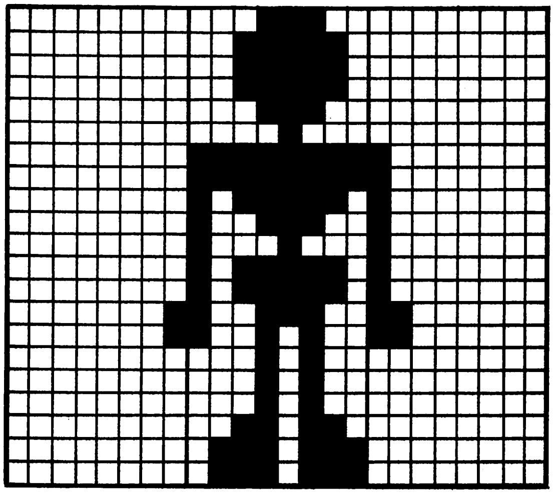
```
1000 REM *** COLOR BIT MAPPED TEXT ***
1240 :    POKE SL, (INT(RND(1) * 15) + 1)
                 * 16
1600 :    POKE CURSR + BYTE,
                        PEEK (SA + 7 - BYTE)
```

3. Load in the program Joyous Collision. Then type in these lines:

```
1000 REM *** WEIRD COLLISION ***
1610 HD = SGN(JR AND 8) - SGN(JR AND 4)
1620 VD = SGN(JR AND 2) - SGN(JR AND 1)
```

# Chapter 7

# Starting To Make Sounds



Enough has been said about silent pictures already. Let's make some noise. In this chapter, I'll give some short, snappy lectures on the nature of sounds. You'll learn about frequency, amplitude, and waveforms. You'll take a good look at SID, the powerful sound chip Commodore has put into your computer. You'll learn how to set some of SID's registers. I'll talk about music and then close up with a familiar melody.

## 7.1 SOME ASPECTS OF SOUND

Things that vibrate create sounds. The classic beginner's sound experiment involves a tuning fork. If you have one, give it a good whack. Listen to it a moment, and then touch it. Feel the vibrations? If you don't have a tuning fork handy, here's a neat little substitute experiment:

Get two pieces of dental floss or string, each about two feet long. Then take a rack out of an oven. Attach one end of a piece of floss to one corner of the rack, then attach the second piece to another corner. Wrap the lose end of one piece of floss around your left index finger, then wrap the end of the other piece around your right index finger. You may want to do the next step in private. Stick your fingers in your ears. Bump the rack against something. Watch, feel, and listen. See Fig. 7-1.

### 7.1.1 Waves

One complete vibration makes a wave. Things that vibrate make lots of waves. These waves like to travel. They travel really well in metal and stretched pieces of floss. They even travel in the air. When sound waves make it to your ear, they crash into sensitive little hairs,

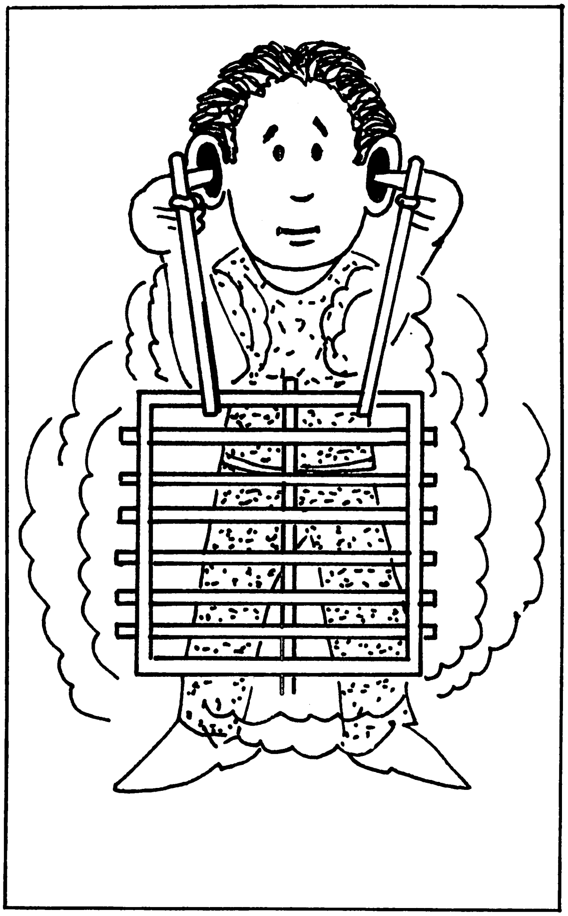


Fig. 7-1. You might want to try this noble sound experiment in the privacy of your own room.

causing the hairs to vibrate. The vibrating hairs are connected to nerves, which send messages to your brain, and you hear sounds.

### 7.1.2 Frequency, or Pitch

There are a number of ways to describe waves. One way is to count how many waves, or cycles, occur in a given amount of time. This count is known as the *frequency* of the waves. For example, if you went to the ocean, you could count the number of waves that occur during one minute. If there were twelve waves, you'd say that the frequency was 12 cycles per minute.

Sound waves occur at a faster rate. You measure the frequency of a sound in cycles per second, also known as hertz. Something vibrating 440 times a second will create a sound with a frequency of 440 hertz.

What we call the pitch of a sound depends on its frequency. Sounds with a low pitch have low frequencies; high-pitched sounds have high frequencies.

People can hear sounds with frequencies between about 15 and 20,000 hertz. A piano can create sounds with frequencies between 33 and 4186 hertz. Your C-64 computer can create sounds with frequencies between .06 and 3995 hertz.

You can draw pictures of sound waves. Figure 7-2 shows waves made by tuning forks. The waves have different frequencies.

### 7.1.3 Amplitude: Volume, or Loudness

You can also measure the size of a wave. This is called *amplitude.* Large waves are more powerful than small waves, as any surfer will testify. With sound waves, amplitude translates into volume, or loudness. The larger the amplitude, the louder the sound.

Frequency and amplitude operate independently of one another. Two sounds can share the same pitch and have different loudness levels. Likewise, two sounds can be equally loud but have different pitches. Figure 7-3 shows waves that have the same frequency but different amplitudes.

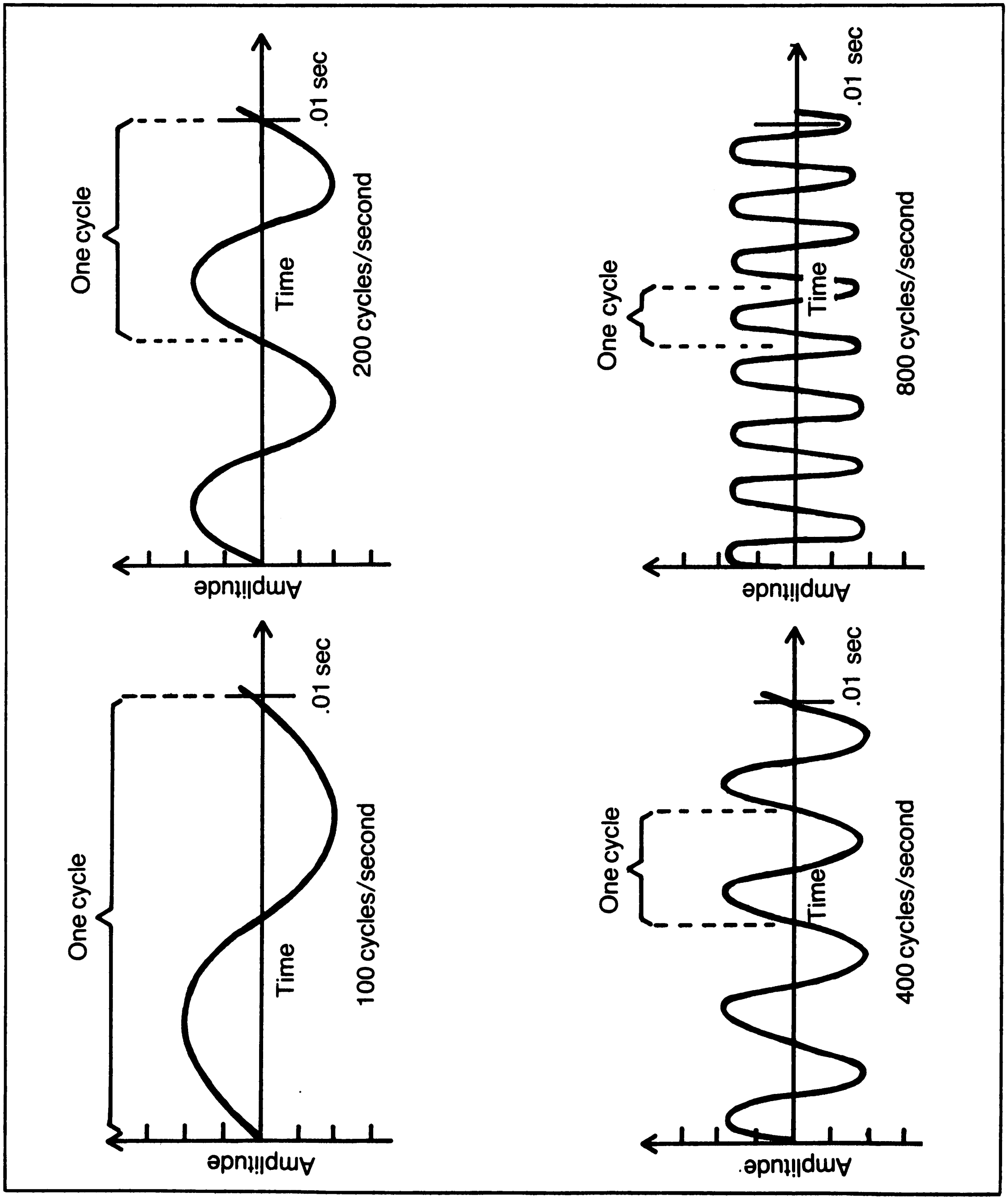



**Fig. 7-2. Pictures of waves made by tuning forks at different frequencies. The waves all have the same amplitude.**

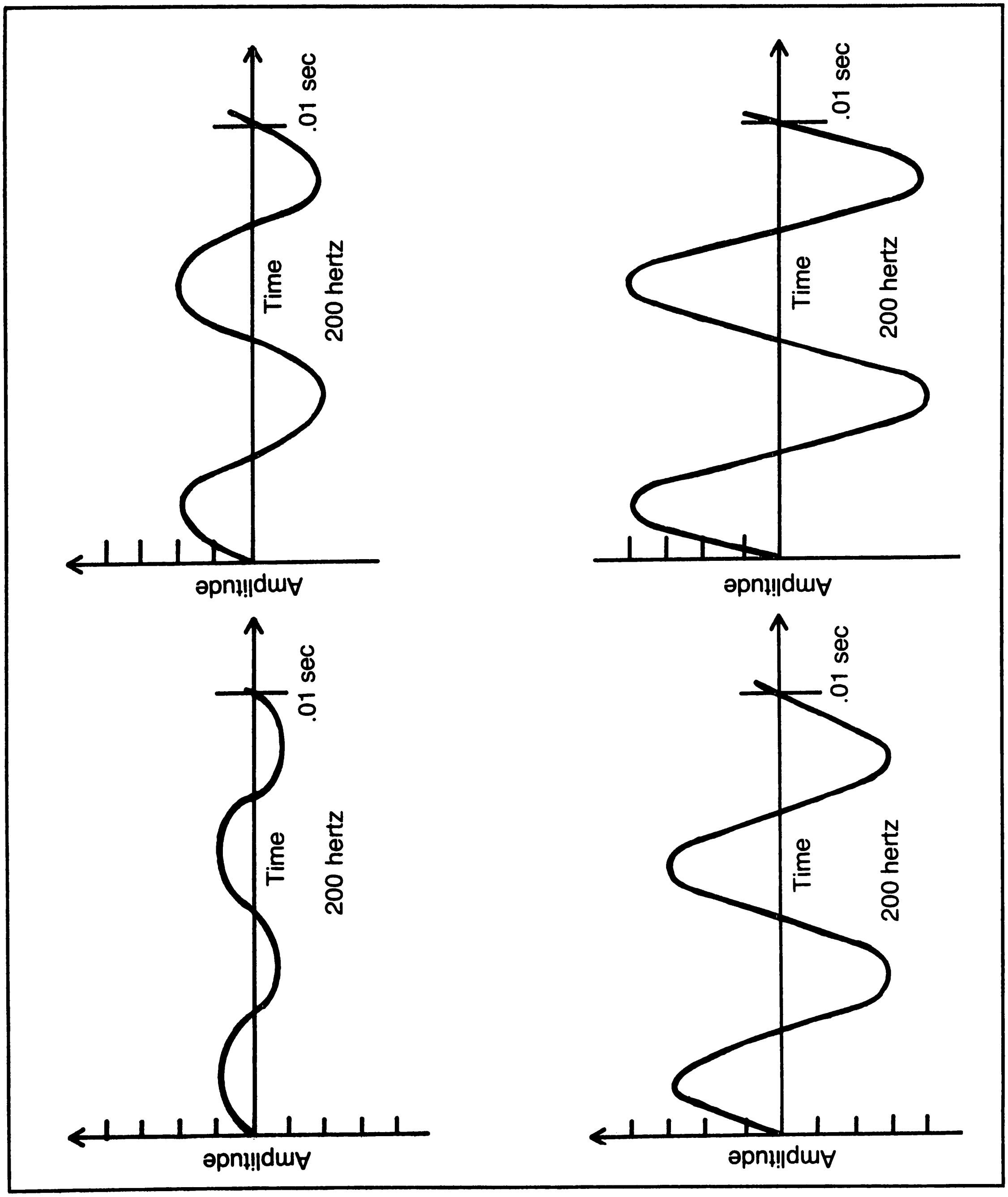



Fig. 7-3. Pictures of tuning fork waves that have the same frequency but different amplitudes.

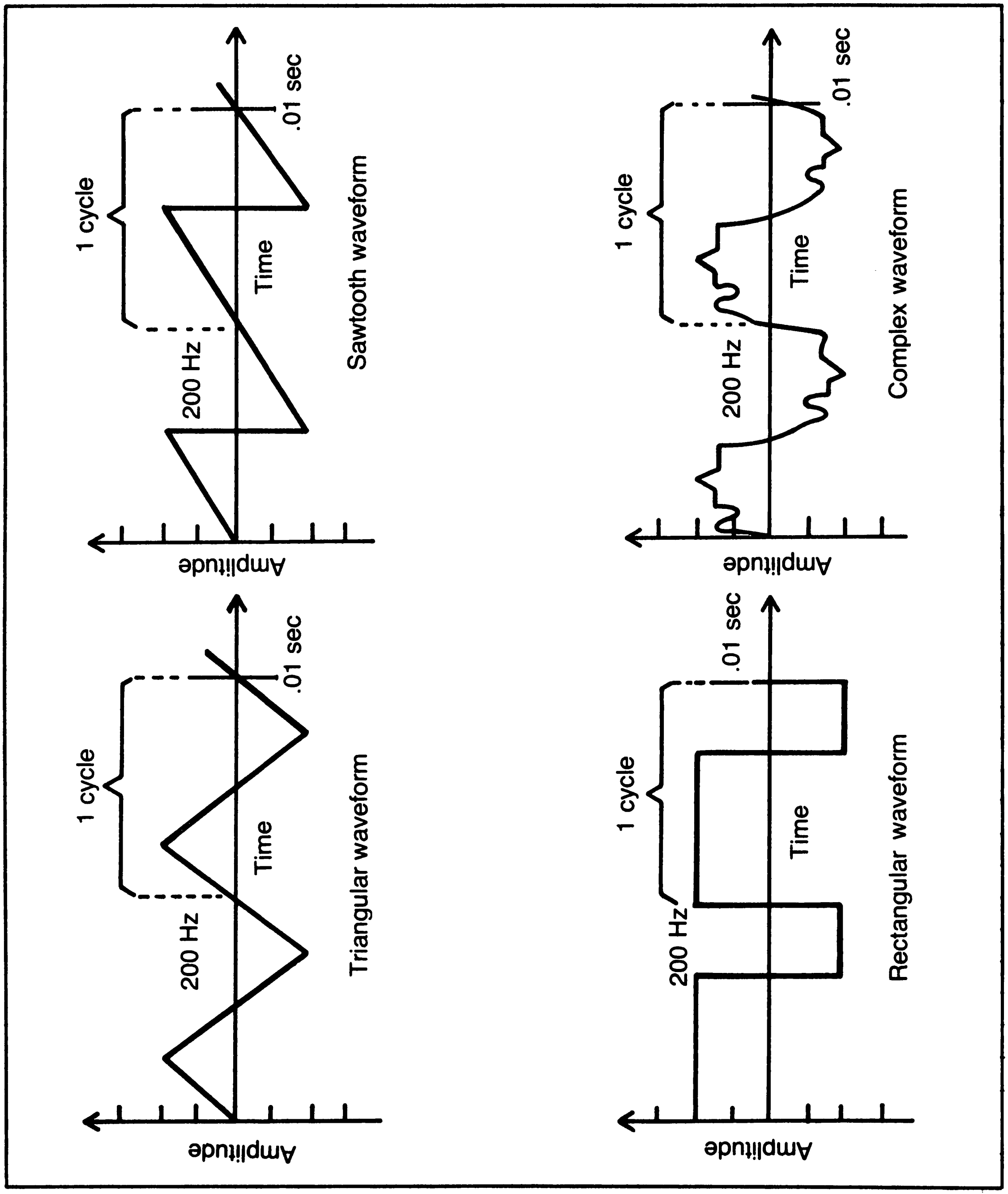



Fig. 7-4. Four more waveforms: triangular, sawtooth, rectangular, and complex.

### 7.1.4 Waveforms

Waves can have many different shapes. The waves shown in Figs. 7-2 and 7-3, created by tuning forks, are known as sine waves. The waves have regular, simple shapes. A particular wave shape is called a *waveform*.

Figure 7-4 shows four more waveforms: a triangular wave, a sawtooth wave, a rectangular wave, and a complex wave. Different waveforms create sounds with different tonal qualities, or timbres. A clarinet playing middle C at a certain volume sounds different from a piano playing the same role at the same volume. The clarinet's waveforms are different than the piano's.

Waveforms are independent of frequency and amplitude. If you look again at Fig. 7-4, you'll notice that I've drawn all four waves with the same frequency and amplitude.

## 7.2 BRIEF INTERLUDE

Your Commodore 64 can make a lot of different sounds. But this versatility has a price: complexity. It'll take us a while to learn how to set all the sound controls.

In the meantime, just to prove that the C-64 can produce sounds, run the short program listed in Fig. 7-5. When you tire of its haunting melody, press any key (other than the stop key) to end it. I'll resist the temptation to explain how this program works; once you learn enough about SID, you'll be able to figure it out on your own.

## 7.3 SID, THE SOUND INTERFACE DEVICE

You've been introduced to VIC-II, the Commodore 64's great graphics chip. Well, get ready to meet SID, the C-64's equally great sound chip. SID stands for Sound Interface Device. Commodore has put a sophisticated sound and music synthesizer onto a single integrated circuit chip. Let's go over some of SID's features.

To start with, SID actually has three separate sound synthesizers. They're also called voices. You can use any one, any two, or all three of these voices to create sounds.

There are a number of ways to control

```
1000 REM *** MINIMAL SIREN ***
1010 POKE 54296,15     :REM VOLUME ON HI
1020 POKE 54278,240    :REM SET SUSTAIN
1030 POKE 54276,33     :REM NOTE ON
1040 FOR N = 1 TO 100  :REM SIREEEN
1050 :    POKE 54273, 15 + ABS (50 - N)
1060 NEXT N
1070 GET KP$           :REM MORE ?
1080 IF KP$ = "" THEN 1040
1090 POKE 54276,0      :REM NOTE OFF
1100 POKE 54296,0      :REM VOLUME OFF
```

Fig. 7-5. Listing of the program Minimal Siren.

each voice. To begin with, each voice has a device called a *tone oscillator.* By setting the proper registers, you can make the tone oscillator produce sound waves at any frequency between 0 and 3995 hertz. That's about the same pitch range that pianos have.

Each voice also has a waveform generator. You can choose one of four waveforms for a voice: triangle, sawtooth, pulse, or noise. Triangular and sawtooth waves are shown in Fig. 7-4. Pulse is just another name for the rectangular waveform, also shown in Fig. 7-4. The noise waveform is a random signal that sounds like a TV set once all the stations have signed off. It comes in really handy for sound effects. It's also called *white noise.*

Finally, each voice has its own *envelope generator* and *amplitude modulator.* These strangely-named devices let you control the loudness of each voice in a very precise way. If you pluck a note on a guitar, you'll notice that the loudness changes throughout the life of that note. The envelope generator and amplitude modulator let you control the loudness of a SID voice in a similar way.

Each SID voice uses 7 registers. SID contains a total of 29 registers. The other eight registers let you control the overall loudness of all the voices, mix and synchronize the voices in funny ways, filter out certain frequencies, add in sounds from outside sources, read game paddles, and monitor the output of voice #3.

So much for a brief introduction to SID. Let's go into more detail about setting some of its registers.

## 7.4 GENERAL SID REGISTER LAYOUT

The 29 SID registers occupy memory locations 54272-54300. As I did with VIC, I'll usually refer to specific registers by their relative position in the register set. For example, the register at 54278 will be referred to as SID +6.

Appendix L shows the complete SID register layout. The first seven registers control voice #1, the next seven control voice #2, and the third set of seven control voice #3. The next four registers control filters and overall volume. The last four registers control miscellaneous functions.

I'll refer to the seven registers that control a voice as a *voice set.* The three voice sets are set up almost identically. I'll point out any exceptions as I go along.

## 7.5 SETTING A FREQUENCY

The first two registers of a voice set control that voice's frequency. That is, the registers at SID and SID+1 set the frequency for voice #1, SID+7 and SID+8 set it for voice #2, and SID+14 and SID+15 set it for voice #3.

Two 8-bit registers give a total of 16 bits. Values between 0 and 65535 can be represented with 16 bits. So, there are 65536 possible frequency settings for each voice.

How do you figure out the values to poke into the two frequency registers? First you do a little conversion. You divide the frequency in hertz by a special factor and then round it off to the nearest whole number. That'll give you the SID frequency setting. The special factor's based on the computer's clock speed. The factor is .060952, give or take a millionth. For example, say you want a frequency of 440 hertz. Rounding off 440 divided by .0609592 to the nearest whole number gives a frequency setting of 7218.

Now you have to convert the frequency setting into two values to poke into the frequency registers. Due to the complexities of bases 2, 10, and 16, you divide the setting by 256. The integer part goes into the second frequency register (SID+1, SID+8, or SID+15). It's known as the high byte of the frequency setting. The remainder from the division goes into the first frequency register (SID, SID+7, or SID+14). It's known as the low byte of the frequency setting.

Let's apply this second step to our 440 hertz tone. You got a frequency setting of 7218. Divide that by 256. The integer part of the answer is 28; the remainder is 50. If you want to set voice #1 so it produces a 440 hertz sound, you poke 28 into SID+1 and 50 into SID.

## 7.6 SETTING A WAVEFORM

The upper nibble—bits 4, 5, 6, and 7—of the fifth register in each voice set selects a waveform for that voice. SID+4 is the register used for voice #1, while SID+11 and SID+18 perform the chore for voices #2 and #3 respectively.

Setting one of these bits to 1 selects the waveform associated with that bit. Bit 4 selects a triangle wave; bit 5 selects a sawtooth wave; bit 6 selects a pulse (rectangular) wave; and bit 7 selects a white noise. See Fig. 7-6.

If you choose the pulse waveform, you need to set one more item: the pulse width. Let's see how that's done.

## 7.7 SETTING THE PULSE WIDTH

In a rectangular, or pulse, waveform, the amplitude is either high or low, with no intermediate values. The percentage of a wave cycle where the amplitude is high is known as the pulse width. Figure 7-7 shows pulse waveforms with four different pulse widths.

Registers 3 and 4 of a voice set control the pulse width if the pulse waveform is selected. What values do we poke into these two registers for a given pulse width? Take the pulse width (expressed as a percentage) and multiply by 40.95. Round that number off, and you've got the SID pulse width setting.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit value → | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Bit number → | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | Noise | Pulse | Saw-tooth | Triangle | – | – | – | – |

Fig. 7-6. Bits 4, 5, 6, and 7 of a voice's fifth register are used to select that voice's waveform.

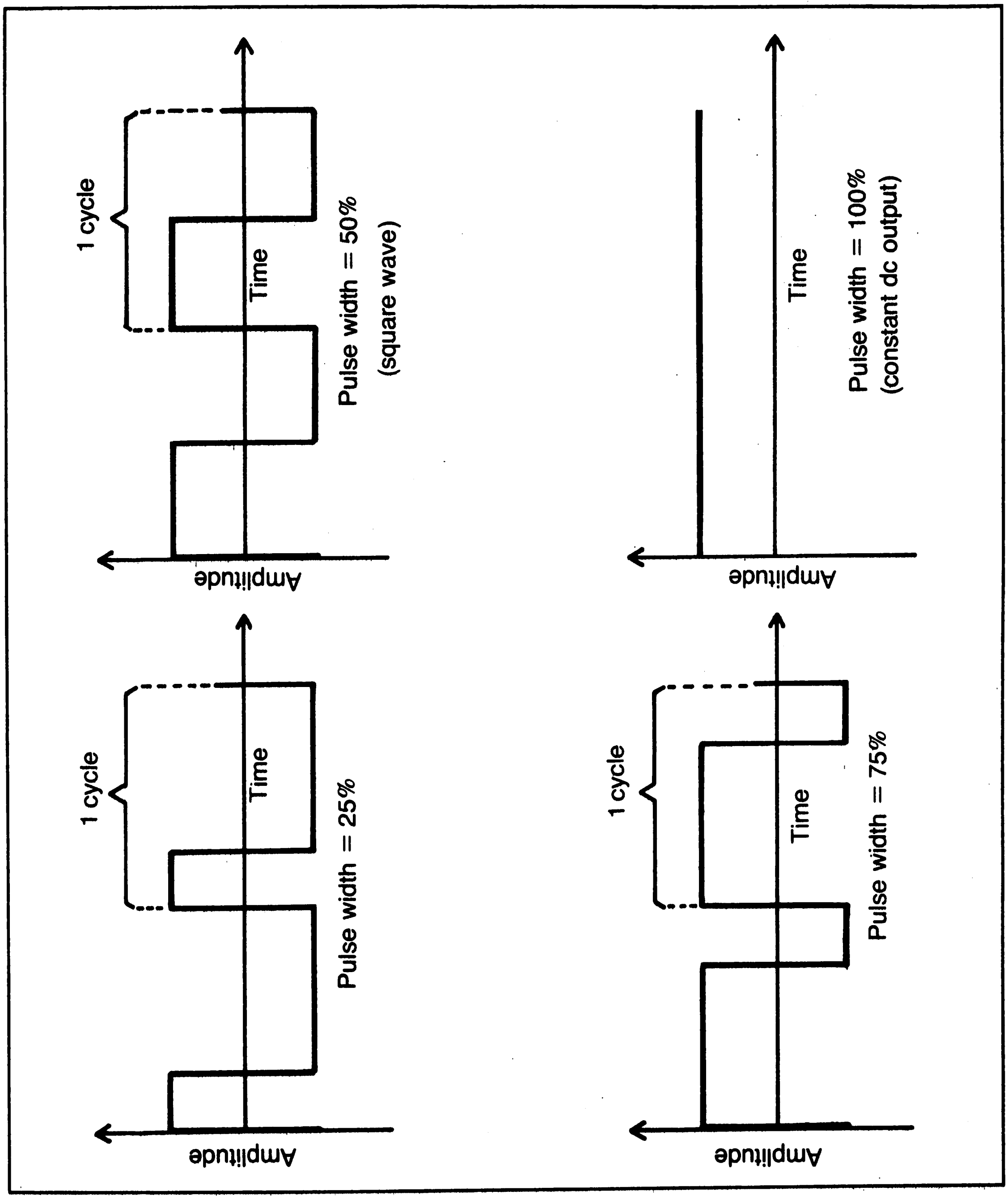



Fig. 7-7. Four pulse waveforms, each with a different pulse width.

Now divide the pulse width setting by 256. Poke the integer part of the result into the fourth register of the voice set. Put the remainder into the voice set's third register.

Here's an example. Let's say you want to set a pulse width of 75% for voice #3. 75 times 40.95 is 3071.25, which rounds off to 3071. 3071 divided by 256 gives 11, with a remainder of 255. So you'd put the value 11 into SID+17, and put the value 255 into SID+16.

## 7.8 SETTING A VOICE'S VOLUME VARIATIONS: THE ADSR ENVELOPE

Back in Section 7.3, I mentioned that each voice has an envelope generator and amplitude modulator. These devices give you precise control over volume during a sound's lifetime. The secret to this control is the ADSR envelope.

ADSR stands for *attack decay sustrain release*. These words define four stages of a typical sound's life. During the first stage, the volume goes from zero to a maximum value. The *attack rate* determines how long this rise in volume takes.

During the second stage, the volume drops from its maximum value to a lower level. The *decay rate* determines how long this drop takes.

The level that the volume drops to is called the *sustain level*. It can be expressed as a percentage of the maximum volume attained. During the third stage of the sound's life, volume stays at this level.

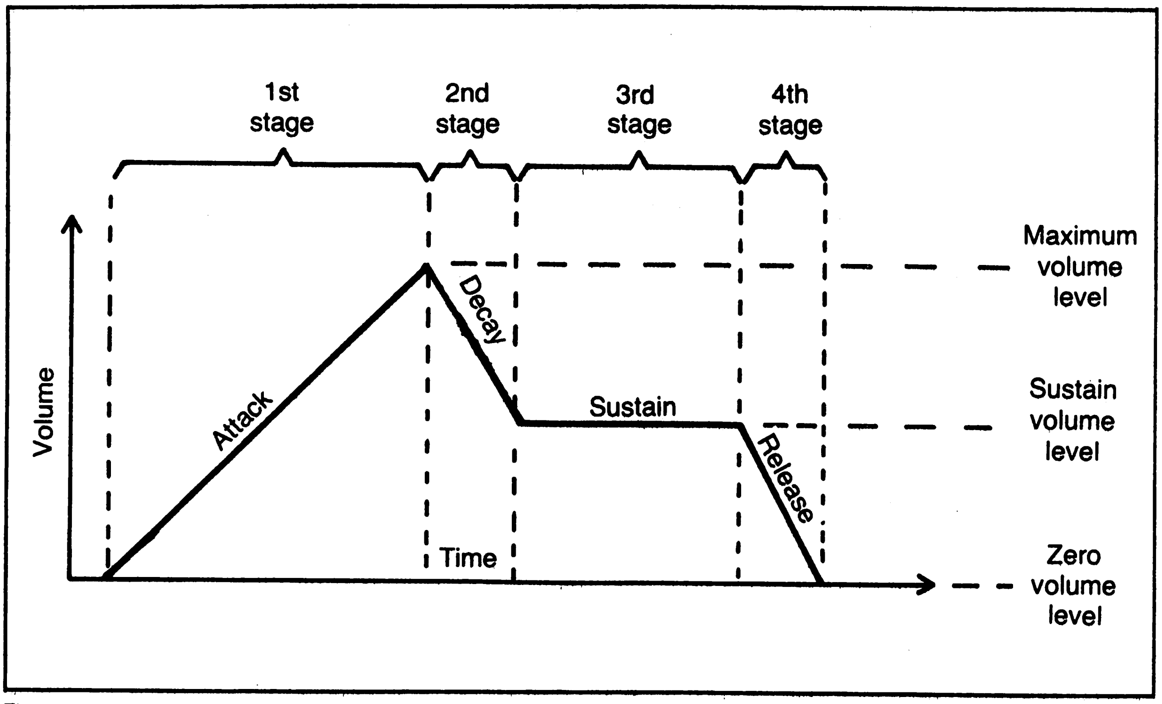



Fig. 7-8. The four stages of a typical note's life, showing the volume changes that make up the ADSR envelope.

Finally, the note stops. The rate at which it drops from the sustain level to zero volume is called the *release rate.*

Take a good look at Fig. 7-8. It shows the four stages of a typical note's life. Compare the picture to the description given above. Take the time to understand this concept. Can you see why the term ADSR envelope is used?

The sixth and seventh registers of each voice set define the ADSR envelope. When a voice is triggered, the values in these ADSR registers control the voice's envelope generator. In turn, the envelope generator controls the amplitude modulator. The amplitude modulator takes the waves coming from the tone oscillator and waveform generator and adjusts their amplitude. Figure 7-9 diagrams this process.

### 7.8.1 Setting Attack and Decay Rates

Values representing attack and decay rates are stored in the sixth register of each voice set. The attack rate value goes in the upper nibble, and the decay rate value goes in the lower nibble.

A nibble can store values from 0 through 15. Figure 7-10 shows how long it will take a sound to rise from zero to peak volume for the 16 different attack rate settings. For example, if the value of the nibble is 12, it'll take almost a full second for the volume to rise to its peak value.

Figure 7-11 shows rates of decay for the 16 possible nibble settings. They're shown as the time it will take a sound to fall from peak volume to zero volume. The time spent getting to a given sustain level will be based on these

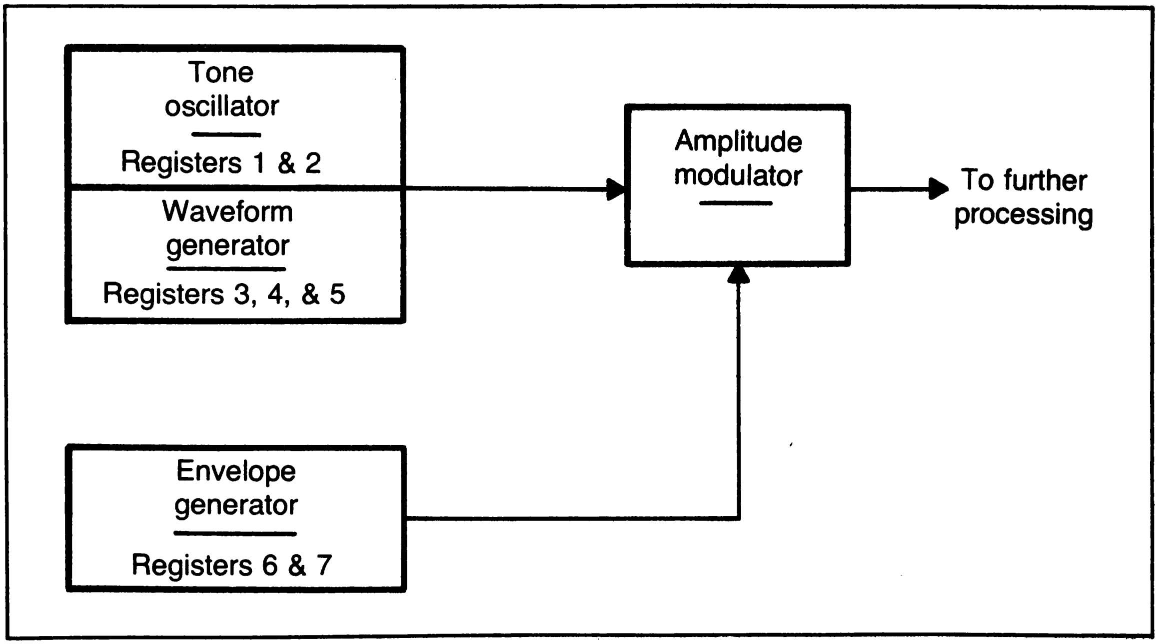



Fig. 7-9. Information from a voice's tone oscillator, waveform generator, and envelope generator comes together at the amplitude modulator; the resulting signal then goes on for final SID processing.

Attack rates 

| Nibble value | Seconds to go from zero to peak volume | Nibble value | Seconds to go from zero to peak volume |
|---|---|---|---|
| 0 | .002 | 8 | .098 |
| 1 | .008 | 9 | .244 |
| 2 | .016 | 10 | .489 |
| 3 | .023 | 11 | .782 |
| 4 | .037 | 12 | .978 |
| 5 | .055 | 13 | 2.933 |
| 6 | .066 | 14 | 4.889 |
| 7 | .078 | 15 | 7.822 |

Fig. 7-10. The 16 attack rates built into SID and selected by the upper nibble of a voice's sixth register.

rates. For example, let's set the sustain level to 80% of peak volume, and the decay value to 6. Using these values, it will take 20% of .199, or about .04 seconds, for the volume to drop from its peak to the sustain level.

Once you've picked values for the attack and decay rates, you need to figure out the value to poke into the register. Just multiply the attack value by 16 and then add in the decay value. For example, set the attack value for voice #1 to 12 and the decay value to 6. 12 times 16 is 192, and adding 6 gives 198. So you'd poke the value 198 into the attack/decay register at SID+5.

### 7.8.2 Setting the Sustain Levels and Release Rate

Values representing the sustain level and release rate are stored in the seventh register of each voice set. The upper nibble holds the sustain value, and the lower nibble holds the decay value.

Sustain levels are set at a percentage of the peak volume. Figure 7-12 shows the percentages for the 16 possible nibble values. For example, setting a sustain level of 9 means the sound will drop to 60% of its peak volume. Setting a sustain level of 15 will hold the volume at its peak value.

## Decay Rates

| Nibble value | Seconds to go from peak volume to zero | Nibble value | Seconds to go from peak volume to zero |
|---|---|---|---|
| 0 | .006 | 8 | .293 |
| 1 | .023 | 9 | .733 |
| 2 | .047 | 10 | 1.467 |
| 3 | .070 | 11 | 2.347 |
| 4 | .111 | 12 | 2.933 |
| 5 | .164 | 13 | 8.800 |
| 6 | .199 | 14 | 14.667 |
| 7 | .235 | 15 | 23.467 |

Fig. 7-11. The 16 decay rates built into SID and selected by the lower nibble of a voice's sixth register.

## Sustain Levels

| Nibble value | % of peak volume | Nibble value | % of peak volume |
|---|---|---|---|
| 0 | 0.0 | 8 | 53.3 |
| 1 | 6.7 | 9 | 60.0 |
| 2 | 13.3 | 10 | 66.7 |
| 3 | 20.0 | 11 | 73.3 |
| 4 | 26.7 | 12 | 80.0 |
| 5 | 33.3 | 13 | 86.7 |
| 6 | 40.0 | 14 | 93.3 |
| 7 | 46.7 | 15 | 100.0 |

Fig. 7-12. The 16 sustain levels built into SID and selected by the upper nibble of a voice's seventh register.

Release Rates 

| Nibble value | Seconds to go from peak volume to zero | Nibble value | Seconds to go from peak volume to zero |
|---|---|---|---|
| 0 | .006 | 8 | .293 |
| 1 | .023 | 9 | .733 |
| 2 | .047 | 10 | 1.467 |
| 3 | .070 | 11 | 2.347 |
| 4 | .111 | 12 | 2.933 |
| 5 | .164 | 13 | 8.800 |
| 6 | .199 | 14 | 14.667 |
| 7 | .235 | 15 | 23.467 |

Fig. 7-13. The 16 release rates built into SID and selected by the lower nibble of a voice's seventh register.

Release rates are shown in Fig. 7-13. This chart is just like Fig. 7-11, which showed decay rates. The times shown tell how long it'll take a sound to fall from peak volume to zero volume. The actual time a sound will spend falling from the sustain level to zero volume is based on these rates. For example, say the sustain level is 50% of peak volume, and you choose a release value of 10. Then it'll take 50% of 1.467, or .733 seconds, for the volume to drop to zero.

Once you pick values for sustain and release, just multiply the sustain value by 16 and add the release value. That's the number to poke into the seventh register. For example, assume you choose a sustain value of 3 and a release value of 11 for voice #2. 3 times 16 is 48, and adding 11 gives 59. Which is the value to poke into the register at SID+13.

## 7.9 TURNING A SOUND ON AND OFF: GATING THE ENVELOPE GENERATOR

The fifth register of each voice set is a waveform controller. As you saw in Section 7.6, its upper nibble is used to select a waveform. Bit 0 of these registers is used to turn a sound on and off. It does this by gating, or triggering, the voice's envelope generator. It's called a gate bit.

Setting a gate bit to 1 tells that voice's envelope generator to start an ADSR cycle. The volume rises from zero to its peak value and then falls to the sustain level. It stays there until the gate bit is reset to 0. When that happens, it triggers the release action, and volume falls to zero.

When you're writing sound programs in BASIC, it's a good idea to combine choosing a waveform with gating the envelope generator. For example, poking SID+4 with the value 17 will select the triangle waveform and start an ADSR cycle. Poking SID+4 with 16 will keep the triangle waveform selected and start the release part of the ADSR cycle. Figure 7-14 shows poking values that'll trigger and release a sound.

## 7.10 THE MASTER VOLUME CONTROL

Let's review a bit. SID has three voices. Each voice has its own tone oscillator and waveform generator, which produce waveforms at set frequencies. These signals go to the voice's amplitude modulator, where the volume gets modified. Each voice uses an envelope generator to control its amplitude modulator.

The signals from the three voices then go to an overall volume control. This device mixes the voices together and sets SID's overall output volume. Sometimes a voice will make a detour to a filtering device on its way to the overall volume control, but you don't need to think about that right now.

| Waveform | Poke this value to trigger | Poke this value to release |
|---|---|---|
| Triangle | 17 | 16 |
| Sawtooth | 33 | 32 |
| Pulse | 65 | 64 |
| Noise | 129 | 128 |

Fig. 7-14. Values to poke into a voice's fifth register to trigger or release the ADSR envelope while selecting a waveform.

Bits 0-3 of the register at SID+24 set the overall volume. It can be set to any value between 0 and 15. A setting of 15 gives maximum volume, while a setting of 0 leads to no output.

That concludes this preliminary look at SID. Let's now take a quick look at musical note frequencies and then close up with a musical program.

## 7.11 THE FREQUENCIES OF MUSICAL NOTES

Most of our culture's music is based on scales that contain twelve notes: C, C#, D, D#, E, F, F#, G, G#, A, A#, and B. A twelve note scale forms an octave. As you move up from one octave to the next, the frequencies double. That is, if an A note in one octave has a frequency of 440 hertz, the A note in the next octave up will have a frequency of 880 hertz.

As you move from one note to the next within a scale, the frequency is the 12th root of 2 times the previous note's frequency. That way, after 12 notes (an octave) the frequency doubles.

In a standard scale, known as concert pitch, the A note in the fourth octave is set to 440 hertz. Once that value is known, all the other frequencies can be figured.

Appendix M gives frequencies in hertz for eight octaves of musical notes, based on concert pitch. It also gives the SID frequency setting for each note, and breaks that setting up into a high and a low byte.

Let's say you want voice #1 to produce a C note in the fourth octave (also known as middle C). According to the chart, that note has a frequency of 261.6 hertz. By poking 16 into SID+1, and 195 into SID, you can set voice 1 to produce notes at that pitch.

## 7.12 FINALLY: A LITTLE MUSIC

Now you're ready to put all of our SID knowledge to work. Figure 7-15 lists the program Play Some Sounds. Type it in, save it, and then run it. It uses voice #1 to play a scale.

Let's go over the program. The first segment clears the screen and sets up two variables: SID's starting address, and the factor used to convert frequencies in hertz to SID frequency settings.

The next segment sets up attack, decay, sustain, and release values. The notes will rise quickly to peak volume, stay there until the gate bit is reset, then fall quickly to zero volume.

Next, the overall volume level is set. You also choose a duration for each note: ¼ second. That's how long you'll let the note go before triggering the release stage of an ADSR cycle.

The next segment reads frequencies from the data statements and converts them into values to poke into the frequency registers at SID and SID+1. Review Section 7.5 if you're wondering where all the formulas come from. The program will end when a frequency of 0 gets read.

You've set the ADSR envelope, overall volume, and frequency. Now it's time to play the note. Line 1480 pokes SID+4 with a value that sets the waveform and triggers the envelope generator. Volume rises to a peak, decays to the sustain level, and then sits there while a delay loop marks time. Line 1530 initiates the release period, and volume drops to zero. Then it's back for another note.

All right, now it's your turn. Fiddle mer-

```
1000 REM *** PLAY SOME SOUNDS ***
1010 :
1020 :
1030 REM ** SET UP SCREEN & VARIABLES
1040 :
1050 PRINT "♥";
1060 SID = 54272
1070 CNF = .0609592
1080 :
1090 :
1100 REM ** SET ADSR ENVELOPE
1110 :
1120 ATK = 0                    :REM QUICK
1130 DKY = 0                    :REM QUICK
1140 AD = ATK*16 + DKY          :REM COMBINE
1150 POKE SID+5, AD             :REM SETIT
1160 :
1170 SST = 15                   :REM TOP VOL
1180 RLS = 0                    :REM SPEEDY
1190 SR = SST*16 + RLS          :REM COMBINE
1200 POKE SID+6, SR             :REM SETIT
1210 :
1220 :
1230 REM ** SET DURATION & MASTER VOLUM
1240 :
1250 DUR = 1/4              :REM IN SECONDS
1260 VOL = 15               :REM TOP VOLUME
1270 POKE SID+24, VOL       :REM SET IT
1280 :
1290 :
1300 REM ** SET WAVEFORM & FREQUENCY
1310 :
1320 WAVFRM = 16               :REM TRIANGLE
1330 :
1340 READ FRQ                  :REM IN HERTZ
1350 IF FRQ = 0 THEN 1590
1360 FRQ = INT(FRQ/CNF)       :REM CONVERT
1370 FHI = INT (FRQ/256)      :REM HI-BYTE
1380 FLO = FRQ - FHI*256      :REM LO-BYTE
1390 POKE SID, FLO            :REM SET IT
1400 POKE SID+1, FHI
1410 :
1420 DATA 261.6, 293.7, 329.6, 349.2
```

```
1430 DATA 392.0, 440.0, 493.9, 523.3, 0
1440 :
1450 :
1460 REM ** PLAY THE NOTE, THEN GO BACK
1470 :
1480 POKE SID+4, WAVFRM + 1
1490 :
1500 FOR TM = 1 TO (DUR * 700)
1510 NEXT TM
1520 :
1530 POKE SID+4, WAVFRM
1540 GOTO 1340
1550 :
1560 :
1570 REM ** CLEAN UP AND END
1580 :
1590 POKE SID+24, 0      :REM VOLUME OFF
1600 :
1610 END
```

Fig. 7-15. Listing of the program Play Some Sounds.

cilessly with this program. Change the frequencies, the ADSR envelope, the overall volume, the waveform—anything you can think of. There aren't any magic formulas to sound making, you've just got to experiment. Try to get an intuitive feel for various SID settings. Have fun.

### 7.13 CHAPTER SUMMARY

This chapter has introduced you to sound making on the Commodore 64. Let's see what we've covered:

* Sounds, vibrations, and waves
* Frequency, amplitude, and waveforms
* SID's three voices, and the devices that create each one: the tone oscillator, waveform generator envelope generator, and amplitude modulator
* The general layout of SID's 29 registers
* How to set a voice's frequency, waveform, pulse width, and ADSR envelope
* How to turn a voice on and off by gating its envelope generator
* How to set an overall volume level
* How the frequencies of musical notes are determined
* How to use all of this information in a program to create sounds

SID's power and versatility make sound production as endless a field for invention as VIC-II does with graphics. In the next chapter, you'll look at more programs that use SID to make music.

## 7.14 EXERCISES

### 7.14.1 Self Test

Answers are in Section 7.14.3.

1. (7.1) Three ways to describe a sound wave are by its ________, its ________, and its ________.
2. (7.3) SID has ________ separate voices.
3. (7.4) The registers from SID+7 through SID+13 control voice # ________.
4. (7.5) By poking SID with the value 16 and SID+1 with the value 39, we give voice# ________ a frequency of ________ hertz.
5. (7.6) Setting bit 7 of SID+18 to 1 selects the ________ waveform for voice# ________.
6. (7.7) To give voice #3 a pulse width of 20%, you'd poke SID+17 with the value ________ and SID+16 with the value ________.
7. (7.8.1) If a voice's attack rate setting is 3, it'll take ________ seconds to go from zero to peak volume.
8. (7.8.2) To give voice #1 a sustain level that's 40% of its peak volume and the slowest available release rate, you'd poke the value ________ into SID + ________.
9. (7.9) Bit 0 of each voice set's fifth register is used to trigger that voice's ________ generator.
10. (7.10) Overall SID output volume is set by the lower four bits of the register at ________.
11. (7.11) If a 7th octave C note has a frequency of 2093 hertz, an 8th octave C note will have a frequency of ________ hertz.

### 7.14.2 Programming Exercises

These are pretty open-ended: play, play, play!

1. Change the program Minimal Siren so it sounds like something from outer space.
2. Change the program Play Some Sounds so it glides up and down the scale until you press a key.
3. Change the program Play Some Sounds so voice #2 joins in. Have voice #2 play sounds a few notes away from voice #1.

### 7.14.3 Answers to Self Test

As usual, note that you may be able to come up with better answers.

1. frequency (pitch); amplitude (loudness or volume); waveform (timbre)
2. three
3. 2
4. 1; 10000
5. noise; 3
6. 3; 51
7. .023
8. 111; 6
9. envelope
10. SID+24 (54296)
11. 4186

## 7.14.4 Possible Solutions to Programming Exercises

1. Load in the program Minimal Siren. Then type in these lines:

```
1000 REM *** FROGS FROM MARS ***
1020 POKE 54278,164    :REM SET SUSTAIN
1030 POKE 54276,17     :REM NOTE ON
1040 FOR N = 1 TO 30   :REM FROG CITY
1050 :    POKE 54273, 1 + N*8
1051 :    POKE 54273, 1+N
1052 :    POKE 54273, 50 - N
```

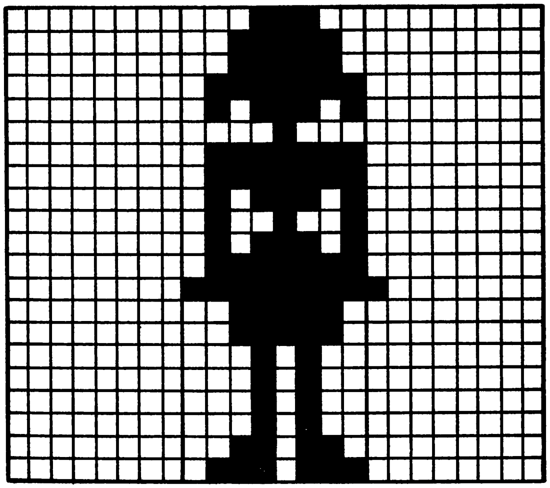
2. Load in the program Play Some Sounds. Then type in these lines:

```
1000 REM *** ROLLER COASTER ***
1250 DUR = 1/50          :REM IN SECONDS
1350 IF FRQ = 0 THEN RESTORE: GOTO 1340
1430 DATA 392.0, 440.0, 493.9, 523.3
1433 DATA 523.3, 493.9, 440.0, 392.0
1436 DATA 349.2, 329.6, 293.7, 261.6, 0
1513 :
1515 GET KP$
1517 IF KP$ <> "" THEN 1590 :REM END IT
```

3. Load in the program Play. Some Sounds. Then type in these lines.

```
1000 REM *** TWO-VOICE SOUNDS ***
1155 POKE SID+12, AD        :REM SET V-2
1205 POKE SID+13, SR        :REM SET V-2
1363 V2FAC = 2↑(5/12)       :REM HARMNY?
1365 FRQ(2) = FRQ * V2FAC :REM V2 FQ
1402 FHI(2) = INT(FRQ(2)/256)  :REM V2
1404 FLO(2) = FRQ(2) - FHI(2)*256
1406 POKE SID+7, FLO(2)    :REM V-2 LO-F
1408 POKE SID+8, FHI(2)    :REM V-2 HI-F
1485 POKE SID+11, WAVFRM + 1
1535 POKE SID+11, WAVFRM
```

# Chapter 8



# Some Fancy Music Making

In the last chapter you learned about SID, the Commodore 64's versatile sound chip. Now you'll use this knowledge to make some interesting music. You'll teach the computer to read notes and store the information in a performance array. Then you'll play the notes through one of SID's voices. Finally, you'll extend these techniques to music that uses all three voices.

## 8.1 READING MUSIC

In the program Play Some Sounds, from the last chapter, you specified musical notes by their frequencies. The program used that value to figure SID settings. Let's make things easier by getting a program to play notes specified by letter names, C, G#, etc., and octave numbers. You'll need a reference table similar to Appendix M in our program. Then you can have the program read a note by letter and octave, look up its SID frequency setting in the table, and use that value to poke the SID registers. But Appendix M is pretty long. Who wants to do all that typing? Let's take a shortcut.

### 8.1.1 Typing Shortcut: Using a Reference Octave

In the last chapter, I mentioned that frequencies double as you move up an octave. For example, an A note in the fourth octave has a frequency of 440 hertz, which is twice the 220 hertz frequency of an A note in the third octave.

You can use this fact. You'll make a reference table that has the SID frequency settings for the twelve notes in the highest octave, octave 7. When the program reads a note, it will see how many octaves it is below the highest octave. Then it will divide the reference setting by 2 for each octave of difference, and round the final result to the nearest whole number. Once you have this frequency setting, you'll just divide it by 256. The integer part of the answer is the high byte of the frequency setting, and the remainder is the low byte.

Here's an example. Let's say the program reads a note that's a second octave F#. That's five octaves below the highest octave. The SID frequency setting for a seventh F# is 48557. Dividing that value by 2 gives you 24278.5. After four more divisions, you end up with the value 1517.4062, which rounds off to 1517. Dividing by 256, you get 5 for the high byte of the setting and 237 for the low byte. Checking with Appendix M, you see that this method has given us the correct values. Figure 8-1 shows the letter names of the twelve notes in the seventh octave, along with their frequencies in hertz and the corresponding SID frequency settings.

To create music you now need to specify a note name and octave number for each note. You can do this with strings. For example, you can represent a fifth octave G# as

G#-5

A program can use string functions to extract the note name and octave from data stored in this form.

| Note | Frequency in hertz | SID frequency setting |
|---|---|---|
| C | 2093.0 | 34334 |
| C# | 2217.5 | 36377 |
| D | 2349.3 | 38539 |
| D# | 2489.0 | 40831 |
| E | 2637.0 | 43258 |
| F | 2793.8 | 45831 |
| F# | 2960.0 | 48557 |
| G | 3136.0 | 51444 |
| G# | 3322.4 | 54502 |
| A | 3520.0 | 57743 |
| A# | 3729.3 | 61177 |
| B | 3951.1 | 64815 |

Fig. 8-1. The twelve notes of the seventh octave, to be used as a reference octave.

### 8.1.2 Note Durations

In the program Play Some Sounds every note lasted for the same amount of time. This gets boring. You can include a duration number for each note in a program's data statements.

Let's take a hint from written music and set up a standard duration, called a beat. Then each note's duration can be given as a number of beats. For example, you can represent an F note in the third octave that lasts for four beats as a string and an integer:

F-3,4

How will the program make one note last for two beats, and another last for three? There are a number of ways to do this. One of the most flexible is to use what I call performance arrays.

## 8.2 PERFORMANCE ARRAYS: A GUIDE TO EVERY BEAT

A performance array holds a SID value for each beat of a song. A program might have a number of different performance arrays. One array could hold the low bytes for voice #1's frequency setting, and another could hold the high bytes. A third array could hold values for voice #1's attack/decay register.

When it comes time for the program to play all the notes, it will simply go through a beat loop. Each time through the loop, that beat's various SID settings will be pulled from the performance arrays and poked into place.

```
1000 REM *** READ MUSIC ***
1010 :
1020 :
1030 REM ** SET UP SCREEN & VARIABLES
1040 :
1050 PRINT "{CLR}";          :REM CLEAR SCREEN
1060 PRINT "{12 DOWN}READING";
1070 :
1080 SID = 54272         :REM SOUND CHIP
1090 :
1100 :
1110 REM ** SET UP REFERENCE ARRAYS
1120 :
1130 DIM SBN(11), NM$(11) :REM BASED ON
1140 FOR N = 0 TO 11       :REM NOTES IN
1150 :    READ SBN(N)      :REM HIGHEST
1160 :    READ NM$(N)      :REM OCTAVE
1170 NEXT N
1180 :
1190 DATA 34334, C,  36377, C#
1200 DATA 38539, D,  40831, D#
1210 DATA 43258, E,  45831, F
1220 DATA 48557, F#, 51444, G
1230 DATA 54502, G#, 57743, A
1240 DATA 61177, A#, 64815, B
1250 :
1260 :
```

```
1270 REM ** READ IN THE MUSIC AND
            STORE IT IN ARRAYS
1280 :
1290 DIM LFP(200), HFP(200)
1300 :
1310 EVENT = 1
1320 :
1330 READ NC$
1340 PRINT ".";
1350 IF NC$ = "XXX" THEN 1670
1360 :
1370 GOSUB 2050 :REM CONVERT TO POKE #S
1380 :
1390 READ DUR
1400 FOR N = 1 TO DUR
1410 :   LFP(EVENT) = LFP
1420 :   HFP(EVENT) = HFP
1430 :   EVENT = EVENT + 1
1440 NEXT N
1450 :
1460 GOTO 1330
1470 :
1480 :
1490 REM ** THE MUSIC : NOTE-OCT, DUR
1500 :
1510 DATA B-4, 4, D-5, 4, C-5, 8
1520 DATA RES, 1, B-4, 4, D-5, 4
1530 DATA A-4, 8, RES, 1, B-4, 4
1540 DATA D-5, 4, C-5, 4, B-4, 2
1550 DATA C-5, 2, D-5, 4, A-4, 4
1560 DATA G-4, 8, RES, 2, B-4, 4
1570 DATA G-4, 4, C-5, 8, RES, 1
1580 DATA B-4, 4, G-4, 4, A-4, 8
1590 DATA RES, 1, B-4, 4, D-5, 4
1600 DATA C-5, 8, RES, 1, B-4, 2
1610 DATA C-5, 2, D-5, 4, A-4, 4
1620 DATA G-4,10, XXX
1630 :
1640 :
1650 REM ** SET ADSR, VOLUME, WAVEFORM
1660 :
1670 ATK = 0          :REM QUICK ATTACK
1680 DKY = 0          :REM QUICK DECAY
1690 AD = ATK*16 + DKY
1700 POKE SID+5, AD
1710 :
1720 SST = 15         :REM SUSTAIN LOUD
1730 RLS = 0          :REM QUICK RELEASE
1740 SR = SST*16 + RLS
1750 POKE SID+6, SR
1760 :
```

```
1770 VLM = 15          :REM MAX VOLUME
1780 POKE SID+24, VLM
1790 :
1800 WVFRM = 16        :REM TRIANGLE WAVE
1810 :
1820 :
1830 REM ** PLAY THE MUSIC, THEN END IT
1840 :
1850 PRINT "♥";
1860 BEATLNGTH = 10
1870 :
1880 FOR N = 1 TO (EVENT - 1)
1890 :    POKE SID+1, HFP(N)
1900 :    POKE SID, LFP(N)
1910 :
1920 :    POKE SID+4, WVFRM + 1   :REM ON
1930 :    FOR TM = 1 TO BEATLNGTH
1940 :    NEXT TM
1950 NEXT N
1960 :
1970 POKE SID+4, 0   :REM WAVEFORM OFF
1980 POKE SID+24,0   :REM VOLUME OFF
1990 END
2000 :
2010 :
2020 :
2030 REM ** CONVERT NOTE-OCTAVE STRING
            TO LO AND HI POKE CODES
2040 :
2050 IF NC$ = "RES" THEN HFP = 0 :
                          LFP = 0 : RETURN
2060 :
2070 NT$ = LEFT$(NC$, LEN(NC$) - 2)
2080 FOR REF = 0 TO 11
2090 :    IF NT$ = NM$(REF) THEN
                   NT = REF : REF = 11
2100 NEXT REF
2110 :
2120 OCT = VAL(RIGHT$(NC$,1))
2130 :
2140 FST = 2 ↑ (7 - OCT)
2150 FST = SBN(NT) / FST
2160 HFP = INT (FST/256)
2170 LFP = INT (FST - 256*HFP)
2180 :
2190 RETURN
```

Fig. 8-2. Listing of the program Read Music.

There will be a short time delay, the length of one beat, and then the program will loop back to deal with the next beat.

A note that lasts for one beat will have one entry in each performance array. A note with a longer duration will have as many entries as it has beats.

Here's an example. Let's say one of our performance arrays stores values for the high byte of voice #1's frequency setting. If a song's first note is a fourth octave D that lasts for three beats, and the second note is a fifth octave F# that lasts two beats, the array would start with these five values:

HF(1) = 18
HF(2) = 18
HF(3) = 18
HF(4) = 47
HF(5) = 47

There are a number of advantages to performance arrays. Since all the SID values are figured before any notes are played, notes can follow one another smoothly, with no delays for lengthy calculations. And since the basic timing unit is a beat, it's easy to have different voices play notes of different lengths, as you'll see later in this chapter. Right now, it's time to move from theory to practice. Let's see how note reading and performance arrays are actually used in a program.

## 8.3 A PROGRAM THAT READS MUSIC AND PLAYS IT BY THE BEAT

Figure 8-2 lists the program Read Music, which uses the ideas discussed above. Read it over; type it in; save it; then run it. If you want to listen to it again, without waiting for the music to be read into the performance arrays, just type in this command:

```
GOTO 1670
```

By the way, the melody this program plays is an old English tune called "Shepherd's Hey."

### 8.3.1 About the Program

Let's go over this program in detail. Lines 1050-1080 clear the screen, print a feedback prompt, and set up SID's starting address. The next module sets up two reference arrays. The SBN array contains the twelve SID frequency settings for the seventh octave, and the NM$ array contains the twelve corresponding note names.

The next segment actually reads the notes and fills the performance arrays. In this case, you've got one performance array that'll hold the low bytes of frequency settings, and one that'll hold the high bytes. Line 1330 reads in a note/octave string, and then line 1340 gives a bit of screen feedback. Line 1350 checks for the string that signals the end of the note/octave data. If it finds it, the note reading is over, and the program goes on to set the ADSR envelope.

Line 1370 jumps to a subroutine that'll take the note/octave string and figure out the appropriate low and high bytes for a SID frequency setting. Let's see how the subroutine works.

### 8.3.2 Decoding The Note/Octave String

Line 2050 first checks for the special string value RES, which stands for a rest. A rest is a pause in the music. A silent note, really. Setting the SID frequency registers to 0 is one way to create silence.

Line 2070 picks the note name out of the string. Then lines 2080-2100 try to match the note name with names from the reference array NM$. When there's a match, the program stores the note's number in the variable NT. This number will be used to pick the appropriate SID reference frequency out of the array SBN.

Line 2120 picks the octave number out of the string. Then line 2140 uses this number to figure out what the reference frequency setting should be divided by. Line 2150 does the division. Finally, lines 2160-2170 figure out the high and low bytes that'll give this setting. The conversion is complete, and the subroutine returns to line 1380.

### 8.3.3 Filling the Performance Arrays

Now it's time to add to the performance arrays. Remember, you've got to enter information for each beat. Line 1390 reads the note's duration, expressed as a number of beats. Lines 1400-1440 then use this value to control a loop that packs the two performance arrays. The body of the loop will be executed once for each beat of the note. Each time through, the low and high bytes of the note's frequency setting get stored in the arrays, and then the beat number increases by 1.

Is this confusing? Let's look at it from another angle. What we're really doing is making copies of a note's settings. As many copies as the number of beats to the note. When it comes time to perform the piece, the program will just grab SID settings a beat's worth at a time.

### 8.3.4 The Music Itself

Lines 1510-1620 store the music itself. The string XXX signals the end of the information. If you want to change the song this program plays, you just need to change these data lines. You can take songs from books on music or make up your own.

If you take songs from music books, you'll have to know how to read music. It's really not too difficult a skill to pick up. If you'd like to read a good book on the subject, try *Henscratches and Flyspecks*, by Pete Seeger, published by G.P. Putnam's Sons. Most libraries have it.

### 8.3.5 Set ADSR and Waveform; Then Play the Tune

Lines 1670-1750 set the attack, decay, sustain, and release values for voice #1. Lines 1770-1780 set an overall volume level, and line 1800 sets up the waveform that'll be used. I designed these lines so it'd be easy to go in and make changes.

Finally, everything is ready. The curtain rises, and the conductor readies her baton (lines 1850-1860). The loop in lines 1880-1950 plays the music, one beat at a time. Each time through the loop, that beat's frequency settings get poked in. Then line 1920 triggers the amplitude modulator, which begins the ADSR cycle.

For the sake of simplicity, I played a bit of a trick here. The performance loop never triggers the release part of the volume envelope.

```
1510 DATA G-4, 4, E-4, 4, G-4, 4
1520 DATA C-5, 4, D-5, 2, E-5, 2
1530 DATA D-5, 4, B-4, 2, C-5, 2
1540 DATA B-4, 4, RES, 1, G-4, 4
1550 DATA E-4, 4, G-4, 4, C-5, 4
1560 DATA D-5, 2, E-5, 2, D-5, 4
1570 DATA B-4, 4, C-5, 4, XXX
```

Fig. 8-3. Changes to Read Music that teach it to play a different tune.

The notes slur together a bit. Try running the program with this line added:

```
1945 :    POKE SID+4, WAVFRM :REM RELEAS
```

Notice how notes longer than one beat get chopped up if you trigger a release stage at the end of each beat. Is there a way to avoid both slurring and chopping? Yes, and you'll get to see the technique later in this chapter.

Finally, lines 1970-1980 turn the waveform and overall volume controls off, and the program ends.

Once again the ball's in your court. Have this program play a different tune. Or make it play at different speeds. See what happens when two or more notes of the same pitch follow one another.

If you can't read music, find a friend who can. Or just make up notes in pleasing patterns. Or type in the data statements shown in Fig. 8-3.

## 8.4 THINKING ABOUT THREE VOICES AND DISTINCTION

There are two improvements you can make to programs like Read Music. First, you can get SID's two other voices into the act. Second, you can find a way to make each note more distinct, without slurring or choppiness.

Both of these are easily done with performance arrays. Let's look at the first improvement. In Read Music, you stored voice #1 frequency information for each beat of the music. You'll just add similar frequency information for the other two voices. You'll store the information in two-dimensional performance arrays. They'll take on the form

ARRAYNAME (voice #, beat #)

Here are some examples of what I mean, using the array names from Read Music:

LFP (1,20) holds the low byte of voice #1's frequency setting for the 20th beat

HFP (3,80) holds the high byte of voice #3's frequency setting for the 80th beat

HFP (2,1) holds the high byte of voice #2's frequency setting for the first beat

Now, on to the second improvement. You want to make each note more distinct. In Read Music, the performance loop just triggered the start of an ADSR cycle, and never dealt with triggering the release stage; but adding a release stage to each beat chopped things up too much.

One thing you can do is trigger a release stage on the last beat of a note. That is, if a note lasts four beats, the first three beats will each trigger the start of an ADSR cycle, and the last beat will trigger the release stage. It's not a totally perfect solution, but it works pretty well. More importantly, it's surprisingly easy to program. You just create a new performance array for waveform control. It'll contain entries for each voice for each beat. These entries will be values to poke into each voice's waveform control register.

Here's an example. Let's say that voice #1 starts off playing a note that lasts for three beats. Assume you select the triangle waveform for voice #1. Name the wave control array WVC. Then WVC(1,1) will contain the value 17. WVC(1,2) will contain the value 17. WVC(1,3) will contain the value 16. The values for the note's first two beats will trigger the start of an ADSR cycle. The value for the note's last beat will trigger the release stage of the cycle.

## 8.5 A THREE VOICE EXAMPLE

Figure 8-4 lists the program Three-Part

```
1000 REM *** THREE-PART SONG ***
1010 :
1020 :
1030 REM ** SET UP SCREEN & VARIABLES
1040 :
1050 PRINT "{CLR}";          :REM CLEAR SCREEN
1060 PRINT "{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}{DOWN}READING";
1070 :
1080 SID = 54272             :REM SOUND CHIP
1090 WV = 16   :REM ALL 3 SAME WAVEFORM
1100 :
1110 :
1120 REM ** SET UP REFERENCE ARRAYS
1130 :
1140 DIM SBN(11), NM$(11)  :REM BASED ON
1150 FOR N = 0 TO 11       :REM NOTES IN
1160 :    READ SBN(N)       :REM HIGHEST
1170 :    READ NM$(N)       :REM OCTAVE
1180 NEXT N
1190 :
1200 DATA 34334, C,  36377, C#
1210 DATA 38539, D,  40831, D#
1220 DATA 43258, E,  45831, F
1230 DATA 48557, F#, 51444, G
1240 DATA 54502, G#, 57743, A
1250 DATA 61177, A#, 64815, B
1260 :
1270 :
1280 REM ** READ IN THE MUSIC AND
            STORE IT IN ARRAYS
1290 :
```

```
1300 DIM LFP(3,200), HFP(3,200),
          WVC(3,200)
1310 :
1320 VOICE = VOICE + 1
1330 IF VOICE = 4 THEN 1890
1340 EVENT = 1
1350 :
1360 READ NC$
1370 PRINT ".";
1380 IF NC$ = "XXX" THEN 1320
1390 :
1400 GOSUB 2440 :REM CONVERT TO POKE #S
1410 :
1420 READ DUR
1430 FOR N = 1 TO DUR-1
1440 :    LFP(VOICE,EVENT) = LFP
1450 :    HFP(VOICE,EVENT) = HFP
1460 :    WVC(VOICE,EVENT) = WV + 1
1470 :    EVENT = EVENT + 1
1480 NEXT N
1490 IF DUR = 1 THEN 1360
1500 :
1510 LFP(VOICE,EVENT) = LFP
1520 HFP(VOICE,EVENT) = HFP
1530 WVC(VOICE,EVENT) = WV
1540 EVENT = EVENT + 1
1550 :
1560 GOTO 1360
1570 :
1580 :
1590 REM ** THE MUSIC : NOTE-OCT, DUR
1600 :
1610 DATA RES,    4, A-5,    4, B-5,    4
1620 DATA A-5,    4, RES,    4, A-5,    4
1630 DATA B-5,    4, A-5,    4, RES,    4
1640 DATA C-6,    4, E-6,    4, C-6,    4
1650 DATA A-5,    4, A-5,    4, B-5,    4
1660 DATA A-5,    4, XXX
1670 :
1680 DATA RES,    4, E-5,    4, E-5,    4
1690 DATA E-5,    4, RES,    4, E-5,    4
1700 DATA E-5,    4, E-5,    4, RES,    4
1710 DATA G-5,    4, G-5,    4, G-5,    4
1720 DATA E-5,    4, E-5,    4, E-5,    4
1730 DATA E-5,    4, XXX
1740 :
1750 DATA C-5,    2, D-5,    2, C-5,    2
1760 DATA A-4,    2, A-4,    2, G-4,    2
1770 DATA A-4,    4, C-5,    2, D-5,    2
1780 DATA C-5,    2, A-4,    2, A-4,    2
1790 DATA G-4,    2, A-4,    4, C-5,    2
```

```
1800 DATA D-5,    2, E-5,    2, E-5,    2
1810 DATA E-5,    2, G-5,    2, E-5,    2
1820 DATA D-5,    2, C-5,    2, A-4,    2
1830 DATA C-5,    2, A-4,    2, A-4,    2
1840 DATA G-4,    2, A-4,    4, XXX
1850 :
1860 :
1870 REM ** SET ADSR'S FOR THE 3 VOICES
1880 :
1890 ATK =  2 : DKY = 3 :REM SETTINGS
1900 AD = ATK*16 + DKY  :REM POKE VALUE
1910 POKE SID+5, AD     :REM VOICE 1 A-D
1920 :
1930 ATK =  2 : DKY = 3 :REM SETTINGS
1940 AD = ATK*16 + DKY  :REM POKE VALUE
1950 POKE SID+12, AD    :REM VOICE 2 A-D
1960 :
1970 ATK =  2: DKY = 0  :REM SETTINGS
1980 AD = ATK*16 + DKY  :REM POKE VALUE
1990 POKE SID+19, AD    :REM VOICE 3 A-D
2000 :
2010 SST = 6  : RLS = 6 :REM SETTINGS
2020 SR = SST*16 + RLS  :REM POKE VALUE
2030 POKE SID+6, SR     :REM VOICE 1 S-R
2040 :
2050 SST = 12 : RLS = 6 :REM SETTINGS
2060 SR = SST*16 + RLS  :REM POKE VALUE
2070 POKE SID+13, SR    :REM VOICE 2 S-R
2080 :
2090 SST = 15 : RLS = 7 :REM SETTINGS
2100 SR = SST*16 + RLS  :REM POKE VALUE
2110 POKE SID+20, SR    :REM VOICE 3 S-R
2120 :
2130 :
2140 REM ** PLAY THE MUSIC, THEN END IT
2150 :
2160 PRINT "♥";
2170 BEATLNGTH = 10
2180 VLM = 15           :REM MAX VOLUME
2190 POKE SID+24, VLM
2200 :
2210 FOR N = 1 TO (EVENT - 1)
2220 :   POKE SID+1, HFP(1,N)
2230 :   POKE SID, LFP(1,N)
2240 :   POKE SID+8, HFP(2,N)
2250 :   POKE SID+7, LFP(2,N)
2260 :   POKE SID+15, HFP(3,N)
2270 :   POKE SID+14, LFP(3,N)
2280 :
2290 :   POKE SID+4,  WVC(1,N):REM V-1
2300 :   POKE SID+11, WVC(2,N):REM V-2
```

```
2310 :     POKE SID+18, WVC(3,N):REM V-3
2320 :
2330 :     FOR TM = 1 TO BEATLNGTH
2340 :     NEXT TM
2350 NEXT N
2360 :
2370 POKE SID+24,0    :REM VOLUME OFF
2380 END
2390 :
2400 :
2410 :
2420 REM ** CONVERT NOTE-OCTAVE STRING
               TO LO AND HI POKE CODES
2430 :
2440 IF NC$ = "RES" THEN HFP = 0 :
                         LFP = 0 : RETURN
2450 :
2460 NT$ = LEFT$(NC$, LEN(NC$) - 2)
2470 FOR REF = 0 TO 11
2480 :     IF NT$ = NM$(REF) THEN
                  NT = REF : REF = 11
2490 NEXT REF
2500 :
2510 OCT = VAL(RIGHT$(NC$,1))
2520 :
2530 FST = 2 ↑ (7 - OCT)
2540 FST = SBN(NT) / FST
2550 HFP = INT (FST/256)
2560 LFP = INT (FST - 256*HFP)
2570 :
2580 RETURN
```

Fig. 8-4. Listing of the program Three-Part Song.

Song. Type it in; save it; then run it. Take some time to compare this program with Read Music, listed in Fig. 8-2. They're very similar. In our discussion, I'll focus in on the differences.

### 8.5.1 Filling Up the Performance Arrays

The first change shows up in line 1090. The program sets up a waveform variable right away; it will be used to fill the waveform control performance array. Other than that, the first two modules are the same: clear the screen, set up for feedback, and fill the reference arrays.

Now it's time to read notes and pack arrays. Lines 1300-1560 do the job. First, line 1300 dimensions three performance arrays. Two will hold frequency values, and the third will hold waveform control values.

This program segment reads notes and packs arrays a voice at a time. The pseudo-note XXX, signals the end of one voice's notes.

The voice number then goes up by one. When it hits 4, all three voice's have been taken care of, and the program moves on to set up the ADSR values.

When line 1360 reads a valid note, the program jumps to the same frequency-figuring subroutine used in the Read Music program. This subroutine sends back values for the high and low bytes of the frequency setting. Then it's time to pack arrays.

If a note has a duration of just one beat, it'll go through the packing loop in lines 1430-1480 just once. Lines 1440-1450 set the low and high frequency bytes. Then line 1460 sets the waveform control array with a value that'll trigger the ADSR envelope. Line 1490 sends the program back to read another note.

A note that lasts longer than one beat gets treated differently. It will go through the loop in lines 1430-1480 one less time than its duration in beats. Thus, on all beats up to the last one, the waveform control array will receive a value that triggers the start of an ADSR envelope. Lines 1510-1540 handle the arrays for the final beat. There's no change in how frequency is handled. However, the waveform control array now gets a value that will trigger the release stage of the ADSR envelope.

### 8.5.2 Setting the ADSR Envelopes

After the notes are read in and the performance arrays filled, it's time to set ADSR envelopes for each voice. The routines used in lines 1890-2110 use the same technique shown in the program Read Music. Here's one hint: low notes need higher sustain levels to be heard as easily as high notes. That's because of the way our ears are built. In this program, voice #1 plays the highest notes, voice #3 the lowest, with voice #2 in between. Therefore I gave voice #3 the highest sustain level, voice #1 the lowest, with voice #2 in between.

### 8.5.3 Playing It

After a few final preparations, the program can play the music. Lines 2160-2190 clear the screen, set the length of a beat, and adjust the overall volume. Then comes the performance loop. It will repeat as many times as there are beats. Lines 2220-2270 set the frequency registers for all three voices. Then lines 2290-2310 pick off values from the new waveform control array and poke them into each voice's waveform control register. The voices operate independently; on any given beat, two voices might trigger the start of an ADSR envelope, and the other one might trigger the release stage.

The technique of releasing a voice on its last beat works well if there's a fairly long release period. Change the release settings in lines 2010, 2050, and 2090 to lower values and then run the program. Do you notice the choppiness?

### 8.5.4 Variations

The data in Three-Part Song is based on the English folk melody "Are You Going To The Fair". Figure 8-5 rounds out our salute to pre-Beatles English music. Load in Three-Part Song and then type the lines from Fig. 8-5. Now your Commodore 64 will play the song "Coventry Carol."

Three-Part Song has a lot of room for experimentation. See if you can get the three voices to sound like completely different in-

```
1000 REM *** COVENTRY CAROL ***
1610 DATA A-4,    4, A-4,    4, G#-4,  4
1620 DATA A-4,    8, C-5,    4, B-4,   8
1630 DATA A-4,    4, G#-4,  12, RES,   1
1640 DATA A-4,    4, B-4,    4, C-5,   4
1650 DATA D-5,    8, B-4,    4, A-4,  20
1660 DATA RES,    1, E-5,    4, D-5,   8
1670 DATA C-5,    4, B-4,    8, C-5,   4
1680 DATA B-4,    8, A-4,    4, G#-4, 12
1690 DATA RES,    1, A-4,    4, G#-4,  4
1700 DATA A-4,    4, D-5,    4, B-4,   8
1710 DATA C#-5,  12, XXX
1715 :
1720 DATA C-4,    8, D-4,    4, E-4,   8
1725 DATA E-4,    4, F-4,    8, F-4,   4
1730 DATA E-4,   12, RES,    1, E-4,   8
1735 DATA A-4,    4, A-4,    4, F-4,   4
1740 DATA G-4,    4, E-4,    8, D-4,   4
1745 DATA C-4,    8, RES,    1, G-4,   4
1750 DATA A-4,    8, E-4,    4, G-4,   8
1755 DATA A-4,    4, G-4,    8, E-4,   4
1760 DATA E-4,    8, D-4,    4, RES,   1
1765 DATA E-4,    4, F-4,    4, E-4,   4
1770 DATA F-4,    4, G-4,    8, E-4,  12
1775 DATA XXX
1780 :
1785 DATA A-2,    8, B-2,    4, C-3,   8
1790 DATA A-2,    4, D-3,    8, D-3,   4
1795 DATA E-3,    8, D-3,    4, RES,   1
1800 DATA C-4,    4, B-3,    4, A-3,   4
1805 DATA B-2,    8, E-3,    4, C-3,   8
1810 DATA B-2,    4, A-2,    8, RES,   1
1815 DATA C-3,    4, F-2,    8, A-2,   4
1820 DATA E-3,    8, E-3,    4, E-3,   8
1825 DATA C-3,    4, B-2,   12, RES,   1
1830 DATA C-3,    4, D-3,    4, C-3,   4
1835 DATA B-2,    8, E-3,    4, A-2,  12
1840 DATA XXX
1890 ATK =  2 : DKY = 3 :REM SETTINGS
1930 ATK =  2 : DKY = 3 :REM SETTINGS
1970 ATK =  2 : DKY = 0 :REM SETTINGS
2010 SST =  4 : RLS = 6 :REM SETTINGS
2050 SST =  9 : RLS = 6 :REM SETTINGS
2090 SST = 15 : RLS = 7 :REM SETTINGS
```

Fig. 8-5. Changes to Three-Part Song that teach it to play the song "Coventry Carol."

struments. And remember, although SID can imitate real instruments, it really shines when you come up with sounds never heard from wood or brass or strings.

## 8.6 CHAPTER SUMMARY

You've examined a couple of ways to get interesting music out of your Commodore 64. Here's a summary of what you've covered:

* Setting up a reference octave to help translate note names and octave numbers into SID frequency settings
* Using performance arrays to store SID frequency setup information for each beat of a piece of music
* Using performance arrays to implement three voice music
* Turning voices on and off with a waveform control performance array

In the next chapter, we'll leave harmony behind, and get SID to generate some ear-tickling sound effects.

## 8.7 EXERCISES

### 8.7.1 Self Test

Answers are in Section 8.7.3.

1. (8.1.1) If an A note in the third octave has a frequency of 220 hertz, what's the frequency of a first octave A note?
2. (8.1.1) Using the string notation introduced in Section 8.1.2, B#-6 represents a __________ octave B sharp.
3. (8.2) A performance array can hold SID settings for each __________ of a song.
4. (8.3) The program Read Music stores __________ settings for each beat in the performance arrays LFP(200) and HFP (200).
5. (8.4) One way to handle more than one voice at a time is to use __________-dimensional performance arrays.
6. (8.4) You can avoid slurring and chopping by triggering the __________ stage on the last beat of a note.
7. (8.5) Take a look at the program Three-Part Song. What's the smallest number of beats a note can have and still get its release stage triggered?

### 8.7.2 Programming Exercises

1. Change the program Read Music so it repeats the music if desired. It shouldn't have to set up the performance arrays again.
2. Change the program Three-Part Song so it lets the user adjust the speed (tempo) the music's played at.
3. Change the program Three-Part Song so it lets the user adjust the overall pitch by octaves.

### 8.7.3 Answers to Self Test

As usual, you may come up with better answers.

1. 55 hertz
2. sixth
3. beat
4. frequency
5. two
6. release
7. two

## 8.7.4 Possible Solutions to Programming Exercises

1. Load in the program Read Music. Then type in these lines:

```
1000 REM *** JUKE BOX ***
1830 REM ** PLAY THE MUSIC
1990 :
1991 :
1992 REM ** PLAY IT AGAIN ?
1993 :
1994 PRINT "PRESS ANY ";
1995 PRINT "KEY WITHIN 5 "
1996 PRINT "SECONDS FOR A ";
1997 PRINT "REPLAY"
1998 :
1999 TI$ = "000000"   :REM RESET TIME
2000 :
2001 GET KY$           :REM READ KEYBOARD
2002 IF KY$ <> "" THEN 1770
2003 IF VAL(TI$) < 5 THEN 2001
2004 :
2005 PRINT "";
2006 END
2007 :
```

2. Load in the program Three-Part Song. Then type in these lines:

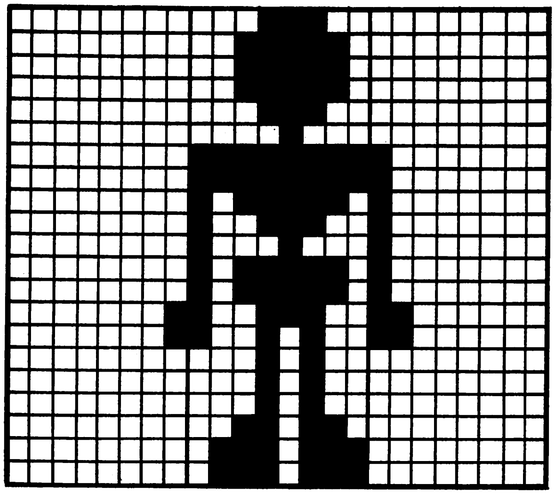
```
1000 REM *** ADJUSTABLE TEMPO ***
1001 :
1002 :
1003 REM ** GET THE TEMPO
1004 :
1005 PRINT "PRESS A KEY ";
1006 PRINT "TO SET THE TEMPO : "
1007 PRINT "(1-SLOWEST     ";
1008 PRINT "9-QUICKEST)";
1009 :
1010 GET KY$
1011 IF KY$ = "" THEN 1010
1012 IF ASC(KY$) < 49  OR  ASC(KY$) > 57
                        THEN 1010
1013 :
1014 PRINT ""; KY$; ""
1015 TEMPO = VAL(KY$)
1016 :
1017 FOR N = 1 TO 500
1018 NEXT N
2170 BEATLNGTH = (10 - TEMPO) ↑ 1.7
```

3. Load in the program Three-Part Song. Then type in these lines:

```
1000 REM *** OCTAVE MOVER ***
1001 :
1002 :
1003 REM ** GET OCTAVE ADJUSTMENT
1004 :
1005 PRINT "HOW MANY ";
1006 PRINT "OCTAVES DO YOU"
1007 PRINT "WANT TO MOVE ";
1008 PRINT "(0 - 3) ? ";
1009 GET ADJ$
1010 IF ADJ$ = "" THEN 1009
1011 :
1012 IF ASC(ADJ$) < 48 OR ASC(ADJ$) > 51
        THEN 1009
1013 :
1014 PRINT ""; ADJ$; "" :REM PRINT IT
1015 ADJ = VAL (ADJ$)
1016 IF ADJ = 0 THEN 1027 :REM NO 2ND ?
1017 :
1018 PRINT "MOVE UP OR ";
1019 PRINT "DOWN (U/D) ? ";
1020 GET UD$
1021 IF UD$ = "" THEN 1020
1022 :
1023 IF UD$ <> "U"  AND  UD$ <> "D"
                    THEN 1020
1024 PRINT ""; UD$; "" :REM PRINT IT
1025 :
1026 IF UD$ = "D" THEN ADJ = -ADJ
1027 FOR N = 1 TO 500 : NEXT N
1028 :
1029 :
2510 OCT = VAL(RIGHT$(NC$,1)) + ADJ
2513 IF OCT < 0 THEN OCT = OCT + 1 :
                    GOTO 2513
2516 IF OCT > 7 THEN OCT = OCT - 1 :
                    GOTO 2516
```

# Chapter 9

# Special Sound Effects



In this chapter you'll get SID to produce some interesting sound effects. You'll listen to clocks, gongs, a SID oscillator, horses, projectiles, and pulsing weirdness. Along the way, you'll think about timing, ADSR envelope design, ring modulation, vibrato, eavesdropping, linkage, rhythm, noise, and variations in volume, frequency, and pulse width.

Keep in mind that the key to sound effects is imaginative variation: changing volume, waveforms, frequencies, timing, rhythms, and so on. Of course, you've got to know what to change and how to do it. Some of this can be learned by playing with SID and programs like those in this chapter. You'll also need to spend time listening to the world around you. Train your ears to be better sound analyzers.

## 9.1 THE CLOCK

Figure 9-1 shows the program Clock. Read it, and then run it. Play around with the numbers. See if you can get a more interesting rhythm out of the ticking clock.

You end up changing a lot of SID's registers when you work with sound effects. This can cause complications if you forget which registers have been set. The programs in this chapter all begin and end by clearing SID's registers.

Let's look at the ADSR envelope this program generates. Attack, decay, and release rates are all set to 0, and the sustain level is 15, the maximum. The sound will quickly rise to peak volume, quickly decay to the same level (huh?), sit there until release is triggered, and then quickly fall to zero. Figure 9-2 shows a picture of the envelope.

Once the envelope and overall volume is set, the program is ready to play a series of ticks and tocks. First, lines 1220-1260 play the

```
1000 REM *** CLOCK ***
1010 :
1020 :
1030 REM ** CLEAR SID & PRINT PROMPT
1040 :
1050 SID = 54272        :REM SOUND CHIP
1060 FOR REG = SID TO SID+24
1070 :    POKE REG, 0
1080 NEXT REG
1090 :
1100 PRINT "♥";
1110 PRINT "PRESS SPACEBAR TO END"
1120 :
1130 :
1140 REM ** INITIALIZE SID REGISTERS
1150 :
1160 POKE SID+6, 240   :REM MAX SUSTAIN
1170 POKE SID+24, 15   :REM MAX VOLUME
1180 :
1190 :
1200 REM ** PLAY IT ; END ON A KEYPRESS
1210 :
1220 POKE SID+1, 80             :REM TICK
1230 POKE SID+4, 17
1240 FOR T = 1 TO 3 : NEXT T
1250 POKE SID+4, 16
1260 FOR T = 1 TO 300 : NEXT T
1270 :
1280 POKE SID+1, 60             :REM TOCK
1290 POKE SID+4, 17
1300 FOR T = 1 TO 3 : NEXT T
1310 POKE SID+4, 16
1320 FOR T = 1 TO 300 : NEXT T
1330 :
1340 GET KP$
1350 IF KP$ = "" THEN 1220
1360 :
1370 :
1380 REM ** CLEAN UP & END
1390 :
1400 FOR REG = SID TO SID+24
1410 :    POKE REG, 0
1420 NEXT REG
1430 PRINT "♥";
1440 :
1450 END
```

Fig. 9-1. Listing of the program Clock.

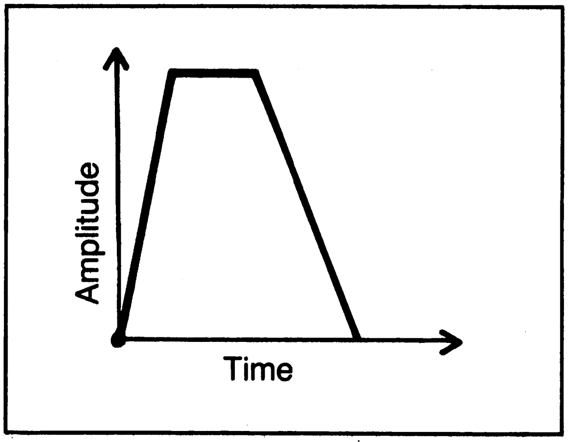



Fig. 9-2. A picture of the ADSR envelope used in Clock.

tick. Line 1220 sets a frequency, and then line 1230 sets the triangle waveform and triggers the sound. There's a short pause, with the tick at peak volume, and then line 1250 releases the sound. Finally, there's a relatively long pause.

Then, it's time for lines 1280-1320 to give you a tock. A new, lower frequency is set. Then the sound is triggered, held a bit, and released. Again, there's a relatively long pause. Line 1340 scans the keyboard; if no key's been pressed, it's back up to line 1220 for another tick.

The top row in Fig. 9-3 shows a few beats' worth of volume information (not to scale) for this program. Notice the regularity of the sketch. The second row shows what would happen if the tick had a longer sustain period and the tock came along sooner. See if you can change Clock so it sounds more like the second row. Drawing these rough pictures gives me a first crack at SID settings and delay loops when I'm planning a new sound.

You need programs that can be easily modified when you're creating sound effects. Put in plenty of delay loops and statements that set the SID registers. It takes a lot of fine tuning to produce the sounds you hear in your imagination.

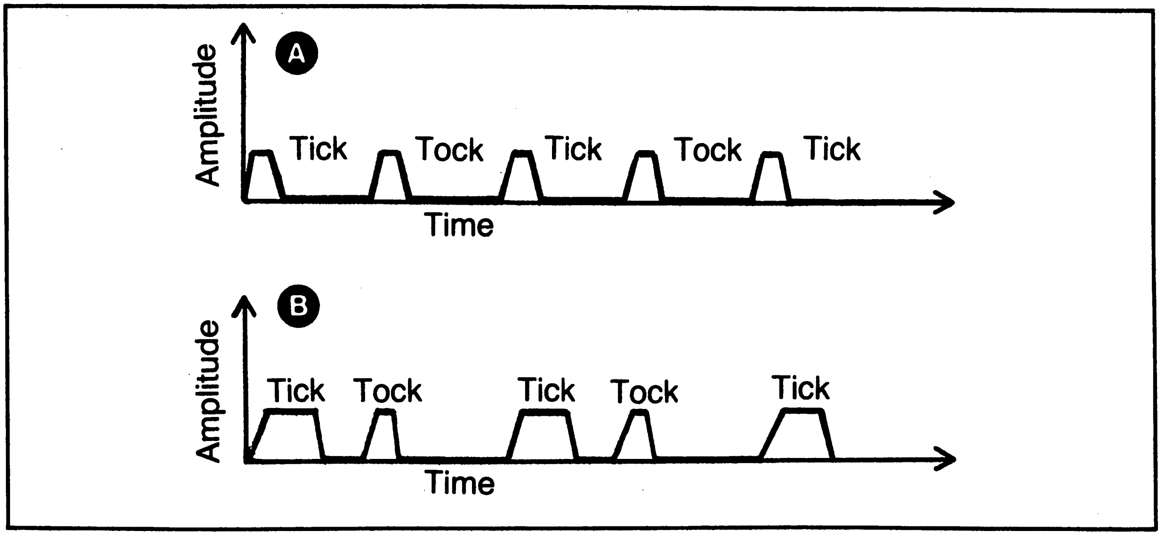



Fig. 9-3. Top: a few beat's worth of volume information for Clock (not to scale). Bottom: A possible variation of Clock with a less uniform beat.

## 9.2 THE GONG MACHINE

You've heard SID produce a clock's ticks. Now let's get some big, reverberating gong noises. You'll start by looking at ring modulation. It's one way to link two voices together.

### 9.2.1 Ring Modulation

There's a fifth SID waveform option I haven't mentioned yet. It's called ring modulation. SID can combine information from two voices to form what's called a ring-modulated output. This ring modulation does a great job on gongs, bells, chimes, and the like.

Here's how you get a voice to produce ring modulated output. First, select the voice's triangle waveform. Next, set its ring modulation control bit, bit 2 of the waveform control register, to 1. Finally, set the voice's partner to a frequency other than 0.

What's a partner? When a voice is set up for ring modulation, it mixes another voice's frequency information with its own. Voice #1 uses voice #3 as a partner, voice #2 uses voice #1, and voice #3 uses voice #2.

Here's an example. Let's set voice #1 up for ring modulation. You need to set the following bits of the wave control register at SID+4: bit 0 to trigger the start of an ADSR envelope, bit 2 to choose ring modulation, and bit 4 to select the triangle waveform. Adding the values of those bits gives you 21, so 21 is the number to put into SID+4. See Fig. 9-4. Then you need to set voice #3 to a nonzero frequency. You can do this by setting the frequency register at SID+15 to a nonzero value, say 19. When it's time to trigger the release stage of the ADSR envelope, you'll just place the value 20 (bit 0 off) into SID+4.

### 9.2.2 The Program

Figure 9-5 lists the program Gong Machine, which uses ring modulation to give you nine different chime sounds. Read it over; then type it in, save it, and run it.

After SID is cleared and the screen's set up, lines 1180-1190 set the ADSR envelope for voice #1. Line 1210 sets the overall volume.

Lines 1260-1310 obtain keypresses. Pressing the spacebar ends the program. Pressing one of the number keys 1-9 will generate a gong sound. Any other keyboard input is ignored.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
| Bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Bit function | Noise | Pulse | Sawtooth | Triangle | — | Ring modulation | Sync | Gate | |
| | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | |
| | | | | 16 | + | 4 | + | 1 | = 21 |

Fig. 9-4. Setting up a voice's fifth register for ring modulation.

```
1000 REM *** GONG MACHINE ***
1010 :
1020 :
1030 REM ** CLEAR SID & PRINT PROMPTS
1040 :
1050 SID = 54272          :REM SOUND CHIP
1060 FOR REG = SID TO SID+24
1070 :    POKE REG, 0
1080 NEXT REG
1090 :
1100 PRINT "♥";
1110 PRINT "PRESS KEYS 1-9 FOR GONGS."
1120 PRINT
1130 PRINT "PRESS SPACEBAR TO END."
1140 :
1150 :
1160 REM ** INITIALIZE SID REGISTERS
1170 :
1180 POKE SID+5,12   :REM ATK=0, DKY=12
1190 POKE SID+6,9    :REM SST=0, RLS=9
1200 :
1210 POKE SID+24,15 :REM MAX VOLUME
1220 :
1230 :
1240 REM ** PLAY IT
1250 :
1260 GET KP$          :REM SCAN KEYBOARD
1270 IF KP$ = "" THEN 1260
1280 IF KP$ = " " THEN 1480 :REM END IT
1290 :
1300 KP = VAL (KP$)   :REM MUST BE 1-9
1310 IF KP<1 OR KP>9 THEN 1260
1320 :
1330 POKE SID+1, KP * 1.5 + KP
1340 POKE SID+15, 19 + KP
1350 POKE SID+4, 21       :REM BONG GONG
1360 FOR T = 1 TO 100
1370 :    QUAVER = T - INT(T/10)*10
1380 :    POKE SID, QUAVER * 20
1390 :    GET KP$
1400 :    IF KP$ <> "" THEN T = 100
1410 NEXT T
1420 POKE SID+4, 20       :REM GONG GONE
1430 GOTO 1270
1440 :
1450 :
1460 REM ** CLEAN UP & END
```

```
1470 :
1480 FOR REG = SID TO SID+24
1490 :    POKE REG, 0
1500 NEXT REG
1510 PRINT "♥";
1520 :
1530 END
```

Fig. 9-5. Listing of the program Gong Machine.

Line 1330 sets the frequency of voice #1 based on the number of the pressed key. Line 1340 does the same for voice #3. Line 1350 then triggers the start of a ring modulated sound.

To add emphasis to the sound, lines 1360-1410 wiggle the frequency of voice #1. This kind of effect is known as vibrato or tremolo. While the program's wiggling, it's also keeping an eye on the keyboard. If a key is pressed, it'll abort the vibrato, release the sound, and pop back up to deal with the keypress. If no key is pressed during the vibrato, the gong calmly fades away, and the program goes back to scan the keyboard.

I spent quite a while trying different formulas in lines 1330 and 1340. The relationship between two voices' frequencies and the resulting ring-modulated sound is complex. You might want to try some formulas of your own.

Another spot worth experimenting with is line 1370, the vibrato formula. You can get all kinds of interesting gong variations by changing this line.

## 9.3 SID LISTENS TO ITSELF

Ring modulation lets one voice affect another. But there's not as much control as you might need in certain situations. It'd be nice if you could eavesdrop on some of SID's output. The registers at SID+27 and SID+28 let you do just that. They give you a more controlled way to link voices together.

### 9.3.1 The Eavesdropping Registers

SID+27 shows the output of voice #3's oscillator. SID+28 shows the output of voice #3's envelope generator. You can read these registers and then use the values to modify other SID settings.

You've got to start up the voice #3 oscillator to get SID+27 to show changing values. This is done by setting a frequency and waveform for voice #3. You won't hear voice #3 as long as you don't trigger the ADSR envelope. So voice #3 can oscillate away, not making a sound, while you read its oscillations from SID+27.

You've got to trigger the voice #3 envelope generator in order to have its values show up at SID+28. This will usually cause voice #3 to put out some sounds. If you don't want to hear voice #3, but still want to monitor its envelope generator, you silence it by setting bit 7 of SID+24 to 1. SID+24 is the same register used to set overall volume. To set bit

7 to 1, just add 128 to your volume setting and poke the new value in.

### 9.3.2 The Mad Computer

Let's look at a program that uses these new eavesdropping capabilities. Figure 9-6 lists the program Mad Computer. Read it, type it, save it, and run it. Pressing any of the number keys 1-9 will change the sound pattern. Pressing any other key ends the programs.

In this program, voice #1 makes sounds whose frequencies are based on the oscillations of voice #3. Line 1300 is the key. It takes a value from SID+27 and plugs it into one of voice #1's frequency registers. After a brief pause, the program looks for a keypress.

What values will be showing up at SID

```
1000 REM *** MAD COMPUTER ***
1010 :
1020 :
1030 REM ** CLEAR SID & PRINT PROMPTS
1040 :
1050 SID = 54272
1060 FOR N = SID TO SID+24
1070 :    POKE N,0
1080 NEXT N
1090 :
1100 PRINT "♥";
1110 PRINT "PRESS KEYS 1-9 TO CHANGE"
1120 PRINT
1130 PRINT "ANY OTHER KEY TO END"
1140 :
1150 :
1160 REM ** INITIALIZE SID REGISTERS
1170 :
1180 POKE SID+6,240   :REM V-1 SST = MAX
1190 :
1200 POKE SID+15,18   :REM SET V-3 FRQ
1210 POKE SID+18,16   :REM SET V-3 WVF
1220 :
1230 POKE SID+24,15   :REM SET VOLUME
1240 :
1250 :
1260 REM ** PLAY IT
1270 :
1280 POKE SID+4,17    :REM TRIG V-1 ATK
1290 :                :REM SET V-1 FREQ
                     BY V-3 OSCILLATIONS
1300 POKE SID+1,
                PEEK(SID+27)
1310 FOR T = 1 TO 5  :REM WAIT A BIT
1320 NEXT T
```

```
1330 :
1340 :
1350 REM ** SCAN KYBD TO PLAY MORE,
              CHANGE SOUND, OR END
1360 :
1370 GET KP$
1380 IF KP$ = "" THEN 1300 :REM MORE
1390 :
1400 IF ASC(KP$) < 49 THEN 1450
1410 IF ASC(KP$) > 57 THEN 1450
1420 :   POKE SID+15, VAL(KP$) * 7
1430 :   GOTO 1300     :REM SOUND CHANGED
1440 :
1450 FOR REG = SID TO SID+24 :REM CLEAN
1460 :    POKE REG, 0          :REM UP
1470 NEXT REG                  :REM & END
1480 PRINT "♥";
1490 :
1500 END
```

Fig. 9-6. Listing of the program Mad Computer.

+27? You have to consider how voice #3 is oscillating. Since the triangle waveform is selected in line 1210, voice #3's output will go from 0 to 255 and back to 0 again, at a rate set by its frequency. The values picked up in line 1300 will depend on this frequency and on how often the sampling takes place.

Now, most of the time voice #1 samples SID+27 at a steady rate, breaking only to decipher an occasional keypress. There will be a certain pattern to the samples it picks up and thus to the sound it makes. Pressing one of the keys 1-9 changes voice #3's frequency. Voice #1, still looking at voice #3's oscillations at a steady rate, will start seeing different patterns of data, and so its sound pattern will change.

There is one last interesting fact about this program: voice #1's volume rises to its peak level and stays there until the program ends. Two settings accomplish this. First, the sustain level is set to a maximum. Second, the

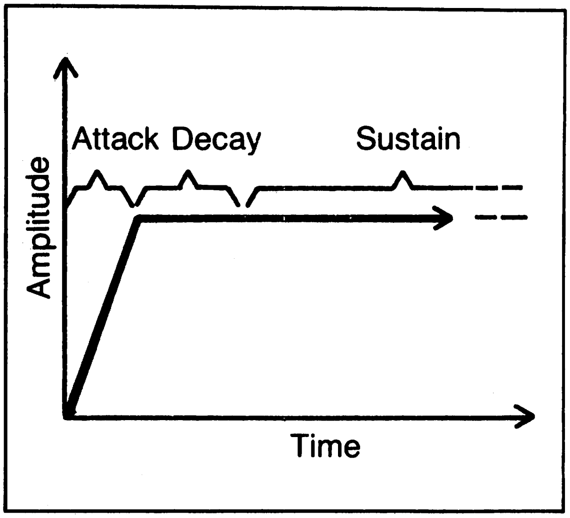



Fig. 9-7. A picture of the ADSR envelope used in Mad Computer.

release stage of the ADSR envelope isn't triggered until the program ends. Figure 9-7 shows what this envelope looks like.

## 9.4 DADADUM DADADUM DADADDUM DUM DUM . . .

The next program uses a number of timing loops to simulate the sound of a galloping horse. If you don't understand where this section's title comes from, just ask someone who grew up listening to tales of the masked man with the silver bullets.

Figure 9-8 lists the program Horse. After you've run it, change the rhythms by fooling with the timing formulas. Can you get the horse to canter? Prance? Race pell-mell down

```
1000 REM *** HORSE ***
1010 :
1020 :
1030 REM ** CLEAR SID & PRINT PROMPTS
1040 :
1050 SID = 54272          :REM SOUND CHIP
1060 FOR REG = SID TO SID+24
1070 :   POKE REG, 0
1080 NEXT REG
1090 :
1100 PRINT "♥";
1110 PRINT "PRESS SPACEBAR TO STOP"
1120 :
1130 :
1140 REM ** INITIALIZE SID REGISTERS
1150 :
1160 POKE SID+5, 4    :REM ATK=0, DKY=4
1170 POKE SID+6, 164 :REM SST=10, RLS=4
1180 :
1190 :
1200 REM ** SET VOLUME, FREQUENCY,
                TIMING
1210 :
1220 VC = 1          :REM VOLUME CHANGE
1230 VLM = 12        :REM STARTING VOLUME
1240 :
1250 VLM = VLM + VC   :REM UPDATE VOLUME
1260 IF VLM = 15 OR VLM = 12
                        THEN VC = -VC
1270 POKE SID+24, VLM
1280 :
1290 FRQ = 35 - VLM   :REM FRQ/VLM LINK
1300 DLY = 17         :REM TIMING FACTOR
1310 :
1320 :
1330 REM ** PLAY THE FOUR HOOVES
```

```
1340 :
1350 POKE SID+1, FRQ + 2     :REM HOOF 1
1360 POKE SID+4,129
1370 FOR T = 1 TO DLY*1.1  :  NEXT T
1380 POKE SID+4, 128
1390 FOR T = 1 TO DLY * 3  :  NEXT T
1400 :
1410 POKE SID+1, FRQ         :REM HOOF 2
1420 POKE SID+4,129
1430 FOR T = 1 TO DLY  :  NEXT T
1440 POKE SID+4, 128
1450 FOR T = 1 TO  DLY * 1.1  :  NEXT T
1460 :
1470 POKE SID+1, FRQ - 2     :REM HOOF 3
1480 POKE SID+4,129
1490 FOR T = 1 TO DLY * 1.2:  NEXT T
1500 POKE SID+4, 128
1510 FOR T = 1 TO DLY * 1.4 :NEXT T
1520 :
1530 POKE SID+1, FRQ         :REM HOOF 4
1540 POKE SID+4,129
1550 FOR T = 1 TO DLY * .8:  NEXT T
1560 POKE SID+4, 128
1570 FOR T = 1 TO DLY * 5.5  :  NEXT T
1580 :
1590 :
1600 REM ** QUIT IF KEY PRESSED
1610 :
1620 GET KP$
1630 IF KP$ = "" THEN 1250
1640 :
1650 FOR REG = SID TO SID+24
1660 :    POKE REG, 0
1670 NEXT REG
1680 PRINT "♥";
1690 :
1700 END
```

Fig. 9-8. Listing of the program Horse.

the stretch? It's all in the timing.

Let's examine the program. The first segment performs the usual SID clearing and prompt printing. The next segment sets up the ADSR envelope for the hoofbeats. This sound will take on a pretty classic envelope. It climbs quickly to peak volume, decays at a moderate rate, holds at about two-thirds of peak volume, and then fades to zero volume at a moderate rate. You can suggest different types of horses, shoes, and surfaces by changing the envelope and waveform.

Lines 1220-1290 form an interesting segment. Each time through, the program will

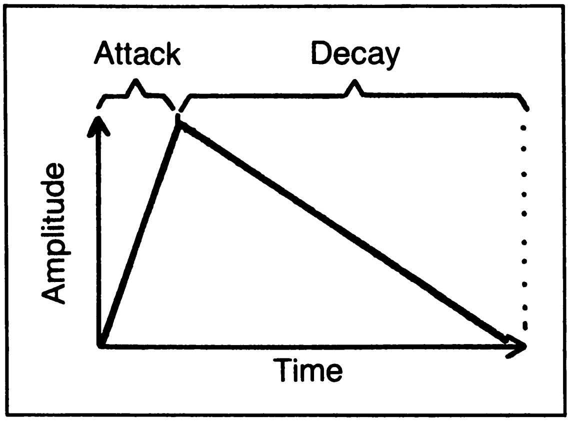



Fig. 9-9. A picture of the ADSR envelope you'll try to set up to simulate a gunshot.

make slight changes to the volume and frequency settings. This variety makes the hoofbeats sound a little more natural. Line 1300 sets a basic timing variable; all the other timing will be based on the value of DLY. You might try inserting a formula that varies DLY's value every now and then.

Lines 1350-1570 play the hooves, one at a time. For each hoof, voice #1 gets gated; there's a short delay; the voice is released; then there's a longer delay. The various delays vary from hoof to hoof; just like snowflakes, no two feet are exactly alike.

See if you can make it seem as if the horse is slowly approaching the listener, passing by, and then moving away. Here are three helpful hints:

As sounds approach, they get louder and the frequency goes up.

As sounds move away, they get softer and the frequency goes down.

A little exaggeration never hurts a sound effect.

## 9.5 BANG BANG

Before the days of electronic noise making, a favorite pastime was playing with rolls of caps. These were long rolls of paper with little explosive bumps every quarter inch or so. They were meant for cap guns, but the guns misfired a lot. Besides, the real fun lay in getting a bunch of 'em to go off at a time. So we usually just laid a roll on the pavement and clobbered it with a good-sized rock. We loved the noise. The smell wasn't bad, either.

We'll leave it to the psychologists to figure out why people enjoy explosive sounds. In the meantime, you can use SID to make some blasts.

### 9.5.1 Thinking About the Sounds

Let's think about simulating the sound of a gun. You've really got two sounds to deal with. First, there's a cracking explosion, as gunpowder ignites and launches a bullet. Then there's the sound of the bullet zipping through the air.

White noise comes in very handy for explosions. Remember, setting bit 7 of a voice's waveform register selects white noise. You'll start each gunshot with a burst of white noise. Also, explosions start out loudly and then fade away. So you'll have to try to set up an ADSR envelope that looks like the one shown in Fig. 9-9.

Now, for the whistling of the bullet as it goes through the air. It takes a moment after the explosion for the bullet to pick up enough speed to be heard. As it accelerates towards a listener, its sound rises in pitch and volume. As it passes and moves on away from a listener, the sound drops in pitch and volume. You'll need an ADSR envelope that gives a discernible rise and fall in volume. Then you'll need to set up some frequency setting loops that go along with the volume changes.

### 9.5.2 Making the Sounds

Figure 9-10 lists the program Bam-P'Twang, which makes shooting noises. Run it. How does it sound? You may want to add an echo with a third voice, or adjust the timing, or change the frequencies. As usual, experimentation will teach you a lot.

Lines 1180-1220 set two ADSR envelopes. Voice #1 will handle the explosion, and voice #2, the flight. Voice #1, with an attack rate of 0, will hit peak volume in 2 thousandths of a second, and then start decaying at a much

```
1000 REM *** BAM-P'TWANG ***
1010 :
1020 :
1030 REM ** CLEAR SID & PRINT PROMPTS
1040 :
1050 SID = 54272        :REM SOUND CHIP
1060 FOR REG = SID TO SID+24
1070 :    POKE REG, 0
1080 NEXT REG
1090 :
1100 PRINT "♥";
1110 PRINT "PRESS SPACEBAR FOR SOUND."
1120 PRINT
1130 PRINT "PRESS RETURN KEY TO END."
1140 :
1150 :
1160 REM ** INITIALIZE SID REGISTERS
1170 :
1180 POKE SID+5,10      :REM V-1 ATK/DKY
1190 POKE SID+1, 10     :REM V-1 FREQ
1200 :
1210 POKE SID+12,89     :REM V-2 ATK/DKY
1220 POKE SID+13,10     :REM V-2 SST/RLS
1230 :
1240 POKE SID+24, 15    :REM MAX VOLUME
1250 :
1260 :
1270 REM ** SCAN KEYBOARD
              FOR SHOT OR END
1280 :
1290 GET KP$
1300 IF KP$ = "" THEN 1290
1310 IF KP$ = CHR$(13) THEN 1550
1320 :
1330 :
```

```
1340 REM ** PLAY IT : VOICE 1 EXPLOSION,
                      VOICE 2 FLIGHT
1350 :
1360 POKE SID+4,128    :REM RELEASE V-1
1370 POKE SID+4,129    :REM START V-1
1380 REM  FOR T = 1 TO 20: NEXT T
1390 POKE SID+11, 16   :REM RELEASE V-2
1400 POKE SID+11, 17   :REM START V-2
1410 :
1420 FOR FRQ = 10 TO 80 STEP 3
1430 :    POKE SID+8, FRQ
1440 NEXT FRQ
1450 FOR FRQ = 77 TO 5  STEP -3
1460 :    POKE SID+8, FRQ
1470 :    FOR T = 1 TO 4 : NEXT T
1480 NEXT FRQ
1490 :
1500 GOTO 1290
1510 :
1520 :
1530 REM ** CLEAN UP & END
1540 :
1550 FOR REG = SID TO SID+24
1560 :    POKE REG, 0
1570 NEXT REG
1580 PRINT "♥";
1590 :
1600 END
```

Fig. 9-10. Listing of the program Bam-P'Twang.

slower 1.5 second rate. Voice #2 has an attack rate of 5. It will take 55 thousandths of a second to reach peak volume, and then decay at a rate close to voice #1s. Run the program with some different values defining the ADSR envelopes. You can simulate different types of guns and bullets.

Next, the program waits for a keypress in lines 1290-1300. Pressing the return key will end the program. Anything else shoots a bullet. Lines 1360-1400 do the shooting.

First comes voice #1, with the explosion. Notice how the previous explosion doesn't get completely released until the last possible moment. There's a brief pause in line 1380 so the bullet can pick up a little speed. Then voice #2 chimes in with the whistling flight.

Lines 1420-1480 then take voice #2's frequency on a roller coaster ride. Unlike Gong Machine, this program doesn't scan the keyboard while it's playing with frequencies. That means you don't have rapid-fire capabilities. Try changing this limitation.

White noise also comes in handy for simulations of waves, wind, slamming doors, and similar phenomena. It's particularly interesting to combine it with more musical waveforms, as Bam-P' Twang does.

## 9.6 NOW ENTERING THE PULSER ZONE

The final sound effect combines pulse waveforms of varying width with smooth volume changes. This creates an eerie noise that would be perfect for disintegration rays or background music in the Twilight Zone.

Figure 9-11 lists the program, Pulser Zone. As usual, read it, type it in, save it, run it, and then make your own modifications. Come on back to the book when you're ready for a little explanation.

Lines 1160-1190 set the frequency, ADSR envelope, and volume. As in the program Mad Computer, volume quickly rises to a

```
1000 REM *** PULSER ZONE ***
1010 :
1020 :
1030 REM ** CLEAR SID & PRINT PROMPT
1040 :
1050 SID = 54272           :REM SOUND CHIP
1060 FOR REG = SID TO SID+24
1070 :   POKE REG, 0
1080 NEXT REG
1090 :
1100 PRINT "♥";
1110 PRINT "PRESS SPACEBAR TO END"
1120 :
1130 :
1140 REM ** INITIALIZE SID REGISTERS
1150 :
1160 POKE SID+1, 20     :REM V-1 FREQ
1170 POKE SID+6, 240    :REM V-1 SST/RLS
1180 :
1190 POKE SID+24, 15    :REM MAX VOLUME
1200 :
1210 :
1220 REM ** PLAY IT
1230 :
1240 POKE SID+4, 65  :REM V-1 PULSE ON
1250 :
1260 VLM = 6 : A = -3
1270 IF VLM = 15 OR VLM = 6 THEN A = -A
1280 VLM = VLM + A
1290 POKE SID+24, VLM   :REM ADJUST VOLM
1300 :
1310 FOR N = 8 TO 15    :REM PULSE WIDTH
1320 :   POKE SID+3, N :REM GROWING
1330 NEXT N
1340 :
```

```
1350 FOR N = 14 TO 9 STEP -1  :REM PULSE
1360 :    POKE SID+3, N        :REM WIDTH
1370 NEXT N                    :REM SHRNK
1380 :
1390 :
1400 REM ** SCAN KEYBOARD
1410 :
1420 GET KP$
1430 IF KP$ = "" THEN 1270   :REM NO KEY
1440 :
1450 :
1460 REM ** CLEAN UP & END
1470 :
1480 FOR REG = SID TO SID+24
1490 :    POKE REG, 0
1500 NEXT REG
1510 PRINT "♥";
1520 :
1530 END
```

Fig. 9-11. Listing of the program Pulser Zone.

peak and then stays there until the program ends.

Line 1240 selects the pulse waveform for voice #1 and triggers the ADSR envelope. Line 1260 gives initial values for volume and volume change variables.

Line 1270 is the top of the main program loop. Overall volume will move between settings of 6 and 15. Line 1270 switches the direction of the changes in volume when those limits are reached. Line 1280 changes the volume by adding in the volume change. Then line 1290 pokes in the new value.

Lines 1310-1330 move the pulse width setting from 8 to 15, one step at a time. This corresponds to pulse widths of 50% to 94%. Look back at Section 7.7 if you forget how pulse widths are set.

Lines 1350-1370 then move the pulse width setting back down, one step at a time. Then, lines 1420-1430 do a quick keyboard scan. If a key's been pressed, the program ends. If not, it's back up to line 1270 for a new volume setting and another sweep through the pulse width loops.

Some changes and additions you might make to Pulser Zone include frequency variations, ring modulation, echo effects, a second voice with pulse widths changing in opposite patterns, and a different ADSR envelope. As usual, imaginative experiments will teach you a lot.

## 9.7 CHAPTER SUMMARY

You've played with six different sound effects programs in this chapter. Here are some highlights of what was covered:

* Using short bursts of triangle waveforms to simulate a ticking clock
* Using ring modulation and frequency

changes to simulate gongs
* Using information from voice #3's oscillator to modulate another voice's frequency, helping to simulate an insane computer
* Using a variety of timing loops to simulate the rhythmic sounds of a galloping horse
* Mixing a noise waveform with a triangle waveform to simulate a gunshot
* Varying pulse width and volume to create an eerie, horror movie sound

The last three chapters have given you a glimpse of SID's sound'making capabilities. In Chapter 10, you'll bring SID and VIC together in programs that combine sound and graphics.

## 9.8 EXERCISES

### 9.8.1 Self Test

Answers are in Section 9.8.3.

1. (9.1) You could slow down the ticking in Clock by using ________ numbers in the delay loops of lines 1260 and 1320.
2. (9.2) ________ voices are used to produce ring modulation.
3. (9.3) The registers at SID+27 and SID+28 let you eavesdrop on the activities of ________.
4. (9.4) In the program Horse, slight variations in volume and frequency are used to make the sound more ________.
5. (9.5) The program Bam-P'Twang uses the ________ waveform to simulate exploding gunpowder.
6. (9.6) The loops in lines 1310-1370 of Pulser Zone are used to change voice #1's ________ ________.

### 9.8.2 Programming Exercises

1. Change the program Clock so it uses all three voices, thereby creating a richer sound.
2. Change the program Bam-P'Twang so the explosive sound comes after the bullet flies through the air.
3. Change the program Pulser Zone so that voice #1's frequency changes along with its pulse width.

### 9.8.3 Answers to Self Test

1. larger
2. two
3. voice #3
4. natural
5. noise or white noise
6. pulse width

### 9.8.4 Possible Solutions to Programming Exercises

1. Load in the program Clock. Then type in these lines:

```
1000 REM *** RICH CLOCK ***
1163 POKE SID+13, 120 :REM V-2 SST/REL
1166 POKE SID+20, 180 :REM V-3 SST/REL
1223 POKE SID+8, 20
1226 POKE SID+15, 40
1233 POKE SID+11, 17
1236 POKE SID+18, 17
1256 POKE SID+18, 16
1253 POKE SID+11, 16
```

```
1283 POKE SID+8, 15
1286 POKE SID+15, 30
1293 POKE SID+11, 17
1296 POKE SID+18, 17
1313 POKE SID+11, 16
1316 POKE SID+18, 16
```

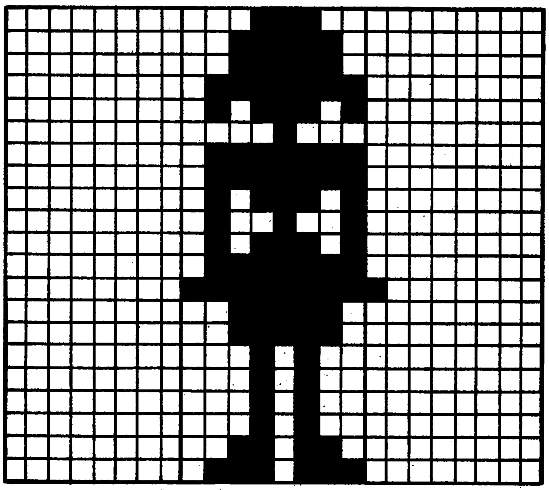
2. Load in the program Bam-P'Twang. Then type these lines:

```
1000 REM *** P'TWANG-BAM ***
1360 :
1370 :
1380 :
1492 POKE SID+4,128    :REM RELEASE V-1
1494 POKE SID+4,129    :REM START V-1
1496 REM  FOR T = 1 TO 20: NEXT T
1498 :
```

3. Load in the program Pulser Zone. Then type in these lines:

```
1000 REM *** SON OF PULSER ***
1160 :
1325 :    POKE SID+1, 2 * N :REM V-1 FRQ
1365 :    POKE SID+1, 2 * N :REM V-1 FRQ
```

# Chapter 10



# Sounds + Graphics = Magic

In the first six chapters, you discovered some of the Commodore 64's graphics abilities. In the last three chapters, you learned how to get it to make sounds. Now it's time to bring graphics and sound together. I'll show you three programs that do this. Along the way, I'll discuss some of the design techniques that I've found helpful with this kind of programming.

## 10.1 SYNERGY

Synergy is a word that comes from biology. It describes situations where two or more things get together and create effects beyond what each component can do alone. Another way to think of it is that the whole becomes greater than the sum of the parts.

Putting pictures and sounds together in a clever way can create some wondrous effects. Imagine the Star Wars movies without their excellent sound tracks. Or playing a silent version of Donkey Kong.

Good sound effects help paint pictures in your mind. Good pictures help suggest certain sounds. If the two elements are carefully brought together, they synergize to create a new level of illusion.

Careful programmers spend a lot of time fine tuning sound and graphics effects. This can be frustrating if you're working with a sloppily designed program. On the other hand, fine tuning a well-designed program can actually be a lot of fun. What makes a program well-designed? One of the most important factors is modularity.

## 10.2 MODULAR THINKING

The easiest job for beginning programmers is learning the rules of a computer langauge and the features of a particular computer. The tough part is learning how to put a large program together.

Good programmers start by thinking. They take a complex problem and start breaking it up into simpler pieces, or modules. Then they break any complex modules down into even simpler pieces. This continues until they've got a set of simple modules that cover every detail of the original problem. Then they start translating their plan into specific computer instructions.

This approach is known as top-down structured programming. It can be used with any computer language on any computer. To most beginners, it seems a waste of time. They want to sit down and start writing code. It usually takes a few experiences wrestling with a badly structured program to see the light.

How do you learn to program this way? Start by reading books and magazines, talking to other programmers, examining all sorts of programs, learning more than one computer language, and trying to pay attention to your mistakes. Keep your mind open, alert, and calm—and write lots of programs.

## 10.3 OF BLIPS AND BEEPS (A HISTORICAL SALUTE)

About ten years ago, the first popular home video game appeared: Pong. Players got to bounce a blip of light around a TV screen. When the blip hit a wall or a simulated ping pong paddle, there was a little beep. This chapter's first program salutes the humble world of blips and beeps.

Figure 10-1 lists the program Bouncer. Type it in, save it, and then run it.

```
1000 REM *** BOUNCER ***
1010 :
1020 :
1030 REM ** DRAW THE BOX & PRINT PROMPT
1040 :
1050 BX$(1) = "┌──────────────────────┐"
1060 BX$(2) = "│                      │"
1070 BX$(3) = "└──────────────────────┘"
1080 :
1090 PRINT "{CLR}{DOWN}{DOWN}"      :REM CLEAR & DOWN
1100 PRINT SPC(10); BX$(1)   :REM TOP
1110 FOR N = 1 TO 3          :REM SIDES
1120 :    PRINT SPC(10); BX$(2)
1130 NEXT N
1140 PRINT SPC(10); BX$(3)   :REM BOTTOM
1150 :
1160 PRINT SPC(10); "{RVS}PRESS ANY ";
1170 PRINT "KEY TO STOP"
1180 :
1190 :
1200 REM ** SET UP SPRITE DATA
```

```
1210 :
1220 FOR N = 12288 TO 12350 :REM MOSTLY
1230 :   POKE N, 0          :REM BLANK
1240 NEXT N
1250 :
1260 FOR N = 12288 TO 12300 STEP 3
1270 : READ SPDTA
1280 : POKE N, SPDTA    :REM BALL SHAPE
1290 NEXT N
1300 :
1310 DATA 60, 126, 255, 126, 60
1320 :
1330 :
1340 REM ** SET UP VIC REGISTERS
1350 :
1360 VIC = 53248       :REM GRAPHICS CHIP
1370 POKE 2040, 192   :REM POINT TO DATA
1380 POKE VIC+39, 7   :REM #0 IS YELLOW
1390 POKE VIC+21, 1   :REM TURN ON #0
1400 :
1410 :
1420 REM ** SET UP THE SOUNDS
1430 :
1440 SID = 54272       :REM SOUND CHIP
1450 POKE SID+5, 24   :REM ATK=1, DKY=8
1460 POKE SID+24, 15  :REM MAX VOLUME
1470 :
1480 :
1490 REM ** INITIALIZE BALL POSITION
            AND MOVES
1500 :
1510 HP = 180 : VP = 89   :REM POSITIONS
1520 HM = 4   : VM = 2.5  :REM MOVES
1530 :
1540 :
1550 REM ** MOVE THE BALL
1560 :
1570 HP = HP + HM     :REM NEW HORZ. POS
1580 VP = VP + VM     :REM NEW VERT. POS
1590 POKE VIC, HP     :REM SET NEW
1600 POKE VIC+1, VP   :REM POSITIONS
1610 :
1620 :
1630 REM ** CHECK FOR A KEYPRESS
1640 :
1650 GET KP$
1660 IF KP$ <> "" THEN 1950 :REM END IT
1670 :
```

```
1680 :
1690 REM ** CHECK FOR A HIT
1700 :
1710 HH = (HP < 111  OR  HP > 249)
1720 VH = (VP < 80   OR  VP > 182)
1730 :
1740 IF (NOT HH) AND (NOT VH) THEN 1570
1750 :
1760 :
1770 REM ** DEAL WITH A HIT
1780 :
1790 IF HH THEN HM = -HM :REM TURN ARND
1800 IF VH THEN VM = -VM :REM TURN ARND
1810 :
1820 POKE SID+4, 16   :REM RELEASE SOUND
1830 POKE SID+1, RND(0)*40 + 10
1840 POKE SID+4, 17   :REM SOUND ATTACKS
1850 :
1860 HUE = (PEEK(VIC+39) AND 15) + 1
1870 IF HUE = 16 THEN HUE = 1
1880 POKE VIC+39,HUE  :REM CHANGE COLOR
1890 :
1900 GOTO 1570        :REM HIT DEALT WITH
1910 :
1920 :
1930 REM ** CLEAN UP AND GO HOME
1940 :
1950 POKE SID+24,0 :REM SOUND OFF
1960 POKE VIC+21,0 :REM SPRITE OFF
1970 PRINT "♥";    :REM CLEAR SCREEN
1980 :
1990 END
```

Fig. 10-1. Listing of the program Bouncer.

In most graphics displays, there are parts of the picture that stay still and parts that move. You can call the parts that stay still static elements and the parts that move dynamic elements.

In Bouncer, the box is the static element, and the moving blip is the dynamic element. The box is drawn with graphics characters, and the blip is a sprite. With the Commodore 64, bit mapping and graphics characters work well for static elements. Graphics characters and sprites work well for dynamic elements.

### 10.3.1 Setting Up the Graphics and Sound

Let's look at Bouncer's modules. Lines 1050-1170 set up the static elements of the screen display. Cursor control characters, strings made up of graphics characters, and the SPC( ) command are all used.

The next two modules set up the sprite.

Lines 1220-1310 load in the data for a very simple sprite, shown in Fig. 10-2. Then lines 1360-1390 set up the necessary VIC registers. Lines 1440-1460 set up the sound chip. The program uses voice #1. Line 1450 sets values for that voice's attack and decay rates. Line 1460 sets an overall SID volume level. Frequency and waveform for voice #1 will be set whenever the blip hits a wall.

### 10.3.2 Getting The Blip Into Motion

The main part of the program forms a large loop. Each time through, the blip moves on the screen. Four variables handle the blip's motion. HP and VP keep track of its vertical and horizontal positions on the screen. HM contains the size and direction of horizontal moves. VM contains the size and direction of vertical moves.

Lines 1510-1520 initialize these four variables. The sprite is put in the middle of the box drawn back in lines 1050-1140, ready to move almost twice as fast horizontally as vertically.

Line 1570 is the top of the main program loop. Lines 1570-1600 figure new horizontal

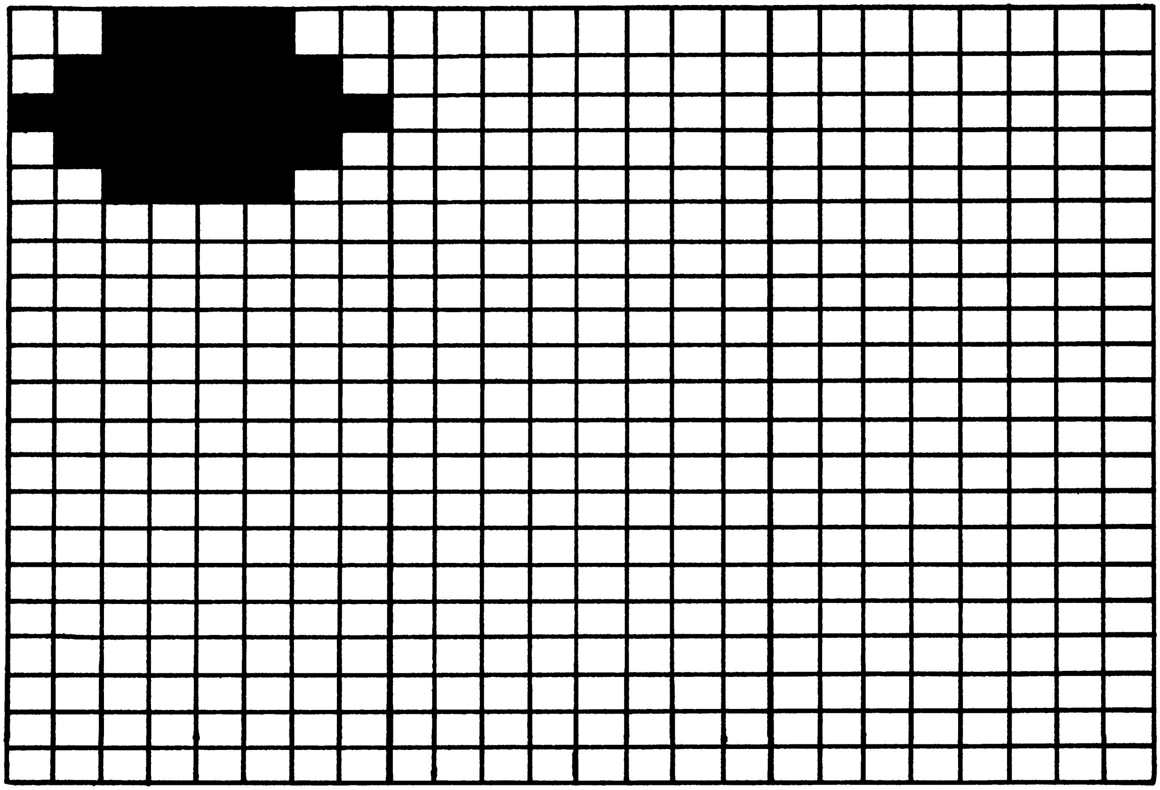


Fig. 10-2. The simple sprite design used in Bouncer.

and vertical positions for the blip and then poke them into sprite #0's position registers.

Next, the program checks for a keypress. Any keypress will cause a jump to the program's closing module.

Lines 1710-1720 use Boolean expressions to see if the blip has hit one of the box's walls. Line 1710 checks for a hit on the side walls, line 1720 for a hit on the top or bottom walls. If no wall has been hit, the program pops on back to the top of the motion loop at line 1570.

### 10.3.3 Dealing With A Hit

The next module, lines 1790-1900, deals with a hit by changing the blip's motion, starting a sound effect, and changing the blip's color.

If the blip has hit a side wall, line 1790 reverses its horizontal motion. If it has hit a top or bottom wall, line 1800 reverses its vertical motion.

Then lines 1820-1840 gives us a sound effect. Line 1820 releases any previous sound. Line 1830 picks a frequency setting at random and then pokes it into the appropriate SID register. Line 1840 then triggers the sound.

Finally, lines 1860-1880 change the blip's color. It will cycle repeatedly through the set of sprite colors, except black. After a hit's been dealt with, the program jumps back to line 1570, which is the top of the motion loop.

### 10.3.4 Cleaning Up

The final module of Bouncer turns off the sound and the sprite and then clears the screen in a straightforward manner. If you wanted to be a bit more thorough, you'd clear all the SID and VIC registers used in the program.

## 10.4 THE PIANORGAN

The next program uses complex character graphics and a speeded-up keyboard scan to create an animated musical instrument. It's listed in Fig. 10-3. Type in Pianorgan; save it; and then run it. When you're playing the instrument, notes will last as long as you hold down a key.

### 10.4.1 Big Strings

This program uses long character strings to quickly draw the singing keys. These strings contain cursor control characters, display option characters, graphics characters, and text characters. Although such strings take time to set up, they make for simple programming and speedy displays.

Pianorgan's first few modules build sixteen character strings to display the instrument's singing keys. There are two strings for each of eight keys, one with a closed mouth and one with an open mouth.

Lines 1050-1070 set up two tabbing strings. D$ contains a home command and 23 cursor down commands. R$ contains 40 cursor right commands. Using these strings in combination with the LEFT$ function lets us move the cursor anywhere on the screen.

Lines 1120-1210 build up eight closed mouth strings. First, lines 1120-1140 build a section that's common to all eight strings. Line 1150 sets a piece that'll finish off all eight strings. Then lines 1160-1210 put together the eight custom strings.

Line 1170 adds the pieces of D$ and R$ that'll get the cursor to the proper starting position on the screen. The eight images will share the same vertical position. However,

```
1000 REM *** PIANORGAN ***
1010 :
1020 :
1030 REM ** SET UP TABBING STRINGS
1040 :
1050 D$ = "[graphics]"
1060 R$ = "[graphics]"
1070 R$ = R$ + R$
1080 :
1090 :
1100 REM ** SET UP CLOSED MOUTH STRINGS
1110 :
1120 CM$ = "   [graphics]- -[graphics] - [graphics]"
1130 CM$ = CM$ + " - [graphics]"
1140 CM$ = CM$ + "   [graphics] "
1150 FP$ = " [graphics]      "
1160 FOR N = 1 TO 8
1170 :   CM$(N) = LEFT$(D$,4) +
              LEFT$(R$, 5*N - 4)
1180 :   CM$(N) = CM$(N) + CM$
1190 :   CM$(N) = CM$(N) + CHR$(48 + N)
1200 :   CM$(N) = CM$(N) + FP$
1210 NEXT N
1220 :
1230 :
1240 REM ** SET UP OPEN MOUTH STRINGS
1250 :
1260 PM$ = "[graphics]o o[graphics] - [graphics] ● [graphics]"
1270 PM$ = PM$ + "[graphics] "
1280 PM$ = PM$ + "[graphics]    [graphics] "
1290 FOR N = 1 TO 8
1300 :   PM$(N) = LEFT$(D$,4) +
              LEFT$(R$, 5*N - 4)
1310 :   PM$(N) = PM$(N) + PM$
1320 :   PM$(N) = PM$(N) + CHR$(48 + N)
1330 :   PM$(N) = PM$(N) + FP$
1340 NEXT N
1350 :
1360 :
1370 REM ** SET UP COLOR CODES
1380 :
1390 FOR N = 1 TO 8   :REM TO COLOR KEYS
1400 :    READ HU(N)
1410 NEXT N
1420 :
1430 DATA 14, 4, 3, 7, 12, 5, 8, 1
1440 :
```

```
1450 :
1460 REM ** SET UP SID AND FREQUENCIES
1470 :
1480 SID = 54272       :REM SOUND CHIP
1490 POKE SID+3, 4     :REM PULSE WIDTH
1500 POKE SID+5, 10    :REM ATK=0, DKY=10
1510 POKE SID+6, 169   :REM SST=10, RLS=9
1520 POKE SID+24,15    :REM MAX VOLUME
1530 WF = 64           :REM PULSE WVF
1540 :
1550 FOR N=1 TO 8      :REM SET FREQUENCY
1560 :   READ FH(N)    :REM VALUES FOR
1570 :   READ FL(N)    :REM 8 NOTES
1580 NEXT N
1590 :
1600 DATA 8, 98, 9, 104
1610 DATA 10, 143, 11,  48
1620 DATA 12, 143, 14,  25
1630 DATA 15, 210, 16, 195
1640 :
1650 :
1660 REM ** SET SCREEN COLORS, ALL KEYS
            REPEAT, & SPEED UP KBD SCAN
1670 :
1680 POKE 53280, 0   :REM BORDER BLACK
1690 POKE 53281, 0   :REM BKGROUND BLACK
1700 POKE 650, 128   :REM ALL KEYS REPT.
1710 POKE 56325, 20  :REM SPEEDIER SCAN
1720 :
1730 :
1740 REM ** PRINT 8 CLOSED MOUTHS
1750 :
1760 PRINT "[CLR]";         :REM CLEAR SCREEN
1770 PRINT "[GRY1]"         :REM DARK GRAY
1780 FOR N = 1 TO 8
1790 :   PRINT CM$(N)    :REM THE MOUTHS
1800 NEXT N
1810 PRINT "[WHT]"          :REM WHITE
1820 :
1830 :
1840 REM ** PRINT PROMPTS
1850 :
1860 PRINT LEFT$(D$,18); SPC(9);
1870 PRINT "PRESS KEYS [RVS ON]1[RVS OFF]-[RVS ON]8[RVS OFF] TO PLAY"
1880 PRINT : PRINT SPC(9);
1890 PRINT "PRESS [RVS ON]SPACEBAR[RVS OFF] TO STOP"
1900 :
1910 :
1920 REM ** SCAN THE KEYBOARD
```

```
1930 :
1940 GET KP$
1950 IF KP$ = "" THEN 1940
1960 IF KP$ = " " THEN 2200
1970 KP = VAL (KP$)
1980 IF KP<1 OR KP>8 THEN 1940
1990 :
2000 :
2010 REM ** PLAY A NOTE
2020 :
2030 POKE 646, HU(KP)   :REM SET CHAR HU
2040 PRINT PM$(KP)      :REM OPEN MOUTH
2050 POKE SID+1,FH(KP)  :REM SET FREQ
2060 POKE SID,FL(KP)    :REM SET FREQ
2070 POKE SID+4, WF+1   :REM START SOUND
2080 :
2090 GET KP$ :REM PLAY TIL KEY RELEASED
2100 IF VAL(KP$) = KP THEN 2090
2110 :
2120 POKE 646, 11      :REM BACK TO GRAY
2130 PRINT CM$(KP)     :REM CLOSE MOUTH
2140 POKE SID+4, WF    :REM END SOUND
2150 GOTO 1950         :REM SCAN AGAIN
2160 :
2170 :
2180 REM ** CLEAN UP AND GO HOME
2190 :
2200 POKE 56325, 66    :REM FIX KBD SCAN
2210 POKE 646, 1   :REM CHAR COLOR WHITE
2220 PRINT "♥";    :REM CLEAR SCREEN
2230 FOR REG=SID TO SID+24    :REM CLEAR
2240 :    POKE REG, 0         :REM SID
2250 NEXT REG
2260 :
2270 END
```

Fig. 10-3. Listing of the program Pianorgan.

each one will have a different horizontal position.

Line 1180 adds the common section built in lines 1120-1140. Then line 1190 uses a cheap trick to add a number to each image. The singing keys have number codes, 1-8. When keyboard keys 1-8 are pressed, the appropriate single key will pop into action. The character codes for numbers run between 48 and 57. Line 1190 simply adds the value of the loop variable N to 48 and then uses the CHR$ function to produce the character that corresponds to the value of N. For example, when N has the value 4, line 1190 will add on CHR$

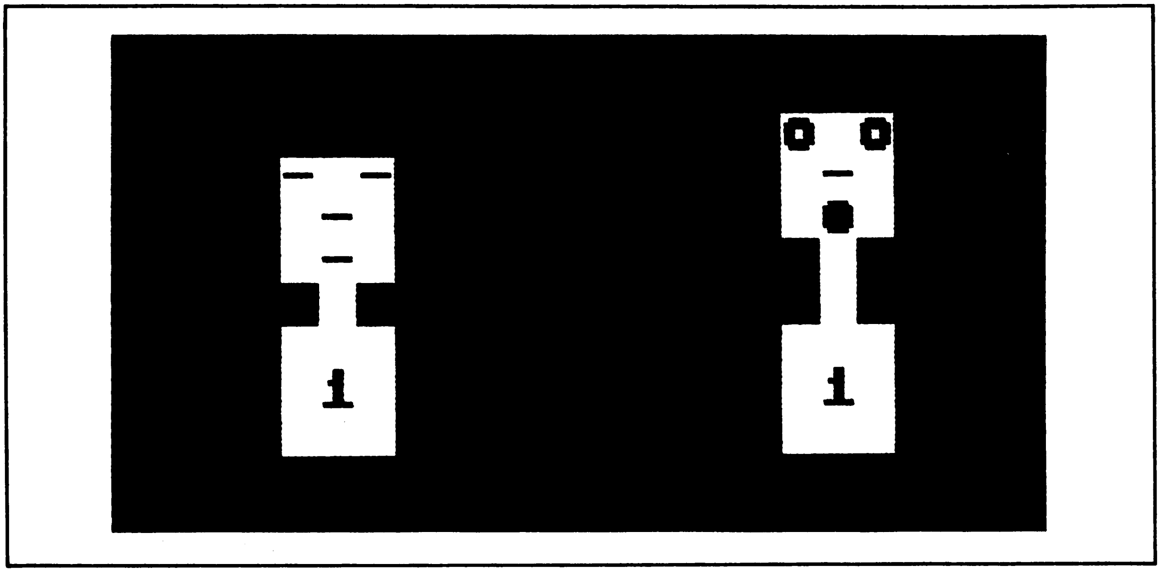



Fig. 10-4. The two singing key images: closed mouth and open mouth.

(52), which is a 4.

After the closed mouth strings are set, lines 1260-1340 set up eight open mouth strings. The process is similar to that in lines 1120-1210. The major differences are the details of the image. Figure 10-4 shows the two different singing key images, one with a closed mouth and the other with an open mouth.

This section's final module stores eight color codes in the array HU ( ). Remember, the singing keys are numbered 1-8. Each key's color code will be used to set the color of that key's open mouth image.

### 10.4.2 Setting Up SID, the Screen, and the Keyboard

This program uses the pulse waveform and a carefully chosen ADSR envelope to create sounds midway between a piano and an organ. Lines 1480-1530 set the necessary SID registers.

Lines 1550-1630 set up two arrays, FH ( ) and FL ( ), that will hold the frequency settings for eight notes. The values in the data statements come from Appendix O. They'll produce the notes C, D, E, F, G, A, and B from the third octave, and C from the fourth octave.

Next, lines 1680 and 1690 set the screen background and border to black. I have a definite preference for a black background, since colors really sing when displayed on it. In this program I decided to enforce my preference.

Line 1700 pulls a stunt you've used before. When memory location 650 contains the value 128, all keys on the keyboard will repeat when held down long enough.

Line 1710 pulls a new trick. One of the joys of working with the Commodore 64 is the measure of control you have over hardware configuration. Normally, the Commodore 64 scans the keyboard for pressed keys 60 times a second. In Pianorgan, you need to scan it more often to get a more responsive instrument. Memory location 56325 is a register that controls the speed of keyboard scanning. Normally, it contains the value 66. By poking it with the value 20, you can get the computer to scan the keyboard 200 times a second. At the end of the program, you'll set it back to normal scan speed. If you didn't, strange things would occur. Try it, if you've got a taste for strangeness.

### 10.4.3 Set the Initial Display

The next two modules of pianorgan are straightforward. Lines 1760-1810 clear the screen and then print the eight closed mouth strings in dark gray. Then lines 1860-1890 print some instructions for playing the instrument. Remember, those weird-looking characters in lines 1770 and 1810 represent color commands. Check back to "How To Use This Book" or Appendix E if you've forgotten about them.

### 10.4.4 The Main Program Loop of Pianorgan

Now comes Pianorgan's main program loop. Lines 1940-1980 scan the keyboard. A space will end the program; one of the number keys in the range 1 to 8 will trigger a note; anything else will be ignored.

Lines 2030-2150 play a note. This section of the program is relatively short and simple, thanks to all the setup work the program did earlier. Line 2030 starts the process by setting a new color. Memory location 646 is used by the Commodore's operating system to figure out what color to draw characters. Then line 2040 draws an open mouth image. The color and the open mouth string correspond to the number of the key that's been pressed. Lines 2050-2060 then set the note's frequency, and line 2070 triggers the sound.

The ADSR envelope for Pianorgan's sounds has a fast attack rate, a fairly slow decay rate, and a sustain level that's about two-thirds of peak volume. The release rate's pretty close to the attack rate. If a note is held for a short time, it will sound like a piano note. The longer the note's held, the more it will sound like an organ note.

Lines 2090-2100 are the reason we speeded up the keyboard scan. First, line 2090 gets a keypress and stores it in the variable KP$. If a key's being held down, the value of KP$ will match KP, the number of the note currently being played. In that case, the program does a quick U-turn back to 2090 to read the keyboard again. As soon as the key's let up, line 2100's matching test will fail, and the program will go on to end the note. With a normal keyboard scan rate, these two lines wouldn't work correctly; the get procedure takes too much time, and it would miss a lot of key action. The speeded-up scan rate solves the problem.

The next four lines finish off the note. Line 2120 sets the drawing color back to dark gray. Lines 2130 draws the appropriate closed mouth image. Line 2140 releases the sound, and then line 2150 jumps on back to line 1950 to check for new keypresses.

### 10.4.5 Closing Thoughts

As mentioned in Section 10.4.3, pressing the spacebar ends Pianorgan. Lines 2200-2250 clean up shop. First, line 2200 restores the normal keyboard scan rate. Line 2210 sets the character color to white ; line 2220 clears the screen; and lines 2230-2250 play an homage to thoroughness by resetting the first 24 SID registers.

There are a number of things you can try to do with this program. You might want to add more keys to the instrument, use different images, add more voices, change the style of animation, or vary the keyboard action. Commodore has put some great hardware into your computer; with clever software, you can create animated musical instruments never before seen or heard.

## 10.5 SOME THOUGHTS ABOUT SOUND/IMAGE COORDINATION

There is a marvelous Charlie Chaplin movie anyone interested in sound/image coordination should see. It's called City Lights. Charlie Chaplin had become an expert movie maker during the days of silent films. He got so good at his craft that you could almost hear sounds in those silent films. City Lights was one of the first films he made with sound.

The sound in that film is used sparingly, cleverly, and to great effect. Chaplin was a master of comic and dramatic timing; he was able to transfer those skills to his work with sound. Often a sound comes earlier than expected, telegraphing a forthcoming action. Sometimes it comes a bit late, increasing the excitement of a scene. He uses sound sparingly, not wanting to clog the audience's taste for it.

The coordination of sounds and images doesn't have to be perfect. Often, subtle offsets can add to the desired effect. Let the minds of your audience do some of the work. Artists, magicians, and master filmmakers understand this. Some of the better computer programmers are starting to learn the same principles.

## 10.6 THE FINAL PROGRAM: SEESAW

Figure 10-5 lists our final program, Seesaw. Type it in, save it, run it, and play around with it. When you finish, come on back for some explanation.

Two strange creatures appear, one suspended from a sky hook, the other poised on a seesaw. When you press the A key, for Action, the sky hook releases its captive, who moves with a falling whistle towards the ground. She/he hits with a ringing vibration, and the other creature gets launched into the air. This creature also moves with a whistle, but now the tone rises until it's cut short by the kerchunk of the sky hook snapping shut on the hapless beast. When the dust clears, the two creatures have traded situations. This happens every time you press A. Pressing the spacebar ends the program.

### 10.6.1 Setting Up Strings, Sprites, and Sounds

Like the other programs in this chapter, Seesaw takes quite a bit of setting up. Each element is prepared in its own module. Lines 1050-1140 set up four hook images: an open and a closed hook for each of the two hook positions. Each hook image is a large string, built up out of all the fancy characters in the Commodore's arsenal: color changers, cursor controls, display options, and graphics char-

```
1000 REM *** SEESAW ***
1010 :
1020 :
1030 REM ** SET UP HOOK STRINGS
1040 :
1050 H1$ = "▛──┬──▜██████▟  ┌──┴──┐  "
1060 H1$ = H1$ + "▜█████████  ┼─  ─┼   ▙"
1070 H2$ = "▛──┬──▜██████▞┌────┴────┐"
1080 H2$ = H2$ + "▜█████████├─┼      ┼─┤▙"
1090 PL$ = "▚▜████████████"
1100 R$ = "█████████████████"
1110 PH$(1,1) = PL$ + H1$
1120 PH$(1,2) = PL$ + H2$
1130 PH$(2,1) = PL$ + R$ + H1$
1140 PH$(2,2) = PL$ + R$ + H2$
1150 :
1160 :
1170 REM ** SET UP SEESAW STRINGS
1180 :
1190 SS$(1) = "▛‾‾‾‾‾‾‾‾‾────────●────────______▙"
1200 SS$(2) = "▛________────────●────────‾‾‾‾‾‾▙"
1210 T$ = "▚▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜▜███████████"
1220 SS$(1) = T$ + SS$(1)
1230 SS$(2) = T$ + SS$(2)
1240 :
1250 :
1260 REM ** LOAD IN SPRITE IMAGE
1270 :
1280 FOR N = 12288 TO 12350
1290 :   READ SPDTA
1300 :   POKE N, SPDTA
1310 NEXT N
1320 :
1330 DATA   0, 255,   0,   1, 129, 128
1340 DATA   3,   0, 192,   6,   0,  96
1350 DATA  12,   0,  48,  24, 231,  24
1360 DATA  48, 165,  12,  32, 231,   4
1370 DATA  32,   0,   4,  32,  36,   4
1380 DATA  38,  60, 100,  35, 129, 196
1390 DATA  48, 231,  12,  24,  60,  24
1400 DATA  14,   0, 112,   3, 255, 192
1410 DATA   0, 129,   0,   0, 129,   0
1420 DATA   0, 129,   0,   0, 129,   0
1430 DATA   3, 231, 192
1440 :
1450 :
1460 REM ** PRINT PROMPTS
```

```
1470 :
1480 POKE 53281, 0      :REM BKGRND BLACK
1490 PRINT "{CLR}{22 DOWN}"
1500 PRINT SPC(10); "PRESS {RVS ON}A{RVS OFF} ";
1510 PRINT "FOR ACTION"
1520 PRINT
1530 PRINT SPC(9); "PRESS ";
1540 PRINT "{RVS ON}SPACEBAR{RVS OFF} TO END"
1550 :
1560 :
1570 REM ** SET UP SPRITES
1580 :
1590 VIC = 53248      :REM GRAPHICS CHIP
1600 POKE 2040, 192 :REM SPRITE 0 PNTR
1610 POKE 2041, 192 :REM SPRITE 1 PNTR
1620 :
1630 POKE VIC, 92    :REM #0 INIT HR POS
1640 POKE VIC+1, 77 :REM #0 INIT VR POS
1650 POKE VIC+2, 220 :REM #1 INIT HR PS
1660 POKE VIC+3, 150 :REM #1 INIT VR PS
1670 :
1680 POKE VIC+39, 4 :REM #0 STARTS PRPL
1690 POKE VIC+40, 3 :REM #1 STARTS CYAN
1700 POKE VIC+23, 3 :REM EXPAND VERTICL
1710 POKE VIC+29, 3 :REM EXPAND HORIZNT
1720 :
1730 POKE VIC+21, 3 :REM SPRITES 0-1 ON
1740 :
1750 :
1760 REM ** INITIALIZE SID
1770 :
1780 SID = 54272          :REM SOUND CHIP
1790 FOR REG = SID TO SID+24
1800 :    POKE REG, 0      :REM CLEAR IT
1810 NEXT REG
1820 POKE SID+24, 15      :REM MAX VOLUME
1830 :
1840 :
1850 REM ** SET VOICE 1 FOR GONG
1860 :
1870 POKE SID+1, 5    :REM V-1 FREQ
1880 POKE SID+5, 11   :REM ATK=0, DKY=11
1890 POKE SID+6, 10   :REM SST=0, RLS=10
1900 :
1910 :
1920 REM ** SET VOICE 2 FOR
            WHISTLING FLIGHT
1930 :
```

```
1940 POKE SID+12, 12 :REM ATK=0, DKY=12
1950 :
1960 :
1970 REM ** SET VOICE 3 FOR HOOK CLICK
1980 :
1990 POKE SID+15, 21 :REM V-3 FREQ
2000 POKE SID+20,192 :REM SST=12, RLS=0
2010 :
2020 :
2030 REM ** INITIALIZE HOOKS, SEESAW
2040 :
2050 FH = 1           :REM HOOK 1 IS FULL
2060 EH = 2           :REM HOOK 2 EMPTY
2070 PRINT PH$(FH, 1) :REM PRINT HOOK 1
2080 PRINT PH$(EH, 2) :REM PRINT HOOK 2
2090 PRINT SSW$(1)    :REM PRINT SEESAW
2100 :
2110 :
2120 REM ** SCAN KEYBOARD
2130 :
2140 GET KP$
2150 IF KP$ = "" THEN 2140  :REM SCAN
2160 IF KP$ = "A" THEN 2230 :REM ACTION
2170 IF KP$ = " " THEN 2980 :REM END IT
2180 GOTO 2140 :REM OTHER KEYS FILTERED
2190 :
2200 :
2210 REM ** RELEASE A SPRITE
2220 :
2230 POKE SID+18,129  :REM START CLICK
2240 PRINT PH$(FH,2)  :REM HOOK OPENS
2250 FOR DL = 1 TO 40 : NEXT DL
2260 POKE SID+18, 128  :REM END CLICK
2270 POKE VIC + FH + 38, 3 :REM GO CYAN
2280 :
2290 :
2300 REM ** RELEASED SPRITE DROPS
2310 :
2320 POKE SID+8,80    :REM V-2 INIT FRQ
2330 POKE SID+11,17   :REM WHISTLE ON
2340 FOR N = 78 TO 145
2350 :    POKE VIC+(FH*2)-1, N :REM DROP
2360 :    POKE SID+8, 158 - N  :REM WISL
2370 NEXT N
2380 POKE SID+11,16   :REM WHISTLE OFF
2390 :
2400 :
2410 REM ** SEESAW ACTION
```

```
2420 :
2430 POKE SID+4,21      :REM START GONG
2440 PRINT SSW$(3-FH) :REM MOVE SEESAW
2450 POKE VIC+(FH*2)-1,150 :REM MOVE
2460 POKE VIC+(EH*2)-1,146 :REM SPRITES
2470 POKE SID+4,20      :REM RELEASE GONG
2480 :
2490 :
2500 REM ** VIBRATE FALLEN SPRITE
2510 :
2520 HR = VIC + (FH*2) - 2 :REM HOR REG
2530 HP = PEEK (HR)        :REM HOR POS
2540 CR = VIC + FH + 38    :REM COLR RG
2550 FOR VB = 1 TO 5       :REM 6 VIBES
2560 :   POKE HR, HP - 4   :REM GO LEFT
2570 :   POKE CR, 1        :REM GO WHIT
2580 :   POKE SID+1, 6     :REM HI FREQ
2590 :   POKE HR, HP       :REM GO MIDL
2600 :   POKE CR, 2        :REM GO RED
2610 :   POKE SID+1, 4     :REM LO FREQ
2620 :   POKE HR, HP + 4   :REM GO RGHT
2630 :   POKE CR, 7        :REM GO YELO
2640 :   POKE SID+1, 5     :REM MID FRQ
2650 NEXT VB
2660 POKE HR, HP  :REM RESTORE POSITION
2670 POKE CR, 3   :REM RESTORE COLOR
2680 :
2690 :
2700 REM ** SEESAWED SPRITE RISES UP
2710 :
2720 POKE SID+8,80      :REM V-2 INIT FRQ
2730 POKE SID+11,17     :REM WHISTLE ON
2740 FOR N = 145 TO 77 STEP -1
2750 :   POKE VIC+(EH*2)-1, N :REM RISE
2760 :   POKE SID+8, 158 - N  :REM WISL
2770 NEXT N
2780 POKE SID+11,16     :REM WHISTLE OFF
2790 :
2800 :
2810 REM ** CAPTURE A SPRITE
2820 :
2830 POKE SID+18,129    :REM START CLICK
2840 PRINT PH$(EH,1)    :REM HOOK CLOSES
2850 FOR DL = 1 TO 40 : NEXT DL
2860 POKE SID+18, 128  :REM END CLICK
2870 POKE VIC + EH + 38, 4 :REM GO PRPL
2880 :
2890 :
2900 REM ** SWITCH FH & EH, GO BACK
```

```
2910 :
2920 TEMP = FH : FH = EH : EH = TEMP
2930 GOTO 2140
2940 :
2950 :
2960 REM ** END IT, CLEAN UP, GO HOME
2970 :
2980 POKE VIC+21,0  :REM SPRITES OFF
2990 POKE SID+24,0  :REM VOLUME OFF
3000 POKE VIC+23,0  :REM VERT EXPAND OFF
3010 POKE VIC+29,0  :REM HORZ EXPAND OFF
3020 PRINT "♥";     :REM CLEAR SCREEN
3030 :
3040 END
```

Fig. 10-5. Listing of the program Seesaw.

characters. Parts common to all four hooks are built up and then combined by lines 1110-1140 into the four strings.

Similar techniques are used in lines 1190-1230 to set up two seesaw images. It took some experimentation to find the keys that would print out line pieces that gradually rose and fell. As with the hook images, cursor commands and color controls are included in the strings; placing the seesaws in the correct screen position becomes a snap.

The same data is used to create both sprites. Lines 1280-1310 load the data in. The data itself is stored in lines 1330-1430.

Lines 1480-1540 print the screen prompts —very straightforward stuff. Then lines 1590-1730 give the sprites their initial VIC settings. Rather than try to calculate the exact sprite positions, I started with an estimate and then used intelligent searching techniques (trial and error) to home in on the right values.

The images are set, so it's time to prepare the sounds. SID's first voice will be used for the gong; its second voice will provide whistling flights; and the third voice will create the clunking hook effects. Lines 1780-1820 clear the 24 important SID registers and set maximum volume. Then lines 1870-2000 poke in the values needed to sculpt the three sounds.

Once the program gets going, two variables will be used to keep track of the hook and creature situation. FH will contain the number of the hook that's holding a creature, and EH will hold the number of the empty hook. Hook 1 and creature 1 are on the left; hook 2 and creature 2 are on the right.

Lines 2050-2060 initialize these variables. Then lines 2070-2090 draw the appropriate hook and seesaw images. The stage is now set.

### 10.6.2 Action Breakdown

Lines 2140-2180 form a familiar keyboard-scanning module. Keys other than A or the spacebar are ignored. Pressing A initiates and action cycle; pressing the spacebar ends the program.

The action cycle breaks down into six modules: First, the creature held in a sky hook is released. Second, it drops down whistling. Third, it hits the seesaw, which switches positions, along with the two creatures. Fourth, the recently-fallen creature vibrates. Fifth, the other sprite rises up into the air, whistling. Sixth, the rising sprite gets nabbed by its hook.

Lines 2230-2270 take care of releasing a sprite. The sky hook noise begins, the hook opens, there's a short delay, the noise ends; and the sprite changes color.

Lines 2320-2380 drop the sprite. First, an initial sound frequency gets set, and the whistling sound starts. Then a loop moves the sprite down the screen, dropping the frequency as the sprite drops. At the bottom, the whistling stops. It has also slowly faded in volume during the trip, thanks to a carefully chosen rate of volume decay.

Then the falling sprite reaches the seesaw, and you're ready for the third part of the action sequence. A gong noise is initiated; the seesaw tilts; the sprite moves; and the gong noise begins a slow fadeout. All of this occurs in lines 2430-2470.

Next, lines 2520-2670 vibrate the fallen sprite. As the frequency of the gong shifts up and down the scale, the sprite moves back and forth horizontally and shifts colors. This activity is repeated several times. Then, as the clanging gong fades away, the shaken creature comes to rest, restored to a healthy cyan color.

Now comes the fifth module of the action cycle. The other sprite rises into the air. Compare lines 2720-2780 to lines 2320-2380, which dropped the hanging sprite creature. The two modules are very much alike. First, voice #2 gets an initial frequency. Then the sound is gated. The module's main loop comes next. As the sprite moves up the screen, voice #2's frequency rises. Finally, at the top, the whistling sound is released.

Now comes the sixth part of the action. Just as a sprite was released in the first part, now the rising sprite is captured. It all happens in lines 2830-2870. The hook noise begins; the hook clamps shut; there's a bit of a delay; the hook noise ends; the sprite is drained of freedom's color.

The action's over, and the sprites have exchanged situations. The empty hook is now full, the once-full hook is empty. Line 2920 updates the variables EH and FH to reflect those sobering facts, and then line 2930 bounces back to read the keyboard again.

### 10.6.3 Cleanup and Reflection

Lines 2980-3020 perform a standard cleanup operation. You might choose to be more thorough about resetting the SID and VIC registers.

When I wrote this program, the broad outlines of the action were implemented first. Fine-tuning the sounds and sprite motions was saved for last. This method of problem solving worked well with Seesaw.

## 10.7 SOME LAST THOUGHTS ABOUT COMBINING SOUND AND GRAPHICS

Before I fade into the final end-of-chapter exercises, here are some things to keep in mind when you're combining sound and graphics:

Timing — A very simple effect can have a solid im-

| | |
|---|---|
| | pact when it comes at the right moment. |
| Fine Tuning | When every element fits seamlessly into the whole effect, synergy is maximized. |
| Simplicity | Remove excess decoration. Every sound and image should have a clear purpose. |
| Unity of Design | The individual elements must aid one another. |

There's a lot of sound and graphics magic waiting inside your Commodore 64. Start waving your wand.

## 10.8 CHAPTER SUMMARY

In this chapter you explored three programs that mix sound and graphics. More specifically, I explained:

* How to cultivate synergy, so that the whole effect of a graphics/sound combination is greater than the sum of the individual parts
* Techniques that are useful for solving complex programming tasks
* The program Bouncer, which mixes character and sprite graphics with simple sound effects and introduces a simple wall-bouncing technique
* The program Pianorgan, which uses complex character strings and a speeded up keyboard scan to create an animated musical instrument
* Coordinating sounds and images in subtle, artistic ways
* The program Seesaw, with a complicated set of actions involving all three SID voices, two sprites, and complex character strings

I hope you've enjoyed our excursions into sound and graphics on the Commodore 64. Stay curious, keep on learning, and have fun!

## 10.9 EXERCISES

### 10.9.1 Self Test

My favorite answers can be found in Section 10.9.3.

1. (10.1) When the whole becomes greater then the sum of the parts, you can tell it __________.
2. (10.2) Breaking a complex programming task down into successively simpler pieces is known as __________________.
3. (10.3) Parts of a picture that stay still are known as __________ elements, and parts that move are __________ elements.
4. (10.3) The program Bouncer uses ______ expressions to check for blip/wall collisions.
5. (10.4) Speeding up the __________ scan in Pianorgan gives us a more responsive musical instrument.
6. (10.6) In Seesaw, the complex action cycle has been broken into __________ smaller modules.

### 10.9.2 Programming Exercises

1. Change the program Bouncer so it makes noises in a more regular pattern when the sprite bounces into walls.
2. Change the program Pianorgan so the heads shimmer colorfully when they sing.
3. Change the program Seesaw so the creatures move vertically as well as horizontally when they hit the seesaw.

### 10.9.3 Answers to Self Test

1. synergy
2. top down structured programming
3. static; dynamic
4. Boolean
5. keyboard
6. six

### 10.9.4 Possible Solutions to Programming Exercises

1. Load in the program Bouncer. Then type in these lines:

```
1000 REM *** ROLLER BOUNCER ***
1463 :
1465 FQ = 10   :REM STARTING FREQUENCY
1468 FC = 1.3 :REM FREQ CHANGE FACTOR
1812 FQ = FQ * FC
1814 IF FQ > 100 THEN FC =  0.6
1816 IF FQ < 10  THEN FC =  1.3
1818 :
1830 POKE SID+1, FQ
```

2. Load in the program Pianorgan. Then type in these lines:

```
1000 REM *** RAINBORGAN ***
1275 JM$ = PM$
1305 :   JM$(N) = PM$(N) + JM$
2093 POKE 646,((PEEK(646)+1)AND 15)OR 1
2096 PRINT JM$(KP)
```

3. Load in the program Seesaw. Then type in these lines:

```
1000 REM *** MORE SEESAW ***
2533 VR = HR + 1
2535 VP = PEEK (VR)
2645 :    POKE VR, VP - VB*2
2740 FOR N = 145 TO 77 STEP -1.6
2765 :    POKE VR,
          VP + (N>115) * (N/3 - 38)
```

# Appendices

## Appendix A

# VIC Register Layout

VIC starting address is 53248 ($D000)

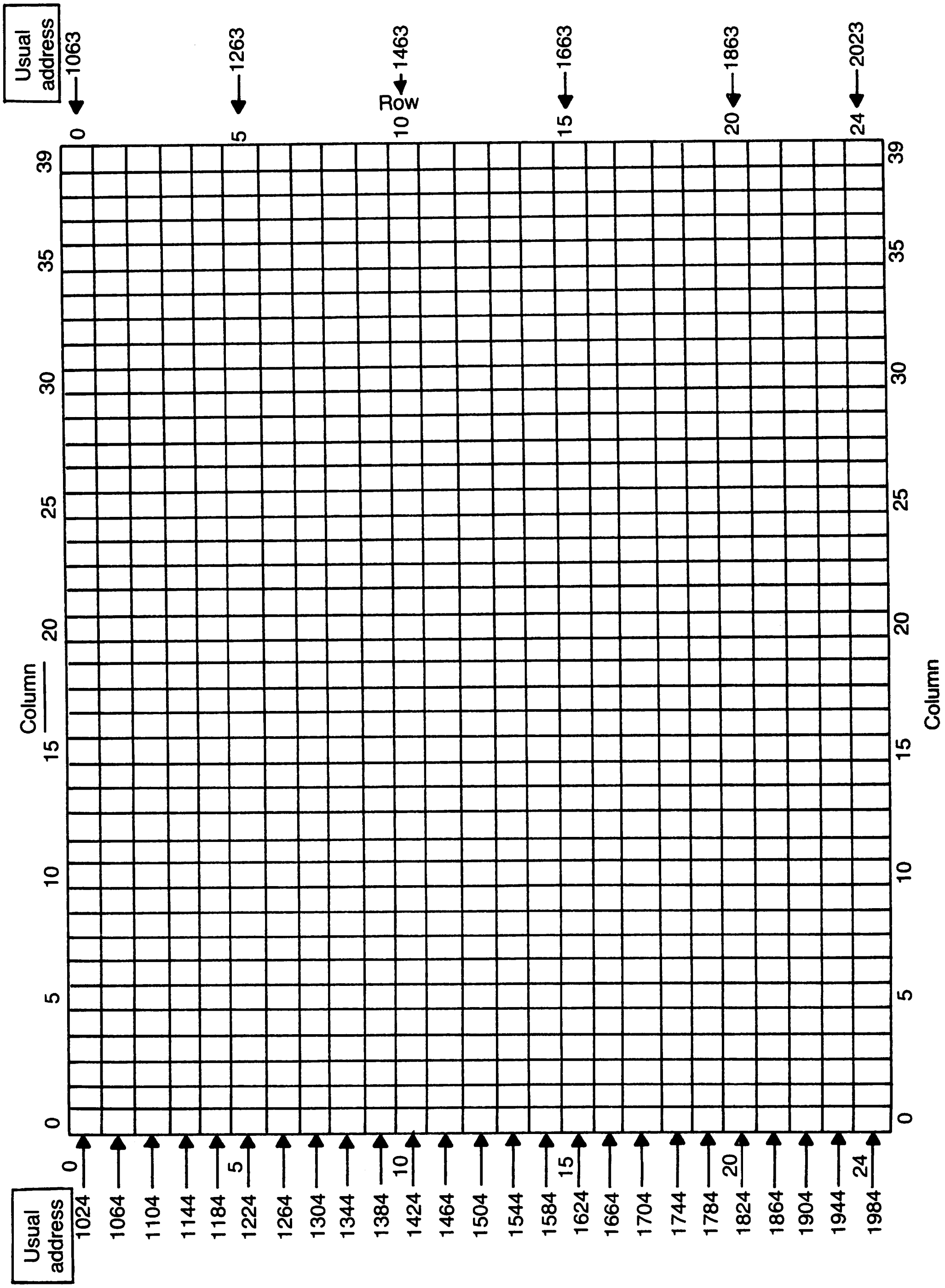
| Register number Decimal | Register number Hex | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | This register controls: |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | $00 | S0 H7 | S0 H6 | S0 H5 | S0 H4 | S0 H3 | S0 H2 | S0 H1 | S0 H0 | Sprite #0 horizontal position |
| 1 | $01 | S0 V7 | S0 V6 | S0 V5 | S0 V4 | S0 V3 | S0 V2 | S0 V1 | S0 V0 | Sprite #0 vertical position |
| 2 | $02 | S1 H7 | S1 H6 | S1 H5 | S1 H4 | S1 H3 | S1 H2 | S1 H1 | S1 H0 | Sprite #1 horizontal position |
| 3 | $03 | S1 V7 | S1 V6 | S1 V5 | S1 V4 | S1 V3 | S1 V2 | S1 V1 | S1 V0 | Sprite #1 vertical position |
| 4 | $04 | S2 H7 | S2 H6 | S2 H5 | S2 H4 | S2 H3 | S2 H2 | S2 H1 | S2 H0 | Sprite #2 horizontal position |
| 5 | $05 | S2 V7 | S2 V6 | S2 V5 | S2 V4 | S2 V3 | S2 V2 | S2 V1 | S2 V0 | Sprite #2 vertical position |
| 6 | $06 | S3 H7 | S3 H6 | S3 H5 | S3 H4 | S3 H3 | S3 H2 | S3 H1 | S3 H0 | Sprite #3 horizontal position |
| 7 | $07 | S3 V7 | S3 V6 | S3 V5 | S3 V4 | S3 V3 | S3 V2 | S3 V1 | S3 V0 | Sprite #3 vertical position |
| 8 | $08 | S4 H7 | S4 H6 | S4 H5 | S4 H4 | S4 H3 | S4 H2 | S4 H1 | S4 H0 | Sprite #4 horizontal position |
| 9 | $09 | S4 V7 | S4 V6 | S4 V5 | S4 V4 | S4 V3 | S4 V2 | S4 V1 | S4 V0 | Sprite #4 vertical position |
| 10 | $0A | S5 H7 | S5 H6 | S5 H5 | S5 H4 | S5 H3 | S5 H2 | S5 H1 | S5 H0 | Sprite #5 horizontal position |
| 11 | $0B | S5 V7 | S5 V6 | S5 V5 | S5 V4 | S5 V3 | S5 V2 | S5 V1 | S5 V0 | Sprite #5 vertical position |
| 12 | $0C | S6 H7 | S6 H6 | S6 H5 | S6 H4 | S6 H3 | S6 H2 | S6 H1 | S6 H0 | Sprite #6 horizontal position |
| 13 | $0D | S6 V7 | S6 V6 | S6 V5 | S6 V4 | S6 V3 | S6 V2 | S6 V1 | S6 V0 | Sprite #6 vertical position |
| 14 | $0E | S7 H7 | S6 H6 | S7 H5 | S7 H4 | S7 H3 | S7 H2 | S7 H1 | S7 H0 | Sprite #7 horizontal position |
| 15 | $0F | S7 V7 | S7 V6 | S7 V5 | S7 V4 | S7 V3 | S7 V2 | S7 V1 | S7 V1 | Sprite #7 vertical position |
| 16 | $10 | S7 H8 | S6 H8 | S5 H8 | S4 H8 | S3 H8 | S2 H8 | S1 H8 | S0 H8 | Most significant bit of horizontal positions |
| 17 | $11 | Raster bit 8 | Extended color text mode | Bit map mode | Blank screen | 24 or 25 rows of text | Vertical scroll bit 2 | Vertical scroll bit 1 | Vertical scroll bit 0 | Miscellaneous functions |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 18 | $12 | Raster bit 7 | Raster bit 6 | Raster bit 5 | Raster bit 4 | Raster bit 3 | Raster bit 2 | Raster bit 1 | Raster bit 0 | Raster register |
| 19 | $13 | LP H7 | LP H6 | LP H5 | LP H4 | LP H3 | LP H2 | LP H1 | LP H0 | Light pen horizontal position |
| 20 | $14 | LP V7 | LP V6 | LP V5 | LP V4 | LP V3 | LP V2 | LP V1 | LP V0 | Light pen vertical position |
| 21 | $15 | S7 On/off | S6 On/off | S5 On/off | S4 On/off | S3 On/off | S2 On/off | S1 On/off | S0 On/Off | Turn sprites on/off |
| 22 | $16 | — | — | Reset-always set to 0 | Multi-color mode | 38 or 40 columns of text | Horizontal scroll bit 2 | Horizontal scroll bit 1 | Horizontal scroll bit 0 | Miscellaneous functions |
| 23 | $17 | S7 EV | S6 EV | S5 EV | S4 EV | S3 EV | S2 EV | S1 EV | S0 EV | Expand sprite (2x) vertically |
| 24 | $18 | Text screen bit 3 | Text screen bit 2 | Text screen bit 1 | Text screen bit 0 | Char defs bit 2 | Char defs bit 1 | Char defs bit 0 | — | Memory pointers for character display, bit map, & screen |
| 25 | $19 | Interrupt from VIC | — | — | — | Light pen latched | Sprite to sprite collision | Sprite to bkgrnd collision | Raster count match | Interrupt register |
| 26 | $1A | — | — | — | — | Light pen latched | Sprite to sprite collision | Sprite to bkgrnd collision | Raster count match | Enable interrupts |
| 27 | $1B | S7 SBP | S6 SBP | S5 SBP | S4 SBP | S3 SBP | S2 SBP | S1 SBP | S0 SBP | Sprite to background priorities |
| 28 | $1C | S7 MCM | S6 MCM | S5 MCM | S4 MCM | S3 MCM | S2 MCM | S1 MCM | S0 MCM | Select multicolor mode for sprites |
| 29 | $1D | S7 EH | S6 EH | S5 EH | S4 EH | S3 EH | S2 EH | S1 EH | S0 EH | Expand sprite (2x) horizontally |
| 30 | $1E | S7 SSC | S6 SSC | S5 SSC | S4 SSC | S3 SSC | S2 SSC | S1 SSC | S0 SSC | Sprite to sprite collision |
| 31 | $1F | S7 SBC | S6 SBC | S5 SBC | S4 SBC | S3 SBC | S2 SBC | S1 SBC | S0 SBC | Sprite to back-ground collision |

VIC starting address is 53248 ($D000)

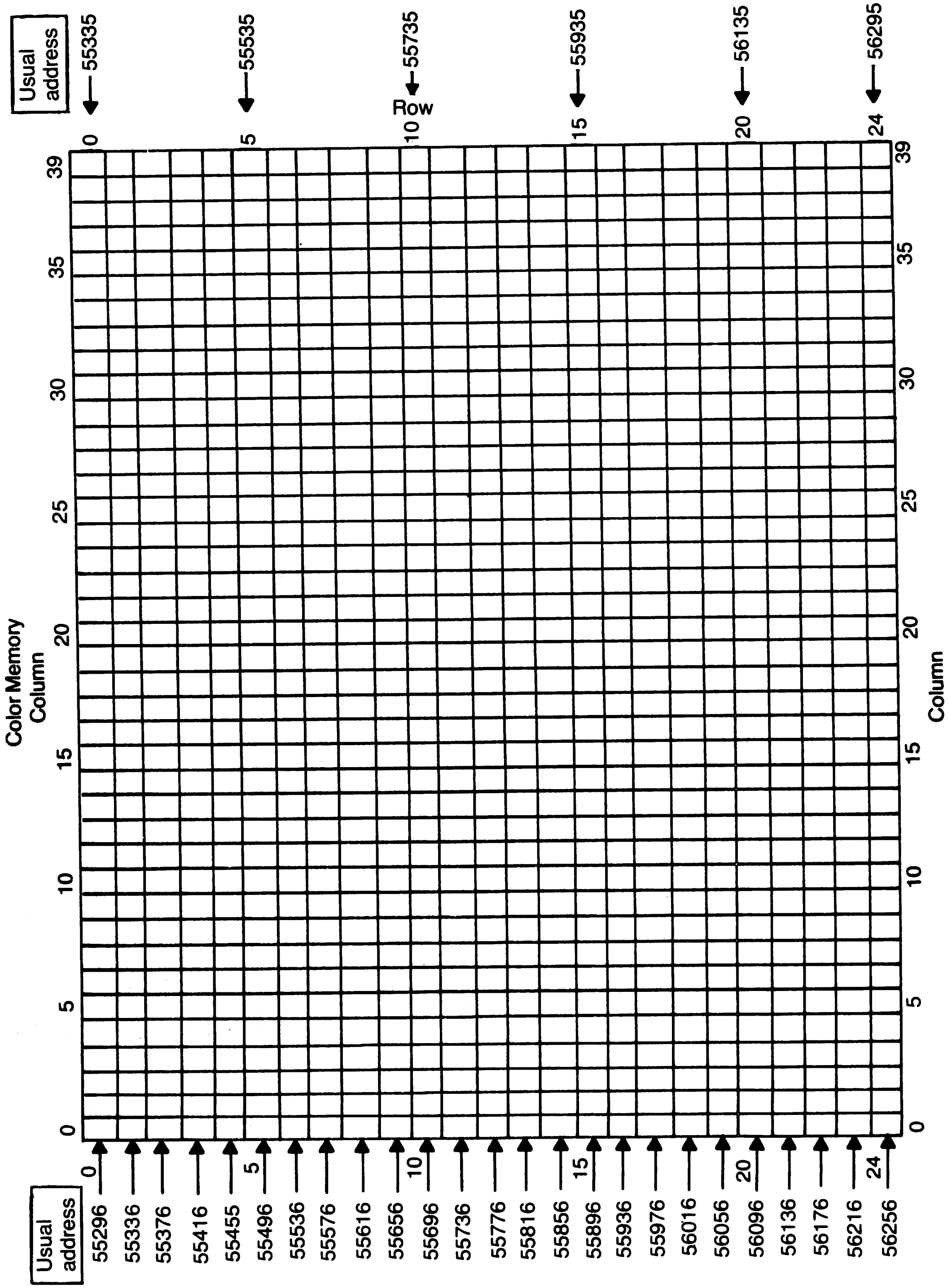
| Register number Decimal | Hex | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | This register controls: |
|---|---|---|---|---|---|---|---|---|---|---|
| 32 | $20 | – | – | – | – | Border C3 | Border C2 | Border C1 | Border C0 | Border color |
| 33 | $21 | – | – | – | – | Bkg 0 C3 | Bkg 0 C2 | Bkg 0 C1 | Bkg 0 C0 | Background #0 color |
| 34 | $22 | – | – | – | – | Bkg 1 C3 | Bkg 1 C2 | Bkg 1 C1 | Bkg 1 C0 | Background #1 color |
| 35 | $23 | – | – | – | – | Bkg 2 C3 | Bkg 2 C2 | Bkg 2 C1 | Bkg 2 C0 | Background #2 color |
| 36 | $24 | – | – | – | – | Bkg 3 C3 | Bkg C2 | Bkg 3 C1 | Bkg 3 C0 | Background #3 color |
| 37 | $25 | – | – | – | – | SMC 0 C3 | SMC 0 C2 | SMC 0 C1 | SMC 0 C0 | Sprite multicolor #0 |
| 38 | $26 | – | – | – | – | SMC 1 C3 | SMC 1 C2 | SMC 1 C1 | SMC 1 C0 | Sprite multicolor #1 |
| 39 | $27 | – | – | – | – | S0 C3 | S0 C2 | S0 C1 | S0 C0 | Sprite #0 color |
| 40 | $28 | – | – | – | – | S1 C3 | S1 C2 | S1 C1 | S1 C0 | Sprite #1 color |
| 41 | $29 | – | – | – | – | S2 C3 | S2 C2 | S2 C1 | S2 C0 | Sprite #2 color |
| 42 | $2A | – | – | – | – | S3 C3 | S3 C2 | S3 C1 | S3 C0 | Sprite #3 color |
| 43 | $2B | – | – | – | – | S4 C3 | S4 C2 | S4 C1 | S4 C0 | Sprite #4 color |
| 44 | $2C | – | – | – | – | S5 C3 | S5 C2 | S5 C1 | S5 C0 | Sprite #5 color |
| 45 | $2D | – | – | – | – | S6 C3 | S6 C2 | S6 C1 | S6 C0 | Sprite #6 color |
| 46 | $2E | – | – | – | – | S7 C3 | S7 C2 | S7 C1 | S7 C0 | Sprite #7 color |

# Appendix B
# Screen Memory


Usual address
1063
1263
1463
1663
1863
2023
Row
0
5
10
15
20
24
Column
0
5
10
15
20
25
30
35
39
Usual address
1024
1064
1104
1144
1184
1224
1264
1304
1344
1384
1424
1464
1504
1544
1584
1624
1664
1704
1744
1784
1824
1864
1904
1944
1984

## Appendix C

# Color Memory


Color Memory
Column
Row
Column
Usual address
55335
55535
55735
55935
56135
56295
Usual address
55296
55336
55376
55416
55455
55496
55536
55576
55616
55656
55696
55736
55776
55816
55856
55896
55936
55976
56016
56056
56096
56136
56176
56216
56256

## Appendix D

# Screen Display Codes

| Poke code | Set 1 | Set 2 | Poke code | Set 1 | Set 2 | Poke code | Set 1 | Set 2 | Poke code | Set 1 | Set 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | @ | @ | 128 | @ | @ | 64 | ─ | ─ | 192 | ─ | ─ |
| 1 | A | a | 129 | A | a | 65 | ♠ | A | 193 | ♠ | A |
| 2 | B | b | 130 | B | b | 66 | │ | B | 194 | │ | B |
| 3 | C | c | 131 | C | c | 67 | ─ | C | 195 | ─ | C |
| 4 | D | d | 132 | D | d | 68 | ─ | D | 196 | ─ | D |
| 5 | E | e | 133 | E | e | 69 | ─ | E | 197 | ─ | E |
| 6 | F | f | 134 | F | f | 70 | ─ | F | 198 | ─ | F |
| 7 | G | g | 135 | G | g | 71 | │ | G | 199 | │ | G |
| 8 | H | h | 136 | H | h | 72 | │ | H | 200 | │ | H |
| 9 | I | i | 137 | I | i | 73 | ╮ | I | 201 | ╮ | I |
| 10 | J | j | 138 | J | j | 74 | ╰ | J | 202 | ╰ | J |
| 11 | K | k | 139 | K | k | 75 | ╯ | K | 203 | ╯ | K |
| 12 | L | l | 140 | L | l | 76 | └ | L | 204 | └ | L |
| 13 | M | m | 141 | M | m | 77 | ╲ | M | 205 | ╲ | M |
| 14 | N | n | 142 | N | n | 78 | ╱ | N | 206 | ╱ | N |
| 15 | O | o | 143 | O | o | 79 | ┌ | O | 207 | ┌ | O |
| 16 | P | p | 144 | P | p | 80 | ┐ | P | 208 | ┐ | P |
| 17 | Q | q | 145 | Q | q | 81 | ● | Q | 209 | ● | Q |
| 18 | R | r | 146 | R | r | 82 | ─ | R | 210 | ─ | R |
| 19 | S | s | 147 | S | s | 83 | ♥ | S | 211 | ♥ | S |
| 20 | T | t | 148 | T | t | 84 | │ | T | 212 | │ | T |

| Poke code | Set 1 | Set 2 | Poke code | Set 1 | Set 2 | Poke code | Set 1 | Set 2 | Poke code | Set 1 | Set 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | U | u | 149 | | | 85 | | U | 213 | | |
| 22 | V | v | 150 | | | 86 | | V | 214 | | |
| 23 | W | w | 151 | | | 87 | | W | 215 | | |
| 24 | X | x | 152 | | | 88 | | X | 216 | | |
| 25 | Y | y | 153 | | | 89 | | Y | 217 | | |
| 26 | Z | z | 154 | | | 90 | | Z | 218 | | |
| 27 | [ | [ | 155 | | | 91 | + | + | 219 | | |
| 28 | £ | £ | 156 | | | 92 | | | 220 | | |
| 29 | ] | ] | 157 | | | 93 | | | 221 | | |
| 30 | ↑ | ↑ | 158 | | | 94 | | | 222 | | |
| 31 | ← | ← | 159 | | | 95 | | | 223 | | |
| 32 | | | 160 | | | 96 | | | 224 | | |
| 33 | ! | ! | 161 | | | 97 | | | 225 | | |
| 34 | " | " | 162 | | | 98 | | | 226 | | |
| 35 | # | # | 163 | | | 99 | | | 227 | | |
| 36 | $ | $ | 164 | | | 100 | | | 228 | | |
| 37 | % | % | 165 | | | 101 | | | 229 | | |
| 38 | & | & | 166 | | | 102 | | | 230 | | |
| 39 | ' | ' | 167 | | | 103 | | | 231 | | |
| 40 | ( | ( | 168 | | | 104 | | | 232 | | |
| 41 | ) | ) | 169 | | | 105 | | | 233 | | |
| 42 | * | * | 170 | | | 106 | | | 234 | | |

| Poke code | Set 1 | Set 2 |
|---|---|---|
| 43 | + | + |
| 44 | , | , |
| 45 | - | - |
| 46 | . | . |
| 47 | / | / |
| 48 | 0 | 0 |
| 49 | 1 | 1 |
| 50 | 2 | 2 |
| 51 | 3 | 3 |
| 52 | 4 | 4 |
| 53 | 5 | 5 |
| 54 | 6 | 6 |
| 55 | 7 | 7 |
| 56 | 8 | 8 |
| 57 | 9 | 9 |
| 58 | : | : |
| 59 | ; | ; |
| 60 | < | < |
| 61 | = | = |
| 62 | > | > |
| 63 | ? | ? |

| Poke code | Set 1 | Set 2 |
|---|---|---|
| 171 | | |
| 172 | | |
| 173 | | |
| 174 | | |
| 175 | | |
| 176 | | |
| 177 | | |
| 178 | | |
| 179 | | |
| 180 | | |
| 181 | | |
| 182 | | |
| 183 | | |
| 184 | | |
| 185 | | |
| 186 | | |
| 187 | | |
| 188 | | |
| 189 | | |
| 190 | | |
| 191 | | |

| Poke code | Set 1 | Set 2 |
|---|---|---|
| 107 | | |
| 108 | | |
| 109 | | |
| 110 | | |
| 111 | | |
| 112 | | |
| 113 | | |
| 114 | | |
| 115 | | |
| 116 | | |
| 117 | | |
| 118 | | |
| 119 | | |
| 120 | | |
| 121 | | |
| 122 | | |
| 123 | | |
| 124 | | |
| 125 | | |
| 126 | | |
| 127 | | |

| Poke code | Set 1 | Set 2 |
|---|---|---|
| 235 | | |
| 236 | | |
| 237 | | |
| 238 | | |
| 239 | | |
| 240 | | |
| 241 | | |
| 242 | | |
| 243 | | |
| 244 | | |
| 245 | | |
| 246 | | |
| 247 | | |
| 248 | | |
| 249 | | |
| 250 | | |
| 251 | | |
| 252 | | |
| 253 | | |
| 254 | | |
| 255 | | |

## Appendix E

# Display Icons

# COLOR ICONS

| Icon | Key(s) to press | What it does | Icon | Key(s) to press | What it does |
|---|---|---|---|---|---|
| | CTRL-1 | Text color black | | C= – 1 | Text color orange |
| | CTRL-2 | Text color white | | C= – 2 | Text color brown |
| | CTRL-3 | Text color red | | C= – 3 | Text color light red |
| | CTRL-4 | Text color cyan | | C= – 4 | Text color dark gray |
| | CTRL-5 | Text color purple | | C= – 5 | Text color medium gray |
| | CTRL-6 | Text color green | | C= – 6 | Text color light green |
| | CTRL-7 | Text color blue | | C= – 7 | Text color light blue |
| | CTRL-8 | Text color yellow | | C= – 8 | Text color light gray |

# OTHER ICONS

| Icon | Key(s) to press | What it does | Icon | Key(s) to press | What it does |
|---|---|---|---|---|---|
| | CLR/home | Cursor home | | Shift–CLR/home | Clear screen |
| | CRSR ↕ | Cursor down | | Shift–CRSR ↕ | Cursor up |
| | CRSR ↔ | Cursor right | | Shift–CRSR ↔ | Cursor left |
| | CTRL–9 | Reverse on | | CTRL-0 | Reverse off |

Appendix F

# Color Codes

| | |
|---|---|
| 0 - black | 8 - orange |
| 1 - white | 9 - brown |
| 2 - red | 10 - light red |
| 3 - cyan | 11 - dark gray |
| 4 - purple | 12 - medium gray |
| 5 - green | 13 - light green |
| 6 - blue | 14 - light blue |
| 7 - yellow | 15 - light gray |

## Appendix G

# Normal Sprite Coding Form

| Column Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | Number codes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Values | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | | | |
| Row 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 4 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 6 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 7 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 8 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 9 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 10 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 11 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 12 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 13 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 14 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 15 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 16 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 17 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 18 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 19 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 20 | | | | | | | | | | | | | | | | | | | | | | | | | | | |

## Appendix H

# Multicolor Sprite Coding Form

| Column number | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | | 9 | | 10 | | 11 | | Number codes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Values | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | | | |
| Row 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 4 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 6 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 7 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 8 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 9 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 10 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 11 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 12 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 13 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 14 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 15 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 16 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 17 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 18 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 19 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Row 20 | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Transparent screen color | | Multicolor register #0 | | Sprite color | | Multicolor register #1 | |
|---|---|---|---|---|---|---|---|
| | 0 0 | | 0 1 | | 1 0 | | 1 1 |

Appendix I

# Character Coding Form

| Bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Number codes |
|---|---|---|---|---|---|---|---|---|---|
| Bit value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
| Byte 0 | | | | | | | | | |
| Byte 1 | | | | | | | | | |
| Byte 2 | | | | | | | | | |
| Byte 3 | | | | | | | | | |
| Byte 4 | | | | | | | | | |
| Byte 5 | | | | | | | | | |
| Byte 6 | | | | | | | | | |
| Byte 7 | | | | | | | | | |

## Appendix J

# Multicolor Character Coding Form

| Bit value ▶ | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | Number codes ▼ |
|---|---|---|---|---|---|---|---|---|---|
| Byte 0 | | | | | | | | | |
| Byte 1 | | | | | | | | | |
| Byte 2 | | | | | | | | | |
| Byte 3 | | | | | | | | | |
| Byte 4 | | | | | | | | | |
| Byte 5 | | | | | | | | | |
| Byte 6 | | | | | | | | | |
| Byte 7 | | | | | | | | | |

| | | | |
|---|---|---|---|
| Background #0 color (screen color) | Background #1 color | Background #2 color | Lower 3 bits of color memory color |

Appendix K

# 2H × 3V Character Block Coding Form

| | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |

## Appendix L

# SID Register Layout

SID starting address is 54272 ($D400)

| Register number Decimal | Hex | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | This register controls: | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | $00 | FR7 | FR6 | FR5 | FR4 | FR3 | FR2 | FR1 | FR0 | Low byte of frequency | Voice 1 |
| 1 | $01 | FR15 | FR14 | FR13 | FR12 | FR11 | FR10 | FR9 | FR8 | High byte of frequency | |
| 2 | $02 | PW7 | PW6 | PW5 | PW4 | PW3 | PW2 | PW1 | PW0 | Low byte of pulse width | |
| 3 | $03 | – | – | – | – | PW11 | PW10 | PW9 | PW8 | High nibble of pulse width | |
| 4 | $04 | Noise | Pulse | Saw-tooth | Trian-gular | Test | Ring mod | Sync | Gate | Gate and wave-form control | |
| 5 | $05 | ATK3 | ATK2 | ATK1 | ATK0 | DCY3 | DCY2 | DCY1 | DCY0 | Attack/decay | |
| 6 | $06 | SST3 | SST2 | SST1 | SST0 | RLS3 | RLS2 | RLS1 | RLS0 | Sustain/release | |
| 7 | $07 | FR7 | FR6 | FR5 | FR4 | FR3 | FR2 | FR1 | FR0 | Low byte of frequency | Voice 2 |
| 8 | $08 | FR15 | FR14 | FR13 | FR12 | FR11 | FR10 | FR9 | FR8 | High byte of frequency | |
| 9 | $09 | PW7 | PW6 | PW5 | PW4 | PW3 | PW2 | PW1 | PW0 | Low byte of pulse width | |
| 10 | $0A | – | – | – | – | PW11 | PW10 | PW9 | PW8 | High nibble of pulse width | |
| 11 | $0B | Noise | Pulse | Saw-tooth | Trian-gular | Test | Ring mod | Sync | Gate | Gate and wave-form control | |
| 12 | $0C | ATK3 | ATK2 | ATK1 | ATK0 | DCY3 | DCY2 | DCY1 | DCY0 | Attack/decay | |
| 13 | $0D | SST3 | SST2 | SST1 | SST0 | RLS3 | RLS2 | RLS1 | RLS0 | Sustain/release | |

| Register number Decimal | Hex | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | This register controls: | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | $0E | FR7 | FR6 | FR5 | FR4 | FR3 | FR2 | FR1 | FR0 | Low byte of frequency | Voice 3 |
| 15 | $0F | FR15 | FR14 | FR13 | FR12 | FR11 | FR10 | FR9 | FR8 | High byte of frequency | Voice 3 |
| 16 | $10 | PW7 | PW6 | PW5 | PW4 | PW3 | PW2 | PW1 | PW0 | Low byte of pulse width | Voice 3 |
| 17 | $11 | – | – | – | – | PW11 | PW10 | PW9 | PW8 | High nibble of pulse width | Voice 3 |
| 18 | $12 | Noise | Pulse | Saw-tooth | Trian-gular | Test | Ring mod | Sync | Gate | Gate and waveform control | Voice 3 |
| 19 | $13 | ATK3 | ATK2 | ATK1 | ATK0 | DCY3 | DCY2 | DCY1 | DCY0 | Attack/decay | Voice 3 |
| 20 | $14 | SST3 | SST2 | SST1 | SST0 | RLS3 | RLS2 | RSL1 | RLS0 | Sustain/release | Voice 3 |
| 21 | $15 | – | – | – | – | – | CFR2 | CFR1 | CFR0 | Low 3 bits of cutoff/center frequency | Filter/volume |
| 22 | $16 | CFR10 | CFR9 | CFR8 | CFR7 | CFR6 | CFR5 | CFR4 | CFR3 | High 8 bits of cutoff/center frequency | Filter/volume |
| 23 | $17 | RES3 | RES2 | RES1 | RES0 | Filter external | Filter V3 | Filter V2 | Filter V1 | Resonance/filter | Filter/volume |
| 24 | $18 | V3 silent | High pass | Band pass | Low pass | Volume 3 | Volume 2 | Volume 1 | Volume 0 | Filter mode/volume | Filter/volume |
| 25 | $19 | GPX 7 | GPX 6 | GPX 5 | GPX 4 | GPX 3 | GPX 2 | GPX 1 | GPX 0 | Game paddle X | Other |
| 26 | $20 | GPY 7 | GPY 6 | GPY 5 | GPY 4 | GPY 3 | GPY 2 | GPY 1 | GPY 0 | Game paddle Y | Other |
| 27 | $21 | V30 7 | V30 6 | V30 5 | V30 4 | V30 3 | V30 2 | V30 1 | V30 0 | Voice 3 oscillator | Other |
| 28 | $22 | V3E 7 | V3E 6 | V3E 5 | V3E 4 | V3E 3 | V3E 2 | V3E 1 | V3E 0 | Voice 3 envelope | Other |

## Appendix M

# Note Values

| Octave | Note name | Frequency in hertz | SID freq. setting | High byte of SID freq. set. | Low byte of SID freq. set. |
|---|---|---|---|---|---|
| 0 | C | 16.4 | 269 | 1 | 13 |
| 0 | C# | 17.3 | 284 | 1 | 28 |
| 0 | D | 18.4 | 302 | 1 | 46 |
| 0 | D# | 19.4 | 318 | 1 | 62 |
| 0 | E | 20.6 | 338 | 1 | 82 |
| 0 | F | 21.8 | 358 | 1 | 102 |
| 0 | F# | 23.1 | 379 | 1 | 123 |
| 0 | G | 24.5 | 402 | 1 | 146 |
| 0 | G# | 26.0 | 427 | 1 | 171 |
| 0 | A | 27.5 | 451 | 1 | 195 |
| 0 | A# | 29.1 | 477 | 1 | 221 |
| 0 | B | 30.9 | 507 | 1 | 251 |
| 1 | C | 32.7 | 536 | 2 | 24 |
| 1 | C# | 34.6 | 568 | 2 | 56 |
| 1 | D | 36.7 | 602 | 2 | 90 |
| 1 | D# | 38.9 | 638 | 2 | 126 |
| 1 | E | 41.2 | 676 | 2 | 164 |
| 1 | F | 43.7 | 717 | 2 | 205 |
| 1 | F# | 46.2 | 758 | 2 | 246 |
| 1 | G | 49.0 | 804 | 3 | 36 |
| 1 | G# | 51.9 | 851 | 3 | 83 |
| 1 | A | 55.0 | 902 | 3 | 134 |
| 1 | A# | 58.3 | 956 | 3 | 188 |
| 1 | B | 61.7 | 1012 | 3 | 244 |
| 2 | C | 65.4 | 1073 | 4 | 49 |
| 2 | C# | 69.3 | 1137 | 4 | 113 |
| 2 | D | 73.4 | 1204 | 4 | 180 |
| 2 | D# | 77.8 | 1276 | 4 | 252 |
| 2 | E | 82.4 | 1352 | 5 | 72 |
| 2 | F | 87.3 | 1432 | 5 | 152 |
| 2 | F# | 92.5 | 1517 | 5 | 237 |
| 2 | G | 98.0 | 1608 | 6 | 72 |
| 2 | G# | 103.8 | 1703 | 6 | 167 |
| 2 | A | 110.0 | 1804 | 7 | 12 |
| 2 | A# | 116.5 | 1911 | 7 | 119 |
| 2 | B | 123.5 | 2026 | 7 | 234 |
| 3 | C | 130.8 | 2146 | 8 | 98 |
| 3 | C# | 138.6 | 2274 | 8 | 226 |
| 3 | D | 146.8 | 2408 | 9 | 104 |
| 3 | D# | 155.6 | 2553 | 9 | 249 |
| 3 | E | 164.8 | 2703 | 10 | 143 |
| 3 | F | 174.6 | 2864 | 11 | 48 |
| 3 | F# | 185.0 | 3035 | 11 | 219 |
| 3 | G | 196.0 | 3215 | 12 | 143 |
| 3 | G# | 207.7 | 3407 | 13 | 79 |
| 3 | A | 220.0 | 3609 | 14 | 25 |
| 3 | A# | 233.1 | 3824 | 14 | 240 |
| 3 | B | 246.9 | 4050 | 15 | 210 |

Note Values

| Octave | Note name | Frequency in hertz | SID freq. setting | High byte of SID freq. set. | Low byte of SID freq. set. |
|---|---|---|---|---|---|
| 4 | C | 261.6 | 4291 | 16 | 195 |
| 4 | C# | 277.2 | 4547 | 17 | 195 |
| 4 | D | 293.7 | 4818 | 18 | 210 |
| 4 | D# | 311.1 | 5103 | 19 | 239 |
| 4 | E | 329.6 | 5407 | 21 | 31 |
| 4 | F | 349.2 | 5728 | 22 | 96 |
| 4 | F# | 370.0 | 6070 | 23 | 182 |
| 4 | G | 392.0 | 6431 | 25 | 31 |
| 4 | G# | 415.3 | 6813 | 26 | 157 |
| 4 | A | 440.0 | 7218 | 28 | 50 |
| 4 | A# | 466.2 | 7648 | 29 | 224 |
| 4 | B | 493.9 | 8102 | 31 | 166 |
| 5 | C | 523.3 | 8584 | 33 | 136 |
| 5 | C# | 554.4 | 9095 | 35 | 135 |
| 5 | D | 587.3 | 9634 | 37 | 162 |
| 5 | D# | 622.3 | 10208 | 39 | 224 |
| 5 | E | 659.3 | 10815 | 42 | 63 |
| 5 | F | 698.5 | 11458 | 44 | 194 |
| 5 | F# | 740.0 | 12139 | 47 | 107 |
| 5 | G | 784.0 | 12861 | 50 | 61 |
| 5 | G# | 830.6 | 13625 | 53 | 57 |
| 5 | A | 880.0 | 14436 | 56 | 100 |
| 5 | A# | 932.3 | 15294 | 59 | 190 |
| 5 | B | 987.8 | 16204 | 63 | 76 |
| 6 | C | 1046.5 | 17167 | 67 | 15 |
| 6 | C# | 1108.7 | 18188 | 71 | 12 |
| 6 | D | 1174.7 | 19270 | 75 | 70 |
| 6 | D# | 1244.5 | 20415 | 79 | 191 |
| 6 | E | 1318.5 | 21629 | 84 | 125 |
| 6 | F | 1396.9 | 22915 | 89 | 131 |
| 6 | F# | 1480.0 | 24278 | 94 | 214 |
| 6 | G | 1568.0 | 25722 | 100 | 122 |
| 6 | G# | 1661.2 | 27251 | 106 | 115 |
| 6 | A | 1760.0 | 28872 | 112 | 200 |
| 6 | A# | 1864.7 | 30589 | 119 | 125 |
| 6 | B | 1975.5 | 32407 | 126 | 151 |
| 7 | C | 2093.0 | 34334 | 134 | 30 |
| 7 | C# | 2217.5 | 36377 | 142 | 25 |
| 7 | D | 2349.3 | 38539 | 150 | 139 |
| 7 | D# | 2489.0 | 40831 | 159 | 127 |
| 7 | E | 2637.0 | 43258 | 168 | 250 |
| 7 | F | 2793.8 | 45831 | 179 | 7 |
| 7 | F# | 2960.0 | 48557 | 189 | 173 |
| 7 | G | 3136.0 | 51444 | 200 | 244 |
| 7 | G# | 3322.4 | 54502 | 212 | 230 |
| 7 | A | 3520.0 | 57743 | 225 | 143 |
| 7 | A# | 3729.3 | 61177 | 238 | 249 |
| 7 | B | 3951.1 | 64815 | 253 | 47 |

# Appendix N
# ANDing and ORing

ANDing and ORing are logical operations your Commodore 64 uses to play with bits and check on the truth of complex expressions. I'll try to give you a brief glimpse of how they work.

First, a few conventions:

—When the computer tries to decide whether a number is true or false, any nonzero number is considered true.

—When the computer looks over a comparison, and decides that the comparison is true, it assigns it the value −1. A false comparison is assigned the value 0.

Here's a brief program that illustrates these two conventions at work:

```
10  IF 8 THEN PRINT "8 IS TRUE"
20  IF 0 THEN PRINT "0 IS TRUE":
    GOTO 40
30  PRINT "0 IS FALSE"
40  PRINT (9 = 8)
50  PRINT (9 = 9)
```

Running the program will give these results:

```
8  IS TRUE
0  IS FALSE
0
-1
```

The Commodore 64 performs ANDing and ORing on numbers in the range −32768 to +32767. The numbers first have any fractional parts dropped, and then they're converted into 16-bit binary format. Here are some examples:

| ORIGINAL VALUE | FRACTION DROPPED | 16-BIT BINARY |
|---|---|---|
| −1 | −1 | 1111 1111 1111 1111 |
| 254.75 | 254 | 0000 0000 1111 1110 |
| 513 | 513 | 0000 0010 0000 0001 |
| 0 | 0 | 0000 0000 0000 0000 |
| 15.4 | 15 | 0000 0000 0000 1111 |

Note that I have inserted spaces into the 16-bit binary values just to make them easier for humans to read.

When two numbers are ANDed together, they're first put into this chopped-off 16-bit binary format. Then corresponding bits are ANDed together according to the following arbitrary rules:

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| AND 0 | AND 1 | AND 0 | AND 1 |
| 0 | 0 | 0 | 1 |

The result is then converted back to decimal form. Here are some examples of ANDing:

| | | | | | |
|---|---|---|---|---|---|
| | | | | −1 | decimal |
| | | | AND | 0 | decimal |
| | 1111 | 1111 | 1111 | 1111 | binary |
| AND | 0000 | 0000 | 0000 | 0000 | binary |
| | 0000 | 0000 | 0000 | 0000 | binary |
| | | | | 0 | decimal |

| | | | | | |
|---|---|---|---|---|---|
| | | | | 255 | decimal |
| | | | AND | 15 | decimal |
| | 0000 | 0000 | 1111 | 1111 | binary |
| AND | 0000 | 0000 | 0000 | 1111 | binary |
| | 0000 | 0000 | 0000 | 1111 | binary |
| | | | | 15 | decimal |

In graphics and sound programming on the Commodore 64, ANDing is often used to turn certain bits in a register off. For example, if you wanted to turn off bits 4, 5, 6, and 7 in a register, you'd AND the register value with the number 15. Take a look at the last example to see why this is so.

When two numbers are ORed together, they're first put into the familiar chopped-off 16-bit binary format. Then corresponding bits are ORed together according to the following arbitrary rules:
(sound familiar?)

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| OR 0 | OR 1 | OR 0 | OR 1 |
| 0 | 1 | 1 | 1 |

The result is then converted back to decimal form. Here are some examples of ORing:

| | | | | | |
|---|---|---|---|---|---|
| | | | | −1 | decimal |
| | | | OR | 0 | decimal |
| | 1111 | 1111 | 1111 | 1111 | binary |
| OR | 0000 | 0000 | 0000 | 0000 | binary |
| | 1111 | 1111 | 1111 | 1111 | binary |
| | | | | −1 | decimal |

| | | | | | |
|---|---|---|---|---|---|
| | | | | 537 | decimal |
| | | | OR | 131 | decimal |
| | 0000 | 0010 | 0001 | 1001 | binary |
| OR | 0000 | 0000 | 1000 | 0011 | binary |
| | 0000 | 0010 | 1001 | 1011 | binary |
| | | | | 67 | decimal |

In graphics and sound programming on the Commodore 64, ORing is often used to turn certain bits in a register on. For example, if you wanted to turn on bits 0, 1, and 7 in a register, you'd OR the register value with the number 131. Take a look at the last example to see why this is so.

So much for a brief look at ANDing and ORing. They're really quite remarkable functions. In fact, your Commodore 64 spends most of its time, at its deepest subconscious levels, ANDing and ORing away several million times each second.

# Index

# Index

# Commodore 64 Graphics and Sound Programming

If you are intrigued with the possibilities of the programs included in *Commodore 64 Graphics and Sound Programming.* (TAB Book No. 1640), you should definitely consider having the ready-to-run tape or disk containing the software applications. This software is guaranteed free of manufacturer's defects. (If you have any problems, return the tape or disk within 30 days, and we'll send you a new one.) Not only will you save the time and effort of typing the programs, the tape or disk eliminates the possibility of errors that can prevent the programs from functioning. Interested?

Available on tape and on disk for the Commodore 64 at $23.95 for each tape or disk plus $1.00 each shipping and handling.

I'm interested in ready-to-run software for *Commodore 64 Graphics and Sound Programming.* Send me:

_______ tape for the Commodore 64 (6416S)

_______ disk for the Commodore 64 (6417S)

_______ TAB BOOKS catalog

_______ Check/Money order enclosed for $23.95 plus $1.00 shipping and handling for each tape or disk ordered.

_______ VISA _______ MasterCard

Account No. _______________________ Expires _______

Name _______________________

Address _______________________

City _______________ State _______ Zip _______

Signature _______________________

Mail To **TAB BOOKS INC.**
**Blue Ridge Summit, PA 17214**

(Pa. add 6% sales tax. Orders outside U.S. must be prepaid with international money orders in U.S. dollars.)

TAB 1640

# Commodore 64 Graphics And Sound Programming

by Stan Krute

Here's a hands-on, learn-by-doing approach to mastering the full graphics potential offered by your Commodore 64 (sprite, character, and bit mapped graphics), *plus* how to take fullest advantage of the machine's remarkable built-in three-voice music synthesizer chip . . . using a collection of fascinating programs developed by the author.

Written in easy-to-follow, non-technical language, this exceptional handbook shows you how to use BASIC to get the kind of sound and graphic results on your C-64 that would require advanced assembly language skills on any other model computer. It shows you the hows and whys of operating your machine for maximum performance, *without* tangling you in a maze of computer jargon or overwhelming you with complex programming theory.

You'll find out how really simple it is to produce exciting images on your C-64, using only the powerful hardware packed into your machine . . . and find out how the 64's sound chip can produce an amazing range of sounds *and* musical effects. Once you've mastered these fundamentals—using the clear explanations, sample exercises, and ready-to-run programs—you'll be ready to start combining graphics *and* sound for some amazingly sophisticated results—techniques you can use to write your own original programs for business, household use, or game playing applications!

A total of 68 programs has been included, each one written in clear, simple BASIC. You'll find thorough descriptions of what the programs are designed to accomplish, complete program listings, and plenty of information on C-64 programming specifics: screen memory, color memory, screen display codes, display icons, color codes, normal and multicolor sprite coding forms, character and multicolor character coding forms, 2H × 3V character block coding, SID register layout, ANDing and ORing, and more. Plus there are plenty of figures, charts, and helpful advice to get you started designing your own sprites and custom characters.

Stan Krute is a writer, programmer, teacher, and artist. He lives in northern California.



**TAB BOOKS Inc.**

Blue Ridge Summit, Pa. 17214



FPT > $15.50

ISBN 0-8306-0640-8

PRICES HIGHER IN CANADA

1495-1083