INCLUDES VALUABLE C-64 SOFTWARE TOOLS!

COMMODORE MAGIC

Create your own astonishing graphics and sound effects!

A Hard/Soft Press Book

Michael Callery

COMMODORE MAGIC

MICHAEL CALLERY is vice-president of Learningware Corporation, a firm specializing in software for instruction and training. He teaches about computers and computer graphics at the New School for Social Research, New York University, and Manhattan College. Mr. Callery is the author of many reviews and articles that have appeared in *Computer Buying Guide*, *Computers & Programs*, *MicroKids*, and *SoftSide* magazines. He and his nine computers live happily in New York City.

COMMODORE MAGIC

Create Astonishing Graphics and Sound Effects for Your Commodore 64!

MICHAEL CALLERY

A HARD / SOFT PRESS BOOK

E.P. DUTTON, INC.   New York

This one's for Dad

The author would like to thank the people at Hard/Soft Press for their encouragement, prodding, and most of all, trust. Special acknowledgment to Terry Nasta for her inestimable contributions to the manuscript.

Diagrams by Shane Kelley

Technical assistance: Rick Hoffman

Copyright © 1984 by Hard/Soft Inc.

All rights reserved. Printed in the U.S.A.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system now known or to be invented, without permission in writing from the publisher, except by a reviewer who wishes to quote brief passages in connection with a review written for inclusion in a magazine, newspaper, or broadcast.

Published in the United States by E. P. Dutton, Inc.,
2 Park Avenue, New York, N.Y. 10016

Library of Congress Catalog Card Number: 84-70407

ISBN: 0-525-48120-6

Published simultaneously in Canada by Fitzhenry & Whiteside Limited, Toronto

10 9 8 7 6 5 4 3 2 1
COBE
First Edition

CONTENTS

1

Getting Started 1

The magic of Commodore graphics and sound. General program hints and tips, and a peek inside to see just how it works. A guide to the special Commodore keys.

2

A Basic Potpourri 7

Basic language highlights. The anatomy of a byte. Logic made easy. Snooping inside your memory. Painless number conversions. Generating computer poetry.

3

Tricks with Text 22

Mastering Commodore's text displays. How programmers create professional-looking screens. Flashy text, SHIFTy characters, and positioning FUNctionality.

4

An Excursion into Color 32

Controlling the Commodore's dazzling palette of colors. Memory surprises and a program to fine-tune your monitor. A tour of its powerful video display chip and what it can do.

5

An Introduction to Animation 40

Moving figures across the screen may look like magic, but here you'll find the simple secrets of sophisticated internal, one-frame, and multiframe text animation.

6

Quite a Character 47

How the Commodore produces and stores its characters, and how you can enhance and alter them with ease. Customizing tricks and shortcuts. Input/output magic and the phantom ROM.

7

The Character Set Editor 56

A step-by-step guide to constructing a program that will let you create whole new sets of characters—while your computer does all the dirty work. And, meet an elegant decoder.

8

There Are Sprites at the Bottom of My Garden 71

Text may be fun, but sprites are pure magic. How to whip them together, glide them across your screen, and manipulate them most effectively. Sublime axis expansions and 3-D effects.

9

Building the Sprite Editor 84

Constructing and keeping track of sprites can be tricky business. This will teach your Commodore how to handle all the details for you. And add many user-friendly features.

10

The Rainbow Revolution 96

Color, color, and more color. Characters and sprites can be far flashier than you've seen, when you discover multicolor and extended background modes. The importance of mastering masks.

11

Hi-Res Hijinks 105

With all 64,000 dots at your command you can create eye-popping displays. The joys of artifacting, aliasing, aspect correction, and multicolor hi-res. A handful of powerful programming tools.

12

Screen Maker 120

Your television set becomes an artist's canvas as you create a program to make hi-res drawing a snap. Using a joystick as a paintbrush. Mixing hi-res graphics and sprites.

13

Odds and Ends 138

Basic can't do everything by itself. A glimpse into assembler to let you harness the full power of your Commodore 64. New assembler subroutines to enhance the power of your programs. Techniques for effortless saving and loading.

14

Introducing SID 151

If you're impressed by the Commodore graphics, wait until you hear its sound! Put together your own masterful sound editor to create startling multivoice music and sound effects. Music theory without tears.

15

The Multilingual Commodore 174

Your Commodore 64 can speak many languages. Here we'll explore the graphics and sound abilities in several of the most popular ones—and discover new power for Basic.

16

Software Tools 197

Professional programmers are creating programs to help you make the most of your Commodore 64. Here we'll put them under the microscope and examine their good—and not so good—points.

Appendix A: Programming Primer 211

Appendix B: Commodore Character Set 222

Appendix C: Selected References 229

Index 230

Getting Started

Did you ever watch a really expert magician perform an astonishing trick right before your eyes, and all you could think of was "How in the world did he do that?"

This book gives you a generous collection of graphics and sound tricks written expressly for the Commodore 64—and reveals the secrets behind them. Just follow the simple, step-by-step instructions, type in a few words of code, hit the RETURN key, and in seconds, your screen will be performing spectacular feats of computer magic.

But best of all, you'll learn techniques for inventing new and different ones of your own. That way you can transfer your own creativity to the screen.

Valuable Software Tools

Whether you're an old hand at computing or just starting out, *Commodore Magic* will show you how to get the maximum graphics and sound horsepower out of your equipment—with plenty of interesting examples, tips, and tricks along the way.

While most books limit you to the examples between their covers, we provide you with four sophisticated pieces of software—a powerful character editor, sound generator, hi-res graphics editor, and sprite editor—that you can use to create impressive sound and graphics programs of your own design. What's more, you'll also get a programmer's tool kit of valuable mini-programs and routines that make customizing and writing your programs a breeze.

Commodore Magic makes mastering graphics and sound painless, simple, and loads of fun. It's like having a friend who's a computer expert sitting next to you, showing you eye-popping tricks and pointing out how everything works. And when you've finished typing in the examples, you'll have a reuseable collection of powerful software tools that rival those you can buy.

A Magical Sampler

In the following chapter we'll clear up some of the mysteries of Basic, which is actually very easy once it's been explained properly. With a good teacher, even young children can pick it right up. You may not know a PEEK from a POKE from a bowl of spaghetti before you start, but pretty soon you'll be tossing bytes and Boolean binary bits around with ease and assurance.

Ode to a Commodore

With this book as a guide, you can even teach your computer how to write poetry. Okay, so maybe it won't be Robert Frost, but I'll bet your computer never handed you a poem before. We'll also put together a program that converts our number system to the computer's, and vice versa, to save you hours of pencil work later on.

In Chapter 3, we'll cover tricks with text. You'll learn how to dance characters across your screen in dozens of inventive ways. Also, you'll find out how to set the artist in your Commodore 64 free and create exotic, random displays.

Chapter 4 takes you on an excursion into color. Your clever Commodore 64 can juggle 16 colors at once, which is many more than some expensive computers can. We'll conjure up a professional-looking color bar chart, and then give your text vibrant rainbow hues. Kaleidoscopes were never this much fun.

In the fifth chapter we'll start experimenting with what is probably the most impressive of the Commodore's bag of tricks—animation. We'll introduce you to our raster graphics swimmer, and have it do a few laps across your screen. You'll learn how to flick out flicker, and generate some shimmery internal movements.

Next, you'll become an expert at taking out the computer garbage. You'll need to, since you'll be replacing it with your own customized set of letters

and numbers. We'll give you a smashing short-cut to making your own graphics building blocks, by snooping deep inside one of your computer's chips. And in the following chapter, we'll build our first powerful software editor. With it, you'll be able to write upside-down or backwards—or even in Greek—as easily as you type your name.

Animation Magic

In Chapter 8 you'll get to know sprites, those magical little complex shapes that can dart around your screen. Sprites make graphics come alive. You'll use one to create a space ship and launch it once around the galaxy.

Then we'll turn the flying saucer into a Volkswagen. Now if that's not magic, I don't know what is. But there's even more: There are commands to transform that squat little bug of an automobile into a sleek racing car, and then into a fat yellow taxi—all with just one tiny instruction. And to learn about the illusion of depth, you'll drive two cars head-on into each other. Then, you can toss in a whirling animated helicopter for fun. Finally, you'll build the second sophisticated software editor, to hammer your sprites together and give them a life of their own.

In Chapter 11, you'll be entertained by the Commodore's hi-res hijinks. If you like control, this is the place for you. 64,000 dots on the screen will march to your drumbeat. You'll learn how to direct them all with ease, turning your screen into a detailed artist's canvas with a 16-color combination palette.

Professional Secrets

Here you'll get a glimpse of your first assembly-language program (it's far simpler than you think), use a quirk in your television set to produce unusual colored effects, and straighten out some crazy circles. Along the way you can stop to play with a nested globe, and learn how to combat a case of the jaggies. You'll also find a handful of professional graphics programmers' favorite secrets, and plot some delicate swooping waves.

In Chapter 12, you take a break, and make your computer start doing some of the work. By this time you'll be an expert in creating exquisite geometric art, and we'll do some "human engineering" to make graphics a breeze. Plug in your joystick or tap out a few easy commands on your keys, and paint or draw merrily away with your third powerful editor—a multicolor high-resolution joystick-driven screen maker.

Once you begin using the powerful editors you've assembled in earlier chapters, you'll probably want to save the many masterpieces you've created. This chapter will show you how to take a "snapshot" of your Commodore's memory and save the images hidden inside.

Master programmers use assembly language to harness the real power of their computers. Many people mistakenly feel that assembly language is hard to learn. Wrong. It's no more difficult to learn than any computer language, it's just different. But it has one big advantage over other computer languages: blazing, breakneck, raw speed. Chapter 13 gives you several useful routines that are written in assembler. But don't panic. You'll see that these routines are easy to enter, and they'll perform their acceleration miracles for you whether or not you decide to study the chapter.

The Golden-Throated SID

Then, finally, meet SID—which just happens to be the very best music chip in the business. It's a fact—your Commodore has colors to spare, eight separate powerful sprites with eight levels of 3-D priority, and three simultaneous voices. That's miles above most other computers on the market.

SID is a full-fledged magic act all by himself. And a professional-quality sound effects synthesizer. He has an eight-octave range, and can produce four distinct classes of sound. He can warble away in three voices simultaneously, and has enough filters and other electronic wizardry to harmonize with himself.

You'll take a musical tour of your Commodore, showing SID at his very best. And we'll teach you all the complex sound and music tricks without tears. You get a piano keyboard of your very own, complete with dancing notes—and when you're done typing in the examples in this chapter, you'll be the proud owner of a powerful sound editor to compose tunes, whip together sound effects, and keep the neighbors awake.

In the final two chapters, we'll tell you all about the better commercial products designed for the Commodore 64, from programming aids to new languages to hardware input devices.

While the Commodore has awesome abilities, getting at some of its power is not always so easy. We'll review the Basic "extensions" on the market that can eliminate hours of drudgework in front of the screen, and tell you

how to get the most out of them. Some of these packages also let you perform tricks that you really just can't do in the version of Basic supplied by Commodore. We'll give you plenty of examples, complete with programs to illustrate our point.

We'll introduce you to languages such as Logo and Pilot, and highlight the advantages and the drawbacks of each. Then we'll run through some of the command structure, to get you started if you've already purchased one or both. And although you'll be able to do quite nicely, thank you, by using the sophisticated graphics and sound editors supplied inside this book, we'll tell you about some of the best, far more expensive ones you can buy at your local computer store.

How to Use This Book

The programs in this book are sequential; if you try to skip ahead, you may become lost. I've cross-referenced program segments to give you some direction. Almost all the programs are intended to be entered as you come across them. Always type them in exactly as they appear.

Sometimes, this may mean editing lines that you typed in earlier, as new functions are added. I've done this intentionally so that the program development process would be clear. In this way, I hope you can take the programs in this book and really make them yours—customize and modify them. If you feel you have no use for a character editor, enter it anyway—you'll pick up programming skills along the way. Besides, you never know when you might need a special character that the character editor can whip together for you.

If you are already a programmer, just turn the page and jump right in. But if you would like a quick refresher course in programming fundamentals, or if you're just learning the ropes, take a few minutes to read the programming primer in Appendix A.

This Appendix also contains a wealth of useful information on topics such as handling, editing, and merging programs. You'll also get a sprinkling of programming theory, and a quick dose of hardware basics.

If you can't wait, I understand. Just switch your computer on, sneak past these helpful hints, turn the page, and start enjoying the magic.

Commodore Special Keys

The special keys on the Commodore keyboard present a typographic nightmare when preparing a book such as this. What appears in the output of the Commodore printer is quite different from the actual keys you type, although the printout does look exactly like the screen. The inverse heart created by the <SHIFT CLR/HOME> key is a prime example of this. In this book I've decided to use angle brackets to indicate the special keys. You should not type the brackets, only the keys within them. For example:

Listing/text	Keypress
<CLR/HOME>	CLR/HOME key
<SHIFT CLR/HOME>	SHIFT key + CLR/HOME key
<SHIFT F1>	SHIFT key + F1 function key
<CTRL 9>	CTRL key + 9 key
<C=64 1>	Commodore Logo key + 1 key

The cursor keys are a special case:

<CURSOR UP>	SHIFT key + left CRSR key
<CURSOR DOWN>	left CRSR key
<CURSOR RIGHT>	SHIFT key + right CRSR key
<CURSOR LEFT>	right CRSR key

Where multiple keystrokes (including spaces) are called for, I've used the number of repeats inside the angle brackets:

<9 SPACE>	SPACE bar nine times
<4 CURSOR LEFT>	right CRSR key four times

It's easy to tell the difference between the multiple keystrokes and the combination keys—only the SHIFT, CTRL, or Commodore 64 key can be first in a combination keystroke. If a number is first, it's a multiplier.

A Basic Potpourri

PEEK and POKE

Basic is probably the world's most "spoken" language. It's called a general-purpose computer language because it can be used to accomplish nearly anything—from writing a game to writing an accounting system. Unfortunately, computer manufacturers have each developed their own version of Basic.

Commodore Basic 2.0 (the Basic in the Commodore 64) was written for early PET computers. Since these early computers did not have sprites, or sound, or high resolution graphics, Basic 2.0 does not have any commands dealing with them. Instead we'll have to rely on the old standbys **PEEK** and **POKE.** Beginning programmers don't need to use these instructions very much; there's enough to learn with plain Basic.

PEEK and POKE are companion instructions. If you enter PRINT PEEK (1024), for example, Basic finds the number that is stored in location 1024 and prints it; entering POKE 1024,44 causes Basic to store 44 in location 1024. There are about 64,000 possible locations to PEEK or POKE into in the Commodore 64, but most of these will not accomplish anything dramatic. It is possible, however, to destroy the program you are working on by using these instructions incorrectly. A group of special memory locations will enable you to control the advanced features of your computer. Try these:

```
PRINT PEEK (1024)          <RETURN>
POKE 1024,81               <RETURN>
PRINT PEEK (1024)          <RETURN>
```

(Note: from here on I'll assume you know to press the RETURN key after each line of instructions or after each program line.)

The numbers printed by PRINT PEEK will always be between 0 and 255 and this is also the range that is valid for POKEs. If you worked your way through the Commodore 64 Users Guide you've already got a few PEEKs and POKEs under your belt. Just be careful when you are entering a POKE because if you're off by a single digit, you could wipe out all your work!

64K is actually 65,536 **bytes** of memory (see Box 1) and, as in all computers, many of these bytes are dedicated to very specific functions. Since each byte in this 64K range is always in the same place with regard to all the other bytes, each byte may be termed a "memory location." For example, byte

BOX 1.
When Is a Kilo 1,024?

Computer people have always used metric terminology to describe the capacities of their computers or mass storage peripherals like disk drives. A kilo is supposed to be 1,000 but since most computers are digital, operating on 1s and 0s, two possible data values, terminology is expressed in powers of two. 1,024 happens to be two to the tenth power. Try this:

```
FOR I = 1 TO 16:PRINT I,I^2:NEXT I
```

and you'll see all of the common numbers used with microcomputer terminology.

number 1024 (counting up from 0) is the first byte to be displayed on the normal screen (see Box 2). By POKEing into the correct control locations with the right numbers, you can change this so that, for example, location 2048 becomes the first byte displayed on the screen.

BOX 2.
The Commodore 64 Text Screen

The screen of your Commodore 64 is actually a group of 1,000 memory locations. These, like all other locations, are accessible by PEEKs and POKEs. Try this:

```
POKE 53281,1:PRINT CHR$(147);:POKE 53281,6
FOR I = 1024 TO 2023:POKE I,65:NEXT I
```

Press <SHIFT CLR/HOME> before doing anything else.

Commodore was telling a little fib when it named this computer—your Commodore 64 actually has 84K! The extra 20K is Read Only Memory **(ROM)** and contains permanent features such as Basic. Since the 6510 microprocessor, the brain of your Commodore 64, can only locate 64K, you'll never be able to use all 84K at one time.

Instead, programs may use only 64K at one time but can quickly switch part of the memory on or off so that it appears that more memory is available. Furthermore, if you are using Basic or any other feature in ROM, you'll lose the ability to use those memory locations occupied by the ROM. That's why the Commodore 64 gleefully informs you when you turn it on that you have 38911 Basic bytes free. The rest of the 64K memory is being used for other functions and is not free for your use.

A Diversion: The Memory Map

A listing of the important memory locations and their functions is termed a memory map. For most intermediate or advanced applications, you need to become very familiar with this territory. Let's start at the top and work down (see Fig. 2.1).

The heart of your Commodore 64 lies at the top of the memory map. Commodore calls it the **kernal.** The kernal is permanently etched in ROM and is 8K long. It determines how the Commodore behaves at a very low level. For example, when you are typing a Basic program the kernal is actually doing most of the work with the keyboard and the screen. Basic just passes things to the kernal and only really gets busy when you type RUN.

Commodore has released at least three versions of the kernal since the Commodore 64 was introduced. For most functions, you won't notice a difference among the versions (see Box 3).

BOX 3.
Which Kernal Do You Have?

Turn your computer on and off then try this:

```
FOR I = 1024 TO 2023:POKE I,65:NEXT I
```

(Notice this is the same as line 2 in Box 2)

If you see white spades, you have revision 1 of the kernal; if you see nothing, you've revision 2; and if you see light blue spades, you've revision 3 or higher. See Chapter 4 for a further explanation of this.

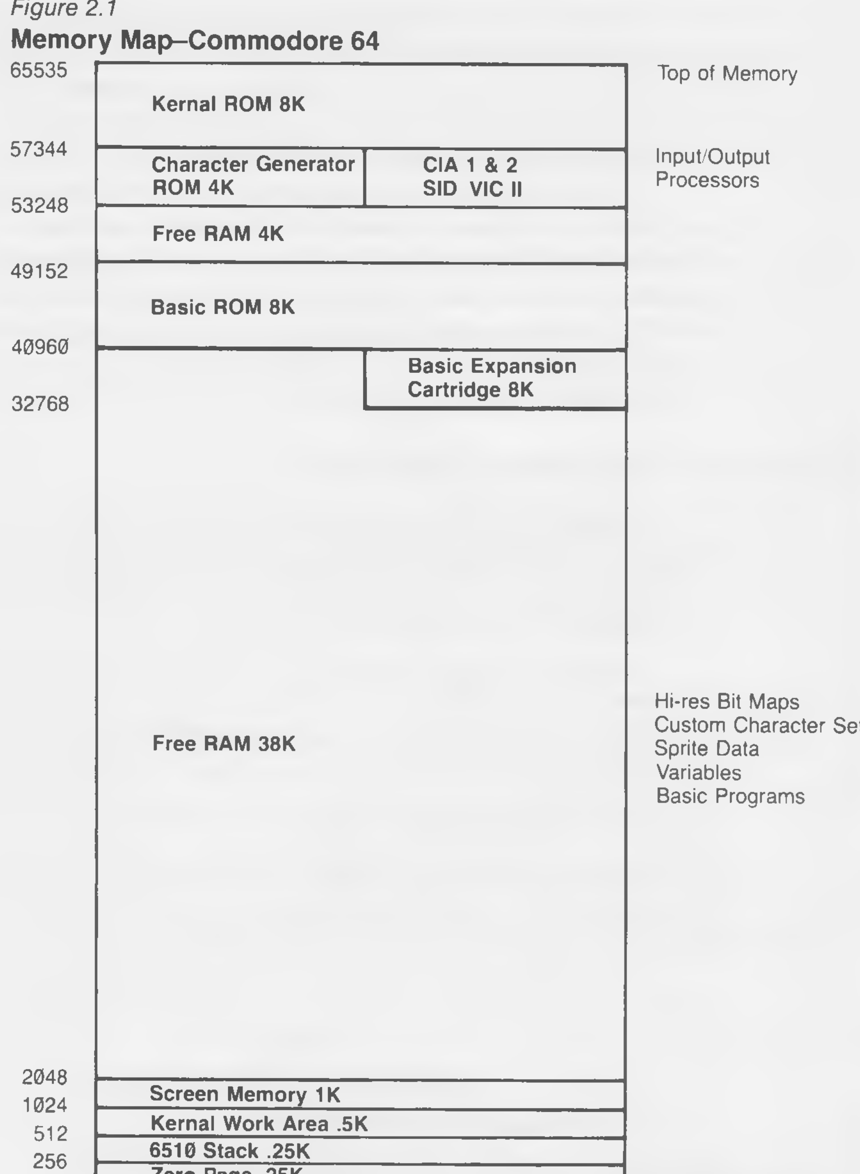


Figure 2.1
Memory Map–Commodore 64

If you have the revision 2 kernal, some commercial software may not run correctly. However, Commodore has informed software developers of the problem, and reputable vendors should have fixed their programs by now. In your own programs, you should include line 1 from Box 2 prior to any

graphics commands. In this way your program will work no matter what kernal is in your computer.

For the most part, we'll ignore the kernal since Basic works well on its own (with the kernal's help).

The character generator ROM is found next in the memory map, directly underneath the kernal, and it occupies 4K of memory. This character generator contains the dots that make up each character the Commodore 64 can display. We'll be doing a lot of work with the character generator later, in Chapter 6.

The very same addresses are also used for a second ROM containing input/output routines **(I/O)**—short programs that drive the use of a printer or disk. The Commodore 64 can switch between these two 4K sections (or banks). When the kernal needs to find the dots that make up, say, the letter "a," it switches on the character generator; at all other times the Commodore 64 has the I/O routines in mind.

So will we, for this area contains the video interface chip **(VIC II),** which does all of the screen work and the sound interface device **(SID),** which does all the music work. It also contains two complex interface adaptors **(CIA),** to handle other input and output joysticks, keyboard, light pens, and so on for the Commodore 64. While this may sound complicated, it's really much simpler than on many other computers. These four special circuits, VIC II, SID, and the two CIAs, do almost all the work, you need only give them instructions by PEEKing or POKEing the correct locations.

Next, there's a 4K bank of **RAM,** memory available for your use. This memory is very useful because it is isolated from the rest of RAM memory and is therefore a safe place to hide graphics information.

We're now 16K into the Commodore 64, and here we find Basic. Basic is stored in an 8K ROM. Commodore chose to use the same Basic as it included in the VIC and earlier PET computers. This Basic is fine except for its lack of commands to govern the sophisticated graphics and sound that the Commodore 64 produces.

The lower 40K of memory is free, all RAM. But it's not as free as we might like. Part of this chunk of memory is used to hold the screen data and other graphics data like sprites or bit maps. We'll be dividing up this area to provide room for our graphics and sound data. It will also hold our Basic program.

The very bottom of memory, the first 256 bytes, is very special. This area is called zero page (a page is 256 bytes, and there are 256 pages in the Commodore 64) and is used by the 6510 microprocessor and the kernal for all sorts of things. The zero page is absolutely off limits to us, although

machine language programmers can save its contents when a program is running then restore it before returning to Basic or the kernal.

Within this very general memory map, there are lots of possible variations. In fact, it is possible to turn off the ROM completely and make the Commodore 64 into an wholly RAM-based machine. As interesting as this may sound, it is not a job for anyone but the most experienced programmer.

With the kernal gone, the Commodore 64 would have no instructions on how to talk—or listen—to the outside world. The computer would be useless. Programs not written in Basic can switch off the Basic ROM and gain an additional 8K of program or data space. Some programs enhance Basic, adding to it to leave a 32K workspace. We'll be talking about some of these later on, for they can make your job, as a Basic programmer, much easier.

That was a pretty long diversion from PEEK and POKE but I hope you get the idea. Your PEEKs and POKEs are not random events, and must be used as carefully as you use the other Basic instructions. We'll get further into the Commodore 64 memory map as it affects our programming projects.

AND and OR

PEEK and POKE are two of the most useful programming structures for intermediate programmers. They are frequently used in conjunction to allow graphics or music data to be read into the proper memory locations. But for many purposes, these alone may not be enough. Many of the control locations in the Commodore 64 require only a part of the byte—a **bit**—to be changed.

This would be easy to do if Basic supported a binary number format, but it doesn't. For example, address 56320, located in the first CIA ROM space, contains data from the keyboard, the paddles, and the joystick. To read only the joystick, we must find some way to get at only bits 0 to 3 of this location. Similarly, to change from the normal 25-line display to a 24-line display, we must find a way to change only bit 3 of location 53265 without changing the other bits. These tasks can be accomplished by using PEEK and POKE in conjunction with **AND** and **OR.**

However, before we can explore these, it's important to know the anatomy of a byte. As you already know, a byte may store any number from 0 to 255. Just as in our more familiar decimal counting system, counting in binary in bits has a very simple rule: when you run out of digits, shift over one. For example, in decimal we have no problem counting until we reach 9. At 9 we are out of unique digits so we move over one and start again at 10.

In binary, we run out of digits very fast . . . 0 for zero, 1 for 1 and we're out of digits! So we start again having moved over one place: 10 is 2 in binary and 11 is 3. Once again, no more digits, so 4 is 100, 5 is 101, 6 is 110 and 7 is 111 and we're out of digits. Eight then becomes 1000 and so on. Fig. 2.2 contains a summary of this method of counting.

Figure 2.2
Binary Representation

Binary	Decimal
0000 0000	0
0000 0001	1
0000 0010	2
0000 0100	4
0000 1000	8
0001 0000	16
0010 0000	32
0100 0000	64
1000 0000	128

As if things weren't already confusing, it is traditional to start counting bits from 0 rather than 1. Since all the Commodore documentation follows this convention, we'll do it too. Thus, bit number 3 is the 8s digit, bit number 5 is the 32s digit, and bit number 7 is the 128s digit.

It's a simple matter to convert from decimal to binary. First find the largest number from the decimal list in Fig. 2.2 that will divide into your number. Write down the binary representation of that number. Then subtract it from your number and repeat. When you're out of numbers, combine all the binary values. Here's an example:

```
Your number: 156

Largest divisor: 128 . . . . . . . . . . . . .  1000  0000

leaving 28
Largest divisor: 16  . . . . . . . . . . . . .  0001  0000

leaving 12
Largest divisor: 8   . . . . . . . . . . . . .  0000  1000

leaving 4
Largest divisor: 4   . . . . . . . . . . . . .  0000  0100

leaving 0
Done . . . now combine the bits:                1001  1100
```

Going the other way is just the reverse, for example:

```
  1     1     1    0    0    1    1    0
128 + 64 + 32 + 0 + 0 + 4 + 2 + 0 = 230
```

Such conversions are at the heart of most of the Commodore 64's graphics, so it is worthwhile to learn to do them. The Binary-Decimal Program that follows will enable you to accomplish these conversions quickly and painlessly. Figs. 2.3 and 2.4 show how the screen looks when the program is run.

```
1 REM BINARY-DECIMAL PROGRAM
5 TT$="":V=0:REM INITIALIZE VARIABLES
10 PRINT "<SHIFT CLR/HOME> *** BINARY - DECIMAL PROGRAM ***"
20 PRINT "<2 CRSR DOWN><7 CRSR RIGHT>WHICH DO YOU WANT:"
30 PRINT "<CRSR DOWN><8 CRSR RIGHT>1. BINARY TO DECIMAL"
40 PRINT "<CRSR DOWN><8 CRSR RIGHT>2. DECIMAL TO BINARY"
50 PRINT "<CRSR DOWN><7 CRSR RIGHT>ENTER A 1 OR 2";
60 GET A$:IF A$="" THEN 60
70 IF A$<>"1" AND A$<>"2" THEN 60
80 ON VAL(A$) GOSUB 100,300
90 GOTO 5
100 PRINT "<SHIFT CLR/HOME><4 CRSR DOWN>"
105 PRINT "<24 SPACES>^^^^^^^^"
110 INPUT "<CLR/HOME><3 CRSR DOWN>ENTER A BINARY NUMBER ";A$
120 IF LEN(A$)<>8 THEN PRINT "INVALID NUMBER":GOTO 110
130 J=1
140 PRINT:PRINT "  . . . B I T . .","VALUE <8 SPACES>":
    PRINT "  7 6 5 4 3 2 1 0"
150 FOR I=1 TO 0 STEP -1
160 IF MID$(A$,J,1)="1" THEN V=V+2^I:T=2^I:GOTO 180
170 T=0
180 GOSUB 500:PRINT MID$(A$,J,1);SPC(2+I*2);T
190 J=J+1
200 NEXT
210 PRINT:PRINT "  TOTAL ->",V
220 GOSUB 550
230 RETURN
300 PRINT "<SHIFT CLR/HOME><4 CRSR DOWN>"
305 PRINT "<25 SPACES>^^^"
310 INPUT "<CLR/HOME><3 CRSR DOWN>ENTER A DECIMAL NUMBER ";A$
320 IF VAL(A$)>255 OR VAL(A$)<0 THEN PRINT "INVALID NUMBER":GOTO
    310
330 J=1:A=VAL(A$)
340 PRINT:PRINT"  . . . B I T . .","VALUE<8 SPACES>":PRINT "  7 6
    5 4 3 2 1 0"
350 FOR I = 7 TO 0 STEP -1
360 IF A=>2^I THEN T=2^I:A=A-T:V=V+T:T$="1":GOTO 380
370 T=0:T$="0"
380 GOSUB 500:PRINT T$;SPC(2+I*2);T
390 J=J+1:TT$=TT$+T$
400 NEXT
```

```
41Ø PRINT:PRINT,TT$;" = ",V
42Ø GOSUB 55Ø
43Ø RETURN
5ØØ FOR K = 7 TO I STEP -1:PRINT "<2 CRSR RIGHT>";:NEXT:
    RETURN
55Ø PRINT:PRINT "PRESS <CTRL 9>RETURN<CTRL Ø> FOR MORE OR
    <CTRL 9>Q<CTRL Ø> TO QUIT";
56Ø GET A$:IF A$=CHR$(13) THEN RETURN
57Ø IF A$="Q" THEN END
58Ø GOTO 56Ø
```

Figure 2.3

Output of Binary—Decimal Conversion Program. Conversion of 172 to binary.

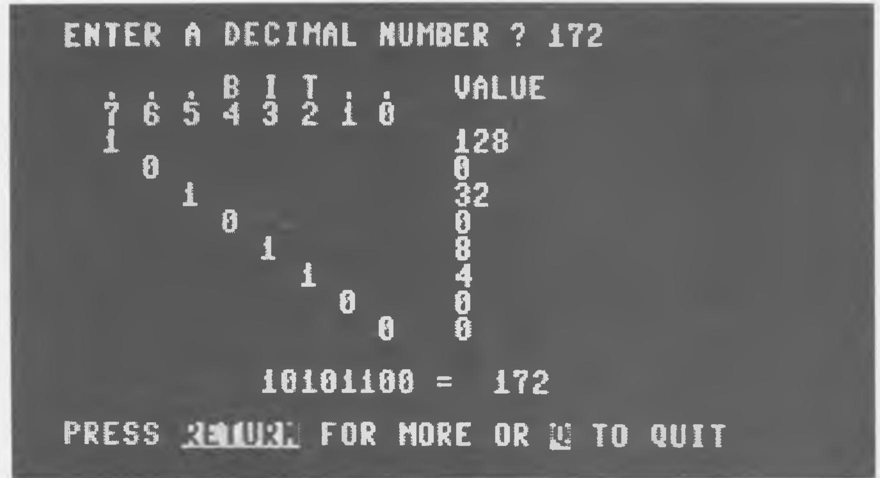
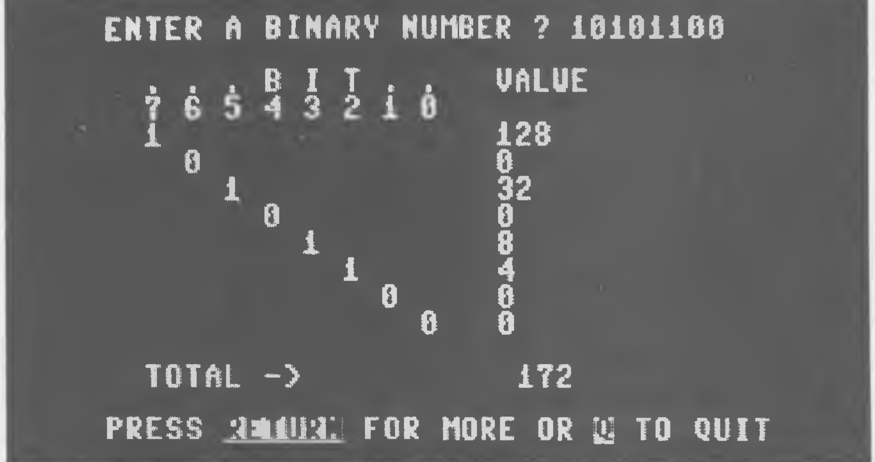


Figure 2.4

Output of Binary—Decimal Conversion Program. Conversion of 10101100 to decimal.



Boolean Values

Now that you have bits and bytes under your belt, we're ready to move on to AND and OR. AND and OR are termed **Boolean functions** because they compare two values and report the result of the comparison as either true or false (see Box 4).

> **BOX 4.**
> **Who Was Boole?**
>
> The term "Boolean" comes from George Boole, a 19th-century English mathematician who first proposed an algebra based on sets. Born of a poor family, he had little education beyond elementary school. He successfully educated himself and gained a full professorship at Queens College in Cork, Ireland, published more than 50 papers and books, and became a fellow of the Royal Society. However, he could never have predicted that his algebra would become one of the foundations of 20th-century computer science.

Since Basic cannot deal with true or false except as a **string** or literal, it uses −1 to represent true and a 0 to represent false. The results of ANDing and ORing are as follows:

1 AND 1	1 OR 1
True	True
1 AND 0	1 OR 0
False	True
0 AND 0	0 OR 0
False	False

Note that the middle comparisons, where the numbers being compared are different, yield different Boolean values for AND and OR. This is where Boolean functions become useful to us.

Back to our problem: how can we change only one bit of a byte? Since the Commodore 64 comes up in 25-line mode, the normal value of the control bit must tell the kernal to use 25 lines. It would be helpful, then, to see what the control byte is normally. Try this:

```
PRINT PEEK(53265)
```

You should get a 27. In binary, this is 0001 1011. Can you figure this out now? 27 = 16 + 8 + 2 + 1.

We want to end up with 0001 0011 since it's the third bit counting from 0 that controls the number of lines in the display. This turns out to be rather simple. Subtract 8 from whatever is already in location 53265. Try this:

```
POKE 53265, PEEK(53265)-8
```

You'll notice the top and bottom border suddenly get smaller. Going back is equally simple. Try this:

```
POKE 53265, PEEK(53265)+8
```

The screen will enlarge. These can equally well be accomplished by POKE 53265, PEEK(53265)AND 247 and POKE 53265, PEEK (53265) OR 8 respectively (247 is 255 − 8).

For a closer look at what we've done here, see Fig. 2.5.

Figure 2.5
Binary Representation of AND and OR Operations

Original Byte		Ø Ø Ø 1 1 Ø 1 1	(27)
	AND	1 1 1 1 Ø 1 1 1	(247)
Result		Ø Ø Ø 1 Ø Ø 1 1 (Note only the 8 bit changed)	(19)
Altered Byte		Ø Ø Ø 1 Ø Ø 1 1	(19)
	OR	Ø Ø Ø Ø 1 Ø Ø Ø	(8)
Result		Ø Ø Ø 1 1 Ø 1 1 (Note only the 8 bit changed)	(27)

Why do this rather than the simple additions and subtractions? For one, these are faster, although speed of this type of operation is rarely a problem. Second, the Boolean operations are easier when one has to do more complex actions than merely setting one bit. Since you will already be at the binary level when you decide which bit to set, it is appropriate to stay there. This will become apparent as we design our own screen characters and sprites.

Finally, and most important, Commodore makes extensive use of Boolean functions in its books and demonstration programs. Learning how they work will make it simpler to cross-reference what happens in this book with what happens in your Commodore manuals or demonstration programs.

READ/DATA: Pointers and Flags

Since we're going to be PEEKing and POKEing a lot of numbers, we will also make extensive use of **READ/DATA.** These Basic instructions are often misunderstood so let's look at them closely. Can you explain the following?

```
10 FOR I = 1 TO 5:READ A$:NEXT I
20 PRINT A$
30 READ B$
40 PRINT B$
50 READ A$,B$
60 PRINT A$,B$
100 DATA APPLES,ORANGES,ZEBRAS,ELEPHANTS,BEATLES,STONES
```

Basic maintains a **pointer** into the data. Pointers are very important. You can think of them as indexes into the data; they point to a data item. Each time Basic encounters a READ instruction, it increments the index or pointer by 1 and gets a data item.

In the program above, line 10 causes Basic to read the first five data items. Each, in turn, is assigned to A$. The pointer is now pointing to the fifth data item. If this seems difficult to grasp, pretend you are Basic and point your finger at each item in the data list as it reads them. Line 20 prints only the most recent value of A$ (BEATLES). The next READ, then, causes Basic to read the sixth item and assign it to B$. Line 40 prints this value (STONES).

When the next READ is encountered, in line 50, Basic has a problem. Even though A$ and B$ have a value now (BEATLES and STONES respectively) we are asking Basic to find new values in the data. But there are only six data items, so when the pointer is moved to seven, Basic finds no more data and reports an error to you.

This can be fixed by entering 45 RESTORE into the program. RESTORE resets the pointer to 0 so that the next READ will come from the head of the data list. But here's the rub. If you have many data statements in a program you have no way to restore to a specific line number or data section; you can only restore the whole thing and start at the head of the list. There is a way to deal with this. The program "POETRY," shown below, demonstrates how this can be done.

"POETRY" was originally written as part of an article for a magazine whose readers own many brands of computers. The article illustrated different features of Basic using simple programs. Computer-generated poetry is an interesting way to illustrate strings. The quality of the poetry produced by this program is doubtful, but it does work.

The simplest way to write such a program is to use a string **array** N$( ) to represent the nouns, read all the nouns into this array, and then pick a random noun from the array. However, all computers do not have string arrays so this approach couldn't be used in the article. Since there is usually more than one solution to a programming problem, I searched for a way to make the program work.

The solution turns out to be simple. To get to the nouns, I need to read all the articles and then a random number of nouns. Verbs are handled similarly. I read all the articles and nouns and then a random number of verbs. Since restore resets the pointer to the start of the data items, it is no problem to start at the beginning. The problem is how to know whether I need to read all the articles and nouns or just all the articles. This is accomplished by using a **flag.**

Flags are data structures that have two values: on or off. In this program I am using 1s and 0s to represent this. First, all flags are set to 0. After each part of the sentence is read, its flag is set to 1. Given this, line 310 does all the work so that the variable W takes on the value of the previously used parts of speech, while X becomes the random number used to read into the data list for the desired part of speech.

```
5 REM PROGRAM POETRY
10 NA = 3 :REM NUMBER OF ARTICLES
20 NN = 10:REM NUMBER OF NOUNS
30 NV = 10:REM NUMBER OF VERBS
40 NP = 10:REM NUMBER OF PHRASES
50 NL = 3 :REM NUMBER OF LINES IN A STANZA
60 NS = 3 :REM NUMBER OF STANZAS
100 FOR S = 1 to NS
110 FOR L = 1 TO NL
120 A = 0: N = 0: V = 0:REM INITIALIZE FLAGS
130 NW = NA:GOSUB 300
140 PRINT W$;" ";
150 A = 1:NW = NN:GOSUB 300
160 PRINT W$;" ";
170 N = 1:NW = NV:GOSUB 300
180 PRINT W$;" ";
190 V = 1:NW = NP:GOSUB 300
200 PRINT W$;"."
210 NEXT L
220 NEXT S
300 X = INT (RND(1) * NW) + 1
310 W = NA * A + NN * N + NV * V + NP * P
320 FOR WD = 1 TO W+X
330 READ W$
340 NEXT WD
350 RESTORE
360 RETURN
```

```
399 REM ARTICLES
400 DATA THE, ,A
409 REM NOUNS
410 DATA HEART,FLOWER,LOVE,FRIEND,
    DREAM,OCEAN,TREE,WAVE,DOVE,HAIR
419 REM VERBS
420 DATA LOVES,BUILT,ROARED,TOOK
    FLIGHT,GREW,SAW,BURST,RODE,WAS,SHOOK
429 REM PHRASES
430 DATA TO THE END,ON THE WING,WITHOUT A CARE
440 DATA BEYOND THE BLUE HORIZON,NEVERMORE,ON THE BEACH
450 DATA FOR A LARK,BEFORE THE DAWN,IN THE HEART,WITH LOVE
```

IF/THEN: Booleans in Disguise

By using a flag that has a value of 0 or −1, we can take advantage of the way IF statements work.

Remember Boolean values? These values are what are used in the condition that sits between the **IF** and the **THEN** of an IF/THEN statement. Try out the following program for size:

```
10 F=0
20 GOSUB 100
30 F=-1
40 GOSUB 100
50 F=2
60 GOSUB 100
90 END
100 IF F THEN PRINT "HELLO"
110 IF NOT F THEN PRINT "GOODBYE"
120 RETURN
```

This may seem strange . . . four words for three **GOSUB**s. Clearly something is happening here. Two small edits will clarify the issue. Change line 100 to read:

```
100 IF F THEN PRINT "HELLO";
```

and add this line:

```
115 PRINT
```

Now rerun it. Better? We've still got four words, but you can see that when F=0, it prints GOODBYE; when F=−1, it prints HELLO; and when F=2, it prints both HELLO and GOODBYE (stuck to each other because of the semicolon in line 100.) You find the same duplicated behavior whenever the value of F is anything other than 0 or −1.

Remember that 0 and −1 are Basic's way to code false and true respectively, but that still doesn't fully explain this program. When F is −1 (or true), it prints HELLO, just as you would expect. But when F is 0, it prints GOODBYE but Basic only executes the THEN part of an IF /THEN statement if the condition is true. The reason is the **NOT.** Try it in immediate mode:

```
PRINT NOT 0
```

NOT is another Boolean operator, but it differs from AND and OR in that it does not require two numbers to function. Rather, it operates on only one number. NOT 0 is −1; −1 is Basic's way of saying true, therefore the program prints GOODBYE!

You may be equally surprised at the behavior of some of your typical IF/THEN functions when they are printed alone. Try:

```
PRINT 7<9
PRINT 10>1
PRINT 2=2
PRINT 2=4
PRINT 2+2=4
```

You've been using these all along and didn't realize it. Knowing how these functions work allows for much more sophisticated programming.

3 Tricks with Text

One of the signs of a professional program is good screen design. Commodore Basic 2.0 does not provide much in the way of instructions to accomplish this task; in fact, it provides only two functions; **TAB** ( ) and **SPC** ( ).

These functions seem similar, but there are important differences between them. TAB( ) causes Basic to move to a tab position indicated by the argument. (The **argument** is the number within the parentheses following a function.) If the cursor is already beyond that position, nothing happens. SPC( ), on the other hand, causes Basic to print the number of spaces indicated by the argument. The following short program should clarify this distinction:

```
10 FOR I = 1 TO 10
20 PRINT SPC(10);"X";TAB(I);"O"
30 PRINT TAB(10);"X";SPC(I);"@"
40 NEXT I
```

When you run this program you'll note that the TAB(I) in line 20 has no effect because the statement is already printing at position 10 thanks to the SPC( ) statement. On the other hand, the SPC( ) in line 30 does have an effect because it tells Basic to print spaces rather than to move to a tab position.

As useful as TAB( ) and SPC( ) are, they do not offer sufficient control to be the sole positioning commands. Fortunately, the Commodore 64 has the ability to incorporate its cursor positioning keys in a string. The CLR/HOME, CRSR UP/DOWN, and CRSR LEFT/RIGHT keys can be entered into the string simply by typing them. The screen will show some unusual characters, such as the inverse heart produced by the shifted CLR/HOME key, and a printer other than a Commodore Graphics model will produce a very interesting printout of these characters. But they do work. Try the following program to illustrate the use of the cursor up key:

```
5 REM PROGRAM MARQUE
10 MSG$="THIS IS THE MESSAGE"
19 REM B$ IS 40 DOTS OR OTHER CHARACTERS
20 B$="........................................"
30 MSG$=B$+MSG$+B$
40 PRINT "<SHIFT CLR/HOME>"
50 Y=20
60 FOR I = 1 TO Y:PRINT:NEXT
70 FOR I = 1 TO LEN(MSG$)-40
80 PRINT MID$(MSG$,I,40);"<CRSR UP>";
90 FOR T = 1 TO 50:NEXT T
100 NEXT I
110 GOTO 70
```

You'll have to press <RUN/STOP> to stop this program.

The key to this program is the CURSOR UP in line 80, which keeps the printing on the same line. Edit line 80 and remove this character and you'll see what I mean. The message no longer stays on the same line.

When the cursor control keys are entered into a program line, they display as a graphics character. It takes some time to get used to this. (See Fig. 3.1 for a list of these cursor control characters.)

Figure 3.1

Cursor Control Keys/Characters*

Cursor Home		
<Shift CLR/Home>		
Cursor UP		
Cursor DOWN		
Cursor RIGHT		
Cursor LEFT		

*Assuming computer is in UPPER case mode.

These characters will only show when you are in **quote mode;** that is, when you have entered an opening quotation mark in a print statement. If the quotation mark is not entered, they will be immediately executed. These characters will also show when you press <SHIFT INST/DEL> to insert text into a line.

Cursor control characters are trickier to use than the more traditional method used by other microcomputers, which specify the X and Y coordinates, but they are equally effective. The key to using them is to remember that Basic issues a line feed character at every PRINT unless it is terminated by a comma or semicolon. Examine the following:

```
10 PRINT "<SHIFT CLR/HOME>"
20 PRINT "<5 CRSR RIGHT>HELLO, WHAT'S YOUR NAME"
30 INPUT NA$
```

From reading this program you might expect the question to be printed on the top line of the screen. It's not, because there is no semicolon at the end of line 10. If you put one there, you'll find that the sentence does end up on the top line. You can get the input question mark to stay on this line by ending line 20 with a semicolon. Now add the following line:

```
40 PRINT "<CRSR DOWN><5 CRSR RIGHT>GOOD ";
        NA$;", LET'S START."
```

This will print the sentence one line lower than you might expect. After all, there are no prints without semicolons anymore. Remember, however, that you must press <RETURN> to end an input statement and push the cursor down an extra notch.

One way to deal with this is to HOME the cursor prior to the prints. That way, each print statement is controlled by reference to the home position:

```
40 PRINT "<CLR/HOME><CRSR DOWN><5 CRSR RIGHT>GOOD ";
   NA$;", LET'S START."
```

While this technique adds extra characters to each print string, it allows for a much calmer debugging process for heavily formatted screens. If you are doing a lot of printing, it might be useful to write a **subroutine** to accomplish the formatting. Here's an example of such a subroutine:

```
5 GOTO 20
10 PRINT "<CLR/HOME>";:FOR I = 1 TO V:PRINT"<CRSR DOWN>";:
          NEXT:FOR I = 1 TO H:PRINT"<CRSR RIGHT>";:NEXT:
          RETURN
20 PRINT "<SHIFT CLR/HOME>;
30 V=5:H=5:GOSUB 10:PRINT "HELLO, WHAT'S YOUR NAME?";
```

While this is easy to read, there are some disadvantages as well. It is a bit slower than directly encoding the cursor moves. Every time Basic does a GOSUB, it must search for the line to GOSUB to. A subroutine that is put at the head of the program, as shown above, will run a little faster than if it were at the end. Another disadvantage is that you cannot print in column 1 or in row 1 with this subroutine.

A peculiarity of Basic causes this problem: when Basic encounters a FOR/NEXT loop, it does not test to see if it is done until it has executed the

loop at least once. This subroutine, as written, will print at least one <CRSR DOWN> and <CRSR RIGHT> no matter what value V or H has. But this problem is easily solved by some creative reprogramming:

```
10 PRINT "<CLR/HOME>";:IF V > 1 THEN FOR I = 1 TO V:PRINT
         "<CRSR DOWN>";:NEXT
12 IF H > 1 THEN FOR I = 1 TO H:PRINT "<CRSR RIGHT>";:NEXT
14 RETURN
```

This little problem with FOR/NEXT loops is really a bug in Basic, but it is present in most microcomputer Basics. As we develop longer programs, notice how we compensated for it.

Getting Some Character

Commodore computers have always featured an extensive array of graphic text characters. These partially compensated for the lack of high resolution graphics in early PET computers. Creative use of these characters can result in rather elegant screens. However, getting the character you want may be difficult, for the Commodore 64 features two complete character sets. One set includes uppercase letters, and the usual array of special characters like asterisks and percent signs, as well as the graphic characters shown on the front of the keys. The second set has upper- and lowercase letters, but lacks most of the graphic characters. You can demonstrate this with the following:

```
FOR I=32 TO 128:PRINT CHR$(I);" ";:NEXT
```

This will print many of the printable characters in the current character set (more about this in a minute). Then press the Commodore key or <C=64> (it's the one beside the left shift) and the SHIFT key together. The set will suddenly change—all the uppercase letters become lowercase and the special graphics characters become the uppercase letters. Actually, there's more:

```
FOR I=160 TO 255:PRINT CHR$(I);" ";:NEXT
```

Many of these characters will also change when you press the Commodore and SHIFT keys together. You may wonder where characters 0 to 32 and 128 to 160 are. These are characters used to control various screen functions. For example, printing character #30—using the Basic commands

PRINT CHR$(30)—is identical to pressing the <CTRL> and <6> key, causing the text to change to green.

If the Commodore 64 did not offer quote mode, we'd have to use PRINT CHR$(30) in our programs rather than PRINT "<CTRL 6>". These control functions should be used within strings or literals because they save much execution time and memory. But don't use these functions if you do not have a way of preventing them from going to your printer.

The special Commodore screen control codes are also used by many manufacturers of printers for various functions, none of which relates to what Commodore is using them for. These codes might wreak havoc with a printer by causing it to shift into italics or stop printing and serenade you with beeps. If this happens to you, use the **CHR$** functions—they'll work as advertised on the screen and list blissfully on any printer.

There is a total of 255 characters, and if this seems familiar, it should. You do remember bits and bytes don't you? Commodore, and most other computer manufacturers, have used the whole capability of a byte rather than restricting it to the 90 or so characters we use in everyday writing. One character fits into one byte. As we saw earlier, we can directly access the screen by poking the correct code into the correct screen location.

The question here is which CHR$ codes correspond to which display code. This is an issue because, if you think about it, there'd be no sense in allowing someone to store the code to switch to green text (30) on the screen. Why? Once it's printed, it will have done its job. Besides, storing these codes on the screen would confuse our screen positioning commands (also CHR$ codes). The Commodore 64 kernal executes the CHR$ codes, activating screen functions and converting the other codes for displayable characters to new codes as it stuffs them onto the screen.

Confused? It's really very simple. If you want to print an A in the upper left of the screen, you have two options:

```
PRINT "<CLR/HOME>A"   or
POKE 1024,1
```

The kernal converts all As (CHR$(65)) to 1s before storing them on the screen. It does this to get as much out of a byte as possible. Try this:

```
FOR I=0 TO 255:POKE 1024+I,I:NEXT
```

There's More than CHR$

The Commodore and SHIFT key will show you the other character set in this form. These screen display codes (as opposed to the CHR$ codes) offer a new wrinkle—**inverse** characters. The first inverse character is an inverse commercial "at" sign (@). The regular at sign is poke code 0 and the inverse at sign is poke code 128.

Back to bits and bytes for a moment—128 is 0 with the high bit on. 0000 0000 is 0 and 1000 0000 is 128. It's simple to convert normal characters to inverse by merely setting their high bits! For those of you who may have forgotten: POKE LOC, PEEK (LOC) OR 128.

Flashing Text

Here's a little program to get flashing text on your Commodore screen:

```
5 REM FLASHING TEXT PROGRAM
10 PRINT "<SHIFT CLR/HOME>";
20 M$="HELLO THERE"
30 H=20-INT(LEN(M$)/2):V=10
40 FOR I = 1 TO V:PRINT "<CRSR DOWN>";:NEXT:FOR I = 1 TO
   H:PRINT "<CRSR RIGHT>";:NEXT
50 PRINT M$
60 FOR I = 1024+V*40+H TO 1024+V*40+LEN(M$)+H-1
70 POKE I,PEEK(I) OR 128
80 NEXT
90 FOR I = 1024+V*40+H TO 1024+V*40+LEN(M$)+H-1
100 POKE I,PEEK(I) AND PEEK(I)-128
110 NEXT
120 GOTO 60
```

If you prefer your text to be flashing without the inverse, try this modification:

```
10 PRINT "<SHIFT CLR/HOME>";
20 M$="HELLO THERE"
30 H=20-INT(LEN(M$)/2):V=10
40 PRINT "<CLR/HOME>;:FOR I = 1 TO V:PRINT "<CRSR DOWN>";:
   NEXT:FOR I = 1 TO H:PRINT "<CRSR RIGHT>";:NEXT
50 PRINT M$
90 FOR I = 1024+V*40+H TO 1024+V*40+LEN(M$)+H-1
100 POKE I,32 :REM 32 IS SCREEN CODE FOR A SPACE
110 NEXT
120 GOTO 40
```

Both programs use basically the same very useful positioning formula. Since we know the screen starts at location 1024 and that there are 40 characters across the screen, we can always find our place. The screen location is equal to the starting location (usually 1024) plus the vertical position times 40 plus the horizontal location: 1024 + V * 40 + H.

This is not as much math as it seems. I'm not much of a mathematician myself; all it takes is a little logic and careful consideration of the problem at hand. Whenever you notice regularity, there is the possibility of finding a formula to ease your programming task. Put this little one in your tool kit; it's useful.

I slipped another useful tool into these programs. The formula to calculate H (the horizontal position of the string) in line 30 is the classic centering formula. It's very simple; typists do it all the time, if they're word-processor-less. Take the length of the word or string to be centered and divide it by two. Subtract this from the middle tab position on the page. Since the Commodore 64 has a 40-column screen, the formula uses 20. The formula may look backwards to you, but remember that Basic first performs division before it subtracts.

Just a Little FUNctionality

Some programs use the screen positioning **algorithm** (see Box 5). Since this algorithm is often used, the logical place to put it is in a subroutine. In a subroutine, it can be used from many parts of your program without excessive duplication of code. Basic supports a special type of statement that provides an ideal place to put our screen positioning function: a defined function.

BOX 5.
ON ALGORITHMS

The term "algorithm" is often misunderstood. An algorithm need not be a complex mathematical formula. Rather, it can be anything from the little one-line trick to position text to a complicated subroutine. An algorithm is a precise solution to a problem. A program can be thought of as a series of algorithms each solving a small piece of the program problem. A branch of computer science is devoted to studying the effectiveness and efficiency of algorithms.

Functions are subprograms that return values. You're probably familiar with the **SQR** ( ) function. Old-timers called these library functions because they are part of the language; you need only call them from the library. Basic provides the capability of defining new functions. Unlike library functions, however, they are usable only within the program you are currently running.

To define a function, the key terms are **DEF FN** name (arg) =. Commodore Basic stores these functions much the way it stores variables, so the same rules hold for function names: only the first two characters count. Functions have parentheses to indicate their argument, the number you want to pass to the function.

For example, in SQR(4), the argument is the number 4. In **user-defined** functions in Commodore 64 Basic, the argument has no meaning so any number will do. To pass the function an argument, you use regular variable names much the way that you do when you write a subroutine. Enough of this theory—let's use them:

```
5 DEF FN LOC(Z)=1024+V*40+H
10 PRINT "<SHIFT CLR/HOME>";
20 M$="HELLO THERE"
30 H=20-INT(LEN(M$)/2:V=10
40 PRINT "<CLR/HOME>";:FOR I = 1 TO V:PRINT "<CRSR DOWN>";:
   NEXT:FOR I = 1 TO H:PRINT "<CRSR RIGHT>";:NEXT
50 PRINT M$
90 FOR I = FN LOC(0) TO FN LOC(0)+LEN(M$)+H-1
100 POKE I,32 :REM 32 IS SCREEN CODE FOR A SPACE
110 NEXT
120 GOTO 40
```

Notice that wherever the formula appeared, the FN LOC (0) now appears. Since the program calculates values for H and V in line 30, Basic uses these to control the value of I and therefore the screen position. Although, in this example, the function is used only once, it does make the program more readable and allows you to change only line 5 if you wanted to take it to a 23-columned VIC or an 80-columned PET. You'd also have to change the 20 in the centering algorithm.

The next program, "RANDOM BALL," makes use of the DEF FN statement. As the program gets more elaborate in the next chapter, you'll see the advantages of programming this way.

```
5 REM RANDOM BALL PROGRAM
10 PRINT "<SHIFT CLR/HOME>";
20 BALL=81
30 X=20:Y=12
```

```
40 SCRN=1024
50 DEF FN LOC(Z)=SCRN+Y*40+X
60 MOVE=INT(RND(0)*8)+1
70 ON MOVE GOTO 80,90,100,110,120,130,140,150
80 Y=Y-1:GOTO 160:REM MOVE NORTH
90 Y=Y-1:X=X+1:GOTO 160:REM MOVE NORTH EAST
100 X=X+1:GOTO 160:REM MOVE EAST
110 X=X+1:Y=Y+1:GOTO 160:REM MOVE SOUTH EAST
120 Y=Y+1:GOTO 160:REM MOVE SOUTH
130 X=X-1:Y=Y+1:GOTO 160:REM MOVE SOUTH WEST
140 X=X-1:GOTO 160:REM MOVE WEST
150 X=X-1:Y=Y-1:REM MOVE NORTH WEST
160 REM
170 IF X>39 THEN X=0
175 IF X<0 THEN X=39
180 IF Y>24 THEN Y=0
185 IF Y<0 THEN Y=24
200 POKE FN LOC(Z),BALL
210 GOTO 60
```

Computed Branching: ON/GOTO

The only statement in this program that you may not have used much is the **ON** variable **GOTO** statement. This statement could easily be replaced with several IF statements, but why use several statements if one will suffice? To understand how it works, we first must consider how the variable MOVE is calculated.

Line 60 contains another classic algorithm. Since Commodore Basic can only generate a random number between 0 and 1, we need a way to enlarge this to encompass, in this case, 1 to 8. The smallest number the **RND** statement can generate is 0. Zero times 8 is 0. The largest is .99999, which times 8 is 7.999. By taking the INTeger part of 7.999 we get 7, which leaves a range of 0 to 7. By adding 1 to this range we get 1 to 8. Thus, another algorithm is born.

Incidentally, the argument of the RND function can be a positive number, a negative number, or 0, depending upon what effect you wish to have. Zero causes the Commodore 64 to use its clock to generate the number; each positive number will cause the same sequence of numbers to be generated; and each negative number will cause the same number to be generated each call.

So MOVE can be any digit from 1 to 8. Sharp-eyed readers will have already noted that there are eight numbers following the ON GOTO statement. These are line numbers; the first is used if MOVE is 1, the second if MOVE is 2, and so on. This saves a lot of IF/THENing.

The lines from 80 to 150 actually increment the X and Y positions of the ball. One line for each move: up, down, left, right, and the four diagonals. Lines 170 to 185 check for out-of-bounds. If X or Y is out of bounds, the value is changed to the maximum or minimum value so that a ball moving off the top of the screen reappears on the bottom and vice versa. The technical term for this is **wrapping.** Finally, we POKE the ball's position and go back and get another MOVE.

Hold on to this one, because it's going to get a lot more exciting as we add color and sound in subsequent chapters.

An Excursion into Color

The Commodore 64 can produce 16 colors simultaneously. If you've studied your keycaps, you'll note that only eight colors are shown on the fronts of the number keys. To get text in these colors, type the color key while pressing the <CTRL> key.

The normal color scheme, a light blue border and dark blue screen area, can be changed as easily as the text color. Two locations in the VIC II chip control these colors. Location 53280 controls the border color and location 53281 controls the screen color. A simple POKE will change the color.

But what value to poke? For the first eight colors, just use the numbers on the keys but subtract 1 first. So to get white text, press <CTRL 2>—but to get a white border, key POKE 53280,1. Ditto with the screen color. By the way, if you make the screen, border, and text color the same, you'll see nothing. The characters, however, are there in screen memory regardless of their display color. When you change the screen color they will appear as if by magic. Try this:

```
5 T=500 :REM TIME FOR DELAY LOOP
10 POKE 53280,7 :REM SET BORDER TO YELLOW
20 POKE 53281,7 :REM SET BACKGROUND TO YELLOW
30 PRINT "<CTRL 8>" :REM SET TEXT TO YELLOW
40 PRINT "<SHIFT CLR/HOME>"
50 PRINT "<12 CRSR RIGHT>HELLO THERE"
60 FOR D = 1 TO T:NEXT
70 FOR I = 1 TO 10
80 POKE 53281,2 :REM SET BACKGROUND TO RED
90 FOR D = 1 TO T:NEXT
100 POKE 53281,7 :REM SET BACKGROUND TO YELLOW
110 FOR D = 1 TO T:NEXT
120 NEXT I
130 PRINT "<CTRL 2>"
```

This effect is interesting to experiment with, but I don't recommend using this technique except under very special circumstances. The flashing of the whole screen can be a bit disconcerting.

So where are the missing eight colors? Since there's already enough written on the keyboard to be very distracting, the extra eight color names are not written on keys. You can generate them, however, by pressing the <C=64> key in conjunction with the number keys (see Fig. 4.1). You can also POKE them for background or border colors.

Figure 4.1
Commodore 64 Colors

Number	Color	Number	Color
0	black	8	orange
1	white	9	brown
2	red	10	light red
3	cyan	11	dark grey
4	purple	12	medium grey
5	green	13	light green
6	blue	14	light blue
7	yellow	15	light grey

The program "COLOR BARS" will show you all 16 colors with labels. This program is handy to have around in case your TV or monitor needs to be adjusted. Incidentally, color 9 is supposed to be brown but on both my TV and my Commodore 1701 Monitor it appears as a dark, brick red. Since the other colors are fine, this seems to have been Commodore's intention.

```
5 REM PROGRAM COLOR BARS
10 PRINT "<SHIFT CLR/HOME><CTRL 2>"
20 SCRN=1024:CRAM=55296:C=-1:L=13
30 FOR I = SCRN TO SCRN+639:POKE I,160:NEXT
40 FOR I = 0 TO 639
50 IF I/40=INT(I/40) THEN C=C+1
60 POKE CRAM+I,C
70 NEXT
80 PRINT "<CLR/HOME>";
90 FOR I = 0 TO 15
100 PRINT I;
110 READ COL$
120 COL$=" "+COL$
130 IF I>9 THEN L=12
140 IF LEN(COL$)<L THEN COL$=COL$+" ":GOTO 140
150 PRINT "<CRSR LEFT>";COL$
```

```
160 NEXT
170 DATA BLACK,WHITE,RED,CYAN,PURPLE,GREEN,BLUE,YELLOW,
    ORANGE,BROWN,LIGHT RED
180 DATA DARK GREY,MEDIUM GREY,LIGHT GREEN,LIGHT BLUE,
    LIGHT GREY
190 RESTORE
200 FOR I = 0 TO 15
210 POKE 53280,I:POKE 53281,I
220 READ COL$
230 PRINT "<2 CRSR DOWN><7 SPACE>BACKGROUND ->";COL$;
    "<5 SPACE><3 CRSR UP>"
235 FOR K=1 TO 300:NEXT K
240 NEXT
250 PRINT "<3 CRSR DOWN>"
```

Colored Text

To print text in color, you simply imbed the correct CTRL code into the text. Like this:

```
PRINT "<CRTL 2>IT'S NOT EASY BEIN' <CTRL 6>GREEN<CTRL 2>"
```

This will result in a bizarre-looking line because the CTRL codes display as inverse characters on the screen (see Fig. 4.2) But it does work.

Figure 4.2
Color Display Characters

Color	Character
black	small box
white	inverse E
red	inverse British Pound sign
cyan	triangle
purple	partial checker board
green	inverse up arrow
blue	inverse left arrow
yellow	inverse pi sign
orange	inverse spade
brown	inverse top left quarter circle
lt red	inverse graphic x
dk grey	inverse dot
med grey	inverse club
lt green	inverse upright bar
lt blue	inverse diamond
lt grey	inverse plus sign

Color RAM

There's another way. We can POKE the color code into **color RAM.** Color random access memory? In addition to screen display area normally starting at 1024 and extending for 1000 memory locations, Commodore built a second RAM area called color RAM. It's actually no more colorful than any other RAM, but the values stored in these locations determine what color the text display will be. There is a one-to-one correspondence between color RAM and screen RAM. Color RAM starts at location 55296. This location contains the color of the text character found in location 1024. Try testing this:

```
PRINT "<CLR/HOME>THIS IS ORANGE THIS ISN'T"
FOR I = 0 TO 13:POKE 55296+I,8:NEXT
```

Note that these statements must be entered in the order given. If you try reversing them, the kernal will assure that the message is printed in the current print color (or last color text command issued).

This is a problem if you want to print more than just a bit of orange or if you are uncertain exactly where it will be printed. Fortunately, there is a solution—location 646 contains the color of text to be printed. The kernal merely puts the correct code number here following a PRINT <CRTL number>. We can directly POKE our color code here and get all the orange text we could possibly want. Try this: POKE 646,8. Eh voila!

Incidentally, numbers higher than 15 act as if they were chopped off at the fourth bit. That is, 16 (in binary 0001 0000) acts as if you POKEd a 0 (in binary 0000 0000). Naturally, computer people have a term for this—**nibble.** Half a byte is a nibble. The high nibble (the first four binary digits) is ignored here. Someday Commodore may make a 256-color computer where the whole byte is used. Actually, the whole byte is used in the multicolor modes, but we're getting ahead of ourselves.

If you try to PEEK locations in color memory, you're in for some surprises. The PEEKs don't necessarily return what you've POKEd there! This is a result of how Commodore has structured the memory of the Commodore 64. Remember that there is all that RAM and all that ROM—more than the 6510 microprocessor can effectively use.

In normal modes of operation, that is, with Basic going, all PEEKs to the upper reaches of memory refer to the ROM while all POKEs refer to the RAM. This is accomplished under the complete control of the 6510 micro-

processor and can be altered. In fact, we'll have to do it ourselves later to get at the character set. For now, if you want to remember specific colors, you'll have to assign them to variables in a program rather than relying on PEEKs.

Random Ball Revisited

Now it's time to spruce up that random ball program from Chapter 3. Retrieve the program from disk or tape and insert these new lines:

```
10 PRINT "<SHIFT CLR/HOME>";
15 POKE 53280,0:POKE 53281,0
20 BALL=81
30 X=20:Y=12
40 SCRN=1024
45 CRAM=55296
50 DEF FN LOC(Z)=SCRN+Y*40+X
55 DEF FN CLOC(Z)=CRAM+Y*40+X
60 MOVE=INT(RND(0)*8+1)
70 ON MOVE GOTO 80,90,100,110,120,130,140,150
80 Y=Y-1:GOTO 160:REM MOVE NORTH
90 Y=Y-1:X=X+1:GOTO 160:REM MOVE NORTH EAST
100 X=X+1:GOTO 160:REM MOVE EAST
110 X=X+1:Y=Y+1:GOTO 160:REM MOVE SOUTH EAST
120 Y=Y+1:GOTO 160:REM MOVE SOUTH
130 X=X-1:Y=Y+1:GOTO 160:REM MOVE SOUTH WEST
140 X=X-1:GOTO 160:REM MOVE WEST
150 X=X-1:Y=Y-1:REM MOVE NORTH WEST
160 REM
170 IF X>39 THEN X=0
175 IF X<0 THEN X=39
180 IF Y>24 THEN Y=0
185 IF Y<0 THEN Y=24
190 POKE FN CLOC(Z),INT(RND(0)*15)+1
200 POKE FN LOC(Z),BALL
210 GOTO 60
```

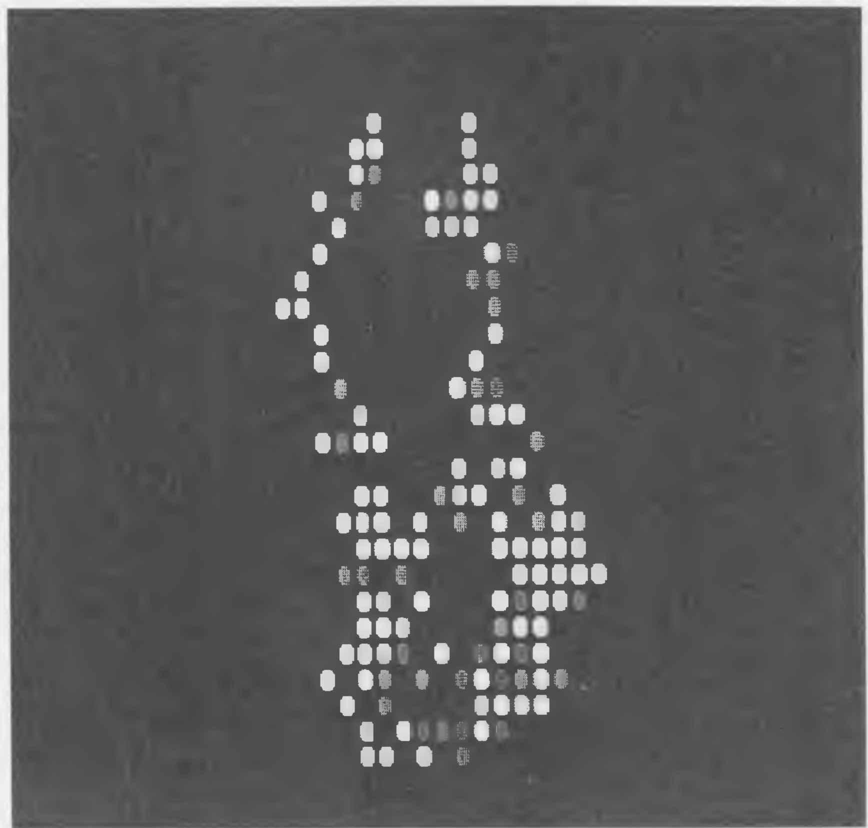
Since color RAM is structured just like screen memory, the function used to change colors can look pretty much like the screen POKE function except for the change in base locations: 1024 for screen POKEs, and 55296 for color POKEs.

Some slight modifications may be more to your liking.

```
20 BALL=81:C=2
160 IF RND(0)>.5 THEN C=INT(RND(1)*15)+1
190 POKE FN CLOC(Z),C
```

With these changes, the ball only changes color when a random number larger than .5 is generated in the IF statements. Therefore, color patterns are created in addition to dot patterns. Fig. 4.3 shows what your screen should look like. Since the dots are placed randomly, however, the screen will be different each time you run the program.

Figure 4.3
Random Ball Program with Colored Balls



Finally, a more comprehensive revision may make this program more interesting.

```
170 IF X>19 THEN X=0
175 IF X<0 THEN X=19
180 IF Y>12 THEN Y=0
185 IF Y<0 THEN Y=12
190 OX=X:OY=Y
200 GOSUB 500
210 X=39-X
220 GOSUB 500
230 Y=24-Y
240 GOSUB 500
250 X=OX
260 GOSUB 500
270 Y=OY
280 GOTO 60
500 POKE FN CLOC(Z),C
510 POKE FN LOC(Z),BALL
520 RETURN
```

The screen and color POKEs are here relegated to a subroutine, and several alterations need to be made to the calculation of X and Y. Note that in the out-of-bounds calculation section (lines 170 to 185), the maximum values for X and Y have been altered to 19 and 12 respectively. This represents half of the original values so the Xs and Ys will be restricted to half their range or the top one-quarter of the screen.

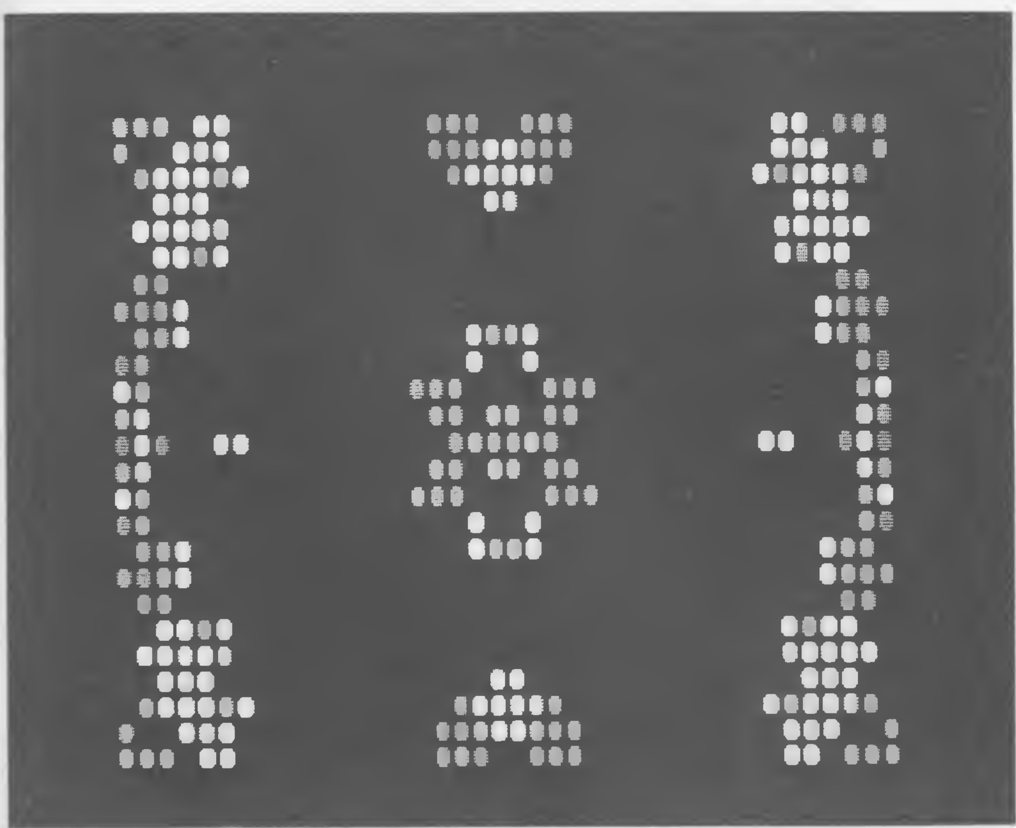
The new variables OX and OY (Old X and Old Y) are used to store these original values of X and Y. A new X is calculated that is the same number of spaces from the right margin. Since Y hasn't changed, this new X will plot in the upper lefthand portion of the screen. Then a new Y is calculated, like X representing the original Y but from the bottom margin.

This will plot in the lower lefthand quarter of the screen. Finally, the original value of X is restored and, with the new value of Y, it causes a dot to be printed in the lower righthand quarter of the screen. Then Y is restored to the original value and we loop back to calculate a new MOVE. Though it's a bit hard to describe, it's really quite simple in practice. Fig. 4.4 shows a sample screen from this program.

You might want to modify this program to print other characters or even generate different characters based on a random number. I prefer the program on a black background, but you might also want to change the color of the background or have the program do it based on a random number.

The Commodore 64 possesses two other color modes: multicolor and extended background color. We'll look at these in future chapters.

Figure 4.4
Random Ball Program—Final Revision



5 An Introduction to Animation

There are two basic forms of computer animation: raster and vector. **Raster animation** is block animation, in which the figure to be animated is described or programmed to be a square area, a block, made up of individual picture elements or **pixels.** Within this block, the pixels can be either turned on or turned off giving form to the design. The best example of raster or block animation is *Pac Man*—you can almost see the block in which Mr. Pac (or is it Mr. Man?) and his chasing ghosts were drawn in.

The other form of animation is **vector animation.** In this type of animation, the figure is presented to the computer as a vector, or line. A vector is something that has direction and magnitude. Unlike block animation, then, vector animation describes the form of the object and lets the special vector hardware decide which pixels to light in drawing the object.

Both types of animation have their own uses. Raster animation is best for situations where one wants a very colorful display with lots of internal movement in the objects—like most arcade games. On the other hand, where objects must be seen in perspective or with three-dimensional effects or must be magnified or rotated, vector animation is the one to choose. No popular home computer comes with true vector hardware. Even most video games are raster games; one popular exception is *Asteroids*.

Since vector animation is not supported on the Commodore 64, we'll stick to the more popular raster or block animation.

Basic Animation

The basic animation cycle is shown in Fig. 5.1. While this may seem overly simplified, it is the basis for all computer animation.

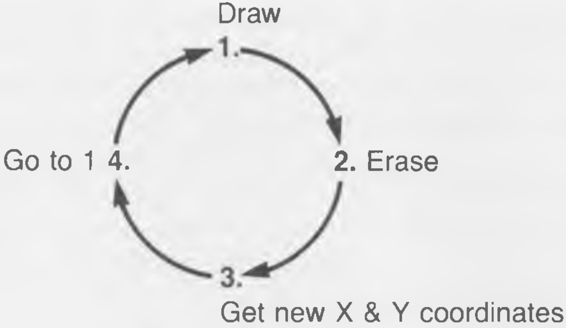


Figure 5.1
The Animation Cycle

The first step in creating an animation is designing the object to be moved. I generally resort to graph paper for this. Fig. 5.2 shows the fish that is the basis for all the animation examples in this chapter.

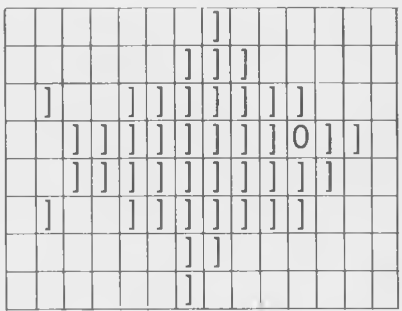


Figure 5.2
The Design for the Text Animated Fish

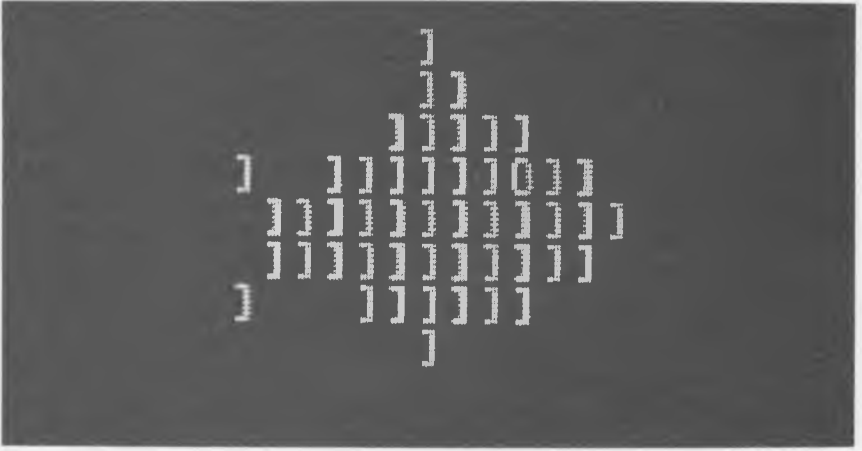
Next, we must create a program that will place this fish into the animation cycle. Here's one:

```
5 REM PROGRAM TEXT ANIMATE
10 GOSUB 100
20 FOR I=1 TO 23
30 PRINT "<CLR/HOME><4 CRSR DOWN>"
40 FOR J=1 TO 8:PRINT SPC(I);A$(J):NEXT
50 PRINT "<CLR/HOME><4 CRSR DOWN>"
60 FOR J=1 TO 8:PRINT SPC(I);A$(J):NEXT
70 NEXT I
```

```
95 END
100 A$(1)="        ]        "
110 A$(2)="       ]]]       "
120 A$(3)="]   ]]]]]]]      "
130 A$(4)=" ]]]]]]]]]Ø]]   "
140 A$(5)=" ]]]]]]]]]]]    "
150 A$(6)="] ]]]]]]]       "
160 A$(7)="       ]]        "
170 A$(8)="       ]         "
180 B$=   "                 "
190 PRINT "<SHIFT CLR/HOME>";
200 RETURN
```

This program may not produce the most attractive fish around, but it sure does swim . . . (see Fig. 5.3). The animation cycle is right there in plain Basic. Line 4Ø prints the fish and line 6Ø erases it. The two other lines in the animation loop assure that the prints occur in the correct screen position. X positioning is handled by the main I loop.

Figure 5.3
The Text Animated Fish in Its Natural Habitat



You may note that the fish flickers a bit. Part of the flickering is due to the slowness of Basic but much of it is because you are watching the erase. For a moment, the screen is blank. This is the flicker. There are several ways to avoid the flickers. First, and simplest, is to include the erase in the draw cycle. If this seems impossible, try the following edits:

```
100 A$(1)="^        ]        "
```

(note: put another space here)

When you run this, the flicker will be gone. The fish will also go a bit faster since there is half the time-consuming PRINTing to do. This program shows how simple it is to accomplish text animation. With smaller blocks and with custom-designed character sets, such techniques can be quite powerful. We'll get to these in the next chapter.

But this program only moves a stationary fish across the screen. Some computer animators claim that true animation requires internal movement. How about internal animation? Why not!

Modify the program as follows (note changes to the basic fish design):

```
5 DIM A$(18)

4Ø FOR I=1 TO 8:PRINT SPC(I);A$(J-1Ø*(I/2=INT(I/2))):NEXT

2ØØ A$(11)="       ]      "
21Ø A$(12)="      ]]]     "
22Ø A$(13)="    ]]]]]]]   "
23Ø A$(14)=" ]]]]]]]]]-]] "
24Ø A$(15)=" ]]]]]]]]]]]  "
25Ø A$(16)="   ]]]]]]]    "
26Ø A$(17)="      ]]      "
27Ø A$(18)="      ]       "
28Ø RETURN
```

Now you have two frames that will alternate with each other as the fish moves across the page. The most difficult part of this program is the change in line 4Ø. I snuck another algorithm in here; one more for your tool kit.

Since we want to alternate frames, we need a way to indicate which fish to draw. I carefully chose the index numbers for the arrays: 11 is 1 + 1Ø, 12 is 2 + 1Ø, and so on. This provides a clear relationship between the indexes and, therefore, between the frames. Since we are using the loop variable, I, to determine the X position why not use it to determine which frame to put up?

Consecutive numbers have a very interesting property: odd and even numbers alternate. How do you tell the computer whether a number is odd or even? Easy. Divide the number in question by two; if the result of this is the same as the integer part of the same division, the number is even. If not, and there is a remainder in the division, the number is odd.

For example: 12/2 = 6 and INT (12/2) = 6 but
13/2 = 6.5 and INT(13/2) = 6 aha!

While this gives us the method of telling whether the loop index is odd or even, now we need a way to bump the array index up from 1 to 11 for one

number and back to 1 for the next number. Remember our old friends Boolean values from Chapter 2? We'll take our algorithm to determine even and odd and frame it so it creates Boolean values: I/2 = INT(I/2). It looks like the usual variable assignment statements (A = K/2).

But look closer. The right side of the equal sign is not a variable name, it's a process: divide I by 2. Try this in immediate mode and you'll get a SYNTAX ERROR. Or, you can add a weird new line to your program—if you substitute numbers by typing 2/2 = INT(2/2), you'll add the creative new program line:

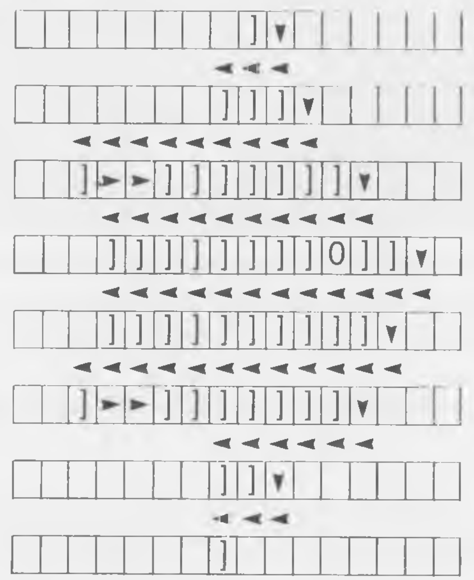
```
2 /2=INT(2/2)
```

and get a SYNTAX ERROR when you run the program.

On the other hand, PRINT 2/2 = INT(2/2) does work. Try it. These expressions create Boolean values representing false and true. In effect you are asking the computer a true or false question: Does 2 divided by 2 equal the integer part of 2 divided by 2?

Terrific. Now we have a 0 and a −1. By simple multiplication we can convert this to a 0 and a +10. 0 times −10 is 0, whereas −1 times −10 is +10. Add this number to our original index numbers (determined by J) and

Figure 5.4
The Design Grid for the Imbedded Cursor Moves



we get 1 to 8 or 11 to 18 depending on whether our main loop index is even or odd.

Sounds so simple, and it will be from now on. By the way, if you want to reverse the order of frames, merely substitute the opposite of equals for the equal sign. You may recall that the opposite of equals is "< >," read as "not equal to" but actually meaning "greater than or less than."

There is another way to present the animation data to the computer. The array we've been using is fine and it works. Furthermore, the technique will also work on other computers. But since the Commodore 64 offers the possibility of encoding the cursor control keys within a string, we can make the array much smaller. The design process becomes a bit more complicated and requires some careful preplanning as shown in Fig. 5.4.

For simplicity, we'll return to a one-frame animation, although this technique works equally well with multiframe animations too.

```
5 REM ONE-FRAME ANIMATION
10 GOSUB 100
20 FOR I=1 TO 23
30 PRINT "<CLR/HOME><4 CRSR DOWN>"
40 PRINT SPC(I+9);A$
50 PRINT "<CLR/HOME><4 CRSR DOWN>"
60 PRINT SPC(J+9);B$
70 NEXT I
95 END
99 REM DON'T HIT RETURN 'TILL AFTER THE CLOSING QUOTE!
100 A$(1)="]<CRSR DOWN><2 CRSR LEFT>]]]<CRSR DOWN><8 CRSR LEFT>
        ]  ]]]]]]]<CRSR DOWN><9 CRSR LEFT>]]]]]]]]]Ø]<CRSR
        DOWN><11 CRSR LEFT>"
110 A$(2)="]]]]]]]]]]<CRSR DOWN><11 CRSR LEFT>]  ]]]]]]]
         <CRSR DOWN><5 CRSR LEFT>]]<CRSR DOWN><2 CRSR LEFT>]"
120 A$=A$(1)+A$(2)
130 B$(1)="<1 SPACE><CRSR DOWN><2 CRSR LEFT><3 SPACE><CRSR DOWN>
         <8 CRSR LEFT><10 SPACE><CRSR DOWN><9 CRSR LEFT>
         <11 SPACE><CRSR DOWN><11 CRSR LEFT>"
140 B$(2)="<10 SPACE><CRSR DOWN><11 CRSR LEFT><10 SPACE>
         <CRSR DOWN><5 CRSR LEFT><3 SPACE><CRSR DOWN>
         <2 CRSR LEFT><1 SPACE>"
150 B$=B$(1)+B$(2)
190 PRINT "<SHIFT CLR/HOME>";
200 RETURN
```

This program is rather difficult to type in. Concentrate on getting lines 100 and 110 right; lines 130 and 140 are exactly the same but with spaces substituted for the brackets.

All you need to do, then, is move the cursor up to line 100, change the line number to 130, and trace the line, substituting spaces for brackets. When you press <RETURN>, you'll have entered line 130 without tears. Do the same for line 140. The rationale for lines 120 and 150 is simple.

The Commodore 64 can only support an 80-column line in Basic. All the cursor moves make the fish considerably bigger than 80 characters. Strings, however, can be up to 255 characters in length. These lines join the top half of the fish to the bottom half. By storing the whole fish in one string, the program runs considerably faster than the previous program.

Try producing a version of our second animation program using imbedded cursor moves. You can edit lines 100 and 110 to insert the extra spaces necessary for the fish to erase itself. This technique results in an amazingly fast, smooth-moving animation. It's worth the effort.

Printing to the screen is slow, however. Basic must deal with text positioning and with character colors in order to get the object on the screen. POKEs directly to the screen should be faster—at least this is the theory. But for some unknown reason, screen POKEs on the Commodore 64 yield a slower animation, even when the correct POKE locations are predefined and Basic has no calculations to do. This is not true of most other personal computers.

The Commodore 64 offers sprites as a better way to accomplish animations (see Chapter 8). There are times, however, when character-style animation is preferable to sprites. If, for example, you want your program to also run on the VIC or PET computers, you must stick to character animation. But you don't need to hold to Commodore's character set. By using custom-designed characters, character animation can reach quite sophisticated levels.

Quite a Character

The fish that we worked on earlier would have looked better if the brackets that were used to create it had looked like scales. How can we change the appearance of the text characters? By actually designing a character to look like a scale, for example, and placing it into the character set where the bracket normally is. Similarly, your programs can display a diacritical mark like an umlaut or a Greek letter or even a space invader. Designing custom characters is a relatively simple task on the Commodore 64.

Where VIC Gets Its Characters

You've probably never thought about where the computer gets the designs for the letters that it puts on the screen as you are typing or as a program is running. But by finding out, we can spiff up our programs and do things, like write subscripts or superscripts, that cannot easily be done without altering the computer's character set.

When you turn the computer on or when you press <RUN/STOP RESTORE> the kernal sets up pointers to the Commodore 64's ROM character set. This set is located in high memory, but an electronic trick makes the kernal think that the set is down near the screen. This is necessary because of the VIC II chip, which can only address 16K of memory. A single VIC II memory location controls where the chip finds its characters. Remember that single memory locations can only hold numbers up to 255. This number is the page where the character set images can be found.

The only problem with this location is that it is also used to tell the VIC II where to put the screen. The location is 53272 and, when you turn your computer on, it looks like this:

```
53272   | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |     (decimal:21)
```

To confirm this, type PRINT PEEK(53272). The high nibble (the four bits counting in from the left) control where the screen is in memory, according to the value in Fig. 6.1. The next three bits control the location of the character set, according to the value in Fig. 6.2. The 1s bit, the rightmost one, has not been defined by Commodore although it seems to be kept on (set to one) even if you POKE it off (set it to zero). Since we only want to change the three bits that control the character set location, we need to do some ANDing and ORing before we can POKE the new value into this location. The following statement will do the job:

```
POKE 53272,(PEEK(53272) AND 240) OR CB
```

where CB contains the correct character bank from Fig. 6.2.

Figure 6.1
Screen Memory Control Values*

Value	Location	Value	Location
0	0	128	8192
16	1024 (normal value)	144	9216
32	2048	160	10240
48	3072	176	11264
64	4096	192	12288
80	5120	208	13312
96	6144	224	14336
112	7168	240	15360

*These values must be added to the bank of memory the VIC is looking at. Since it is normally looking at VIC bank 0, the values in this table can be looked on as absolute locations. However, if bits 0 or 1 of location 56576 are changed from their powerup values, then the correct offset must be added to these values.

Figure 6.2
Character Set Control Values*

Value	Location
0	0
2	2048
4	4096 (normal value)
6	6144 (normal value after press of <C=64 SHIFT>)
8	8192
10	10240
12	12288
14	14336

*See note for Fig. 6.1. Note that the special addressing that allows the VIC II to "see" its high memory character sets in the normal VIC address range is only available for VIC banks 0 and 2. So, if you need characters on the screen, you are limited to these two banks.

Since we haven't used this algorithm yet, let's look at it in detail:

Basic will do whatever is in parentheses first. That is, it will retrieve the value stored in location 53272. If the computer has just been turned on or if you just pressed <RUN/STOP RESTORE>, it should find a value of 21 or 00010101 in binary.

Next, it will AND this value with 240. The result is 16 or 00010000 in binary. This is the screen location control value. Check 16 on Fig. 6.1 to confirm this.

Next Basic will OR the 16 with our character bank. We'll assume that CB has a value of 14, which was obtained from Fig. 6.2, to locate our character set at location 14336. Fourteen in binary is 00001110. The OR operation produces a value of 30.

This value is now POKEed into the VIC control register. The whole process is illustrated in Fig. 6.3.

Figure 6.3
A Bit-by-Bit Analysis of the Algorithm to Change Character Set Location

	Ø	Ø	Ø	1	Ø	1	Ø	1	
Bit Values	128	64	32	16	8	4	2	1	
	Ø +	Ø +	Ø +	16 +	Ø +	4 +	Ø +	1	= 21

	Ø	Ø	Ø	1	Ø	1	Ø	1	(21)
AND	1	1	1	1	Ø	Ø	Ø	Ø	(24Ø)
Result	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	(16)

	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	(16)
OR	Ø	Ø	Ø	Ø	1	1	1	Ø	(14)
Result	Ø	Ø	Ø	1	1	1	1	Ø	(3Ø)

Notice that the AND operation has the effect of extracting the screen location bits and that the OR operation combines them with our desired character set location bits. The nice thing about this little algorithm is that we only have to know where we want to put our character set—the screen is taken care of. If you know that the screen is going to be at its usual place (1024) and that your character set will be at 14336, then it is simpler to just POKE 53272, 30 (or 31 since the 1s bit is unimportant here). However, should you want to move the screen or create a second screen, you must calculate the correct POKE. This algorithm does it all for you.

Designing Your Characters

Now that you know how to tell the Commodore 64 to get its characters from another place in memory, you need to put some characters there. If you don't, you won't be able to read what the computer is printing. Try changing set location now to see what I mean. (You'll have to press <RUN/STOP RESTORE> to return to the real world.) Type:

```
POKE 53272,(PEEK(53272) AND 240) OR 14
```

The seemingly random garbage on the screen is actually the same information that was on the screen before you pressed <RETURN>—we didn't change the location of the screen, the AND took care of that. All that has changed are the character images that the VIC uses to display the screen information.

It's highly unlikely that any of the memory in locations 14336 and up (character bank 14) actually contains a good character image. Rather, it contains random garbage or, if you've been playing in Basic, it may contain some of your variables or even your program. We will put our character images in these locations.

Because the VIC will be unfailingly faithful when it gets its characters, we should have an image of every character that we intend to display in our program. For a game, you might only need a few characters but for most purposes, it's better to have the whole character set available.

In this way, we can replace those characters that we don't need, but still have all the characters required to read the program listing or error messages. Recall that there are 256 characters in the Commodore 64 set and that there are two sets: one consisting of uppercase letters and graphic characters and the other consisting of upper- and lowercase letters. Also recall that when the high bit is turned on, the character is inverted.

Each of these characters represents an eight-dot by eight-dot grid. Each horizontal dot is represented by a single bit in memory. Therefore, each character requires eight bytes of memory: one byte for each of the horizontal lines in the character, eight horizontal lines. When a bit is on, a dot is displayed; when a bit is off, no dot is shown. The dot grid for the letter A is shown in Fig. 6.4.

The two columns of zeroes on either side of the letter provide space between adjacent letters on the screen. Similarly, the bottom row of the character has been left blank (zeroes), which also provides spaces between rows of characters on the screen. Without these empty columns and rows, all

Ø	Ø	Ø	1	1	Ø	Ø	Ø
Ø	Ø	1	1	1	1	Ø	Ø
Ø	1	1	Ø	Ø	1	1	Ø
Ø	1	1	1	1	1	1	Ø
Ø	1	1	Ø	Ø	1	1	Ø
Ø	1	1	Ø	Ø	1	1	Ø
Ø	1	1	Ø	Ø	1	1	Ø
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

Figure 6.4
The Dot Matrix for the Letter A

the letters would run together on the screen. The graphics characters don't have these empty rows and can therefore be used to build solid, continuous boxes and borders.

These eight bytes are stored sequentially. Thus, character number two starts on the ninth byte from the start of the character set, character number three starts on the 17th byte from the start, and so on.

The two character sets, each of 256 characters, make a total of 512 different character designs. Each character requires eight bytes of storage for its dot pattern. This makes a total of 4,Ø96 bytes to completely define the Commodore cast of characters. That's a lot of POKEing if we are starting from scratch. On the other hand, do we really need all of these characters?

If your program never goes from the upper/graphics set to the upper/lower set, we can cut the amount of memory needed in half, to 2,Ø48 locations. If the inverse characters are never used, the amount of memory needed is halved again to 1,Ø24 locations. But even with this reduced set, 1,Ø24 locations requires 1,Ø24 bytes to be POKEd before switching character sets. Not to mention the problem of converting our eight by eight bit designs to eight decimal numbers for POKEing!

Before you panic, remember that the Commodore 64 already has a character set built in. Why not use it for our basic character set and alter only those characters used by the graphics in our program? After all, computers are terrific at pushing bytes around. Let's let our Commodore 64 do the boring work while we concentrate on creative character design.

Conceptually, then, it's a piece of cake: all we need to do is POKE the Commodore set into the locations where we want to store our character set. Since PEEK will tell us what's in the Commodore locations, we've got it made. Needless to say, it's not that easy.

Remember the creative electronics that I alluded to in the footnote to Fig. 6.1? Since the VIC II chip can only address 16K of memory at a time,

Commodore engineers had a problem. The built-in character set would have to be in all four 16K banks of the computer for characters to be displayed in all the banks. They decided to give up two of these banks; after all, with sprites and high resolution graphics many applications wouldn't need characters in all banks.

We need only two copies of the set. But this is a RAM-based computer and RAM disappears when the power is turned off. Character sets are best stored in ROM. Why duplicate a ROM and give up valuable RAM? So, though the character set ROM is physically at locations 53248 to 56896, when the VIC II needs to find a character design, it acts as if it is within either bank 0 or bank 2 of VIC II memory. (Commodore engineers did not give the same capabilities to banks 1 or 4.)

So now we know where the character set is, even if it seems to be elsewhere when it is being used. But there's an interesting conflict here. Haven't we been POKEing around this area? For example, the background color POKE location is 53281—smack in the middle of character number 4. (53281 is 33 bytes up from 53248. At eight bytes per character, this location is in character four.) Enter the phantom ROM.

The ROM character set is only available when the VIC II requests it; at all other times this area consists of RAM locations used to control the VIC II chip and, for that matter, all of the other special purpose processors in the Commodore 64. When you PEEK or POKE this area, then, you're working with the RAM information. After all, it is useless to POKE a ROM. But since the VIC can get to the ROM so should we.

A Brief Interruption

To understand how the ROM character set is obtained, we need to use another location/switch built into the Commodore 64. This location tells the kernal where to find the I/O control information. Without this information the VIC II, the SID, and the other special processors cannot work; the computer might as well be blind, deaf, and dumb.

The I/O information happens to be stored in the ROM locations that underlie our secluded character set. (You might want to refresh your memory by referring to the memory map, Fig. 2.1.) Once we've changed the control location, our Commodore 64 will be incapable of I/O. To handle this, we'll just make it impossible to do I/O until we've restored the I/O control location to its proper value. Does this make sense?

The Commodore 64's I/O is **interrupt driven.** That means that the microprocessor can be directed to stop what it's doing when an I/O operation is required. All modern microprocessors have interrupt capability, which is built into the hardware of the chip. Interrupts need to be timed so that they are accomplished in the correct order.

Every time you press a key on the keyboard, an interrupt is generated to indicate that the kernal needs to start the keyboard input handler. All we have to do is turn off the clock (or in this case timer, because it doesn't keep the time of day, but only elapsed jiffies or microseconds). With the timer off, the computer is incapable of I/O but it still can work internally.

Sounds complicated but it's simple to do:

```
100 POKE 56334,PEEK(56334) AND 254     :REM TURN OFF TIMER &
                                        THEREFORE INTERRUPTS
110 POKE 1,PEEK(1) AND 251             :REM CHANGE I/O LOCs
```

At this point, we've turned off interrupts, and switched the competing RAM out of the way, and we can PEEK the character set.

```
30 CHAR = 14336 :REM LOCATION OF OUR CHARACTER SET
120 FOR I = 0 TO 2047:POKE CHAR+I,PEEK(53248+I):NEXT I
```

This assumes, of course, that CHAR contains the start of our new character set location (for these examples, 14336). The process will take time—2,048 bytes is nothing to sneeze at. Copying all 4,096 characters (both upper/graphics and upper/lower sets) would take even longer. While this is happening, your Commodore will be dead to the world—pound on the keyboard if you want, it'll do no good. When we've finished the copying process, it would be a good idea to restore our Commodore 64's communications capabilities.

```
130 POKE 1,PEEK(1) OR 4               :REM RESTORE I/O LOCs
140 POKE 56334,PEEK(56334) OR 1       :REM RESTORE TIMER
150 POKE 53272,(PEEK(53272) AND 240) OR 14 :REM CHANGE SET
```

Be very careful when you type these statements. A small error could cause your Commodore 64 to freeze and crash. Even <RUN/STOP RE-<STORE> won't help—remember, you turned off the I/O capabilities! It's better to SAVE the program before running it. That way, if you have made a small error that crashes the computer, you can turn the machine off and on

again, retrieve your program from cassette or disk, and fix the error without having to retype the whole program.

This program can now be run. You may find the line numbering a bit eclectic, but keep to it. We're going to write our first major editor—a character set editor. This editor will allow you to design your own characters and save them so that they can be used in other programs. And we'll let the computer do all the hard work!

If you run it, you won't see any major change in the screen. In fact, you won't see any change in the screen! We've copied all of the characters that can be displayed in the upper/lower set. Switching to this set will be a surprise—try it by pressing <SHIFT C=64>. Instant garbage.

We now have a soft character set, a character set in RAM, and the ability to change the characters. The character set is arranged in screen display code order, rather than ASCII or CHR$( ) order. The first character in the display set is the commercial "at" sign. (@). Let's change this to a downward arrow. (Curiously, the Commodore 64 set includes an up arrow and a left arrow, but no right or down arrow.) My design is shown in Fig. 6.5.

Figure 6.5
Downward Arrow

Binary	Decimal
00011000	24
00011000	24
00011000	24
00011000	24
00011000	24
01111110	126
00111100	60
00011000	24

The on and off dots in the design represent bits; each row of the character represents a byte. Basic requires that we convert these binary numbers into decimals. Once converted, they become a series of POKEs:

```
POKE CHAR,24:POKE CHAR+1,24:POKE CHAR+2,24:
POKE CHAR+3,24:POKE CHAR+4,24:POKE CHAR+5,126:
POKE CHAR+6,60:POKE CHAR+7,24
```

These POKEs will all fit on one line of the screen, but you can type them individually if you wish.

```
CHAR=14336
```

If you've just run our character copy program, the line above is not necessary, as CHAR has a value from line 30 of the program. However, if you've typed NEW or done something else, this line will restore the value of CHAR.

Now type a few @s. Turn reverse video on by pressing the <CTRL 9> and try it again. We've only changed the normal @ sign, not the reverse one. (In terms of display codes, remember that the regular @ is code 0 while the inverse @ is 128.)

This altered character will function just as the old @ did. The internal storage code has not been changed, only the display of the internal code. For example, if you change the plus sign to look like a happy face, typing PRINT 2 + 2 would display as PRINT 2 <happy face> 2 but would still print the answer: 4. This may be a hard concept to grasp, but remember that the screen display is only for us humans. The computer cares only what the internal code for the character is, not what it looks like.

It is important to note that on the Commodore 64, all upright lines in a character should be two dots or bits wide. This is true for all computers that were designed to be used with home televisions. The problem is how a television works, not how a computer works. If upright lines are less than two dots wide, the line will be slightly colored rather than pure white (or whatever the display color is). Televisions just don't have the resolution to display one dot clearly enough. If you are working on a high quality monitor, your characters can be as slender as you want them to be.

To restore the characters to the regular set, you can press <RUN/STOP RESTORE> or type POKE 53272,21. Both of these instructions restore the character set pointer to the ROM set and make it impossible to POKE in a new design. Assuming your redesigned set is still in memory, however, you can restore the pointer and regain the edited set merely by POKEing the location into 53272. There's no need to go through the lengthy copy process.

In the next chapter, we'll construct the editor.

The Character Set Editor

Our character set editor should be easy to use—after all, we're writing it to give us a tool to make programming easier. There must be a minimum of three items on the screen: the character being edited in its normal form, the character as seen in its current state, and an edit grid where the actual editing is taking place.

The editor will be constructed from subroutines. This chapter follows my program design process. Occasionally we'll have to edit an existing line to accommodate a new subroutine as we add functions and revise existing ones. However, the whole program works while we're refining it. That is the secret to successful programming of large projects. The pros call this step-wise refinement, but I'd rather think of it as common sense. You'll find the full program listing at the end of the chapter.

Building a Screen

We'll include the whole 128 normal display character set on the screen. This will make it simpler to be consistent in the design process. The following subroutine will build this display:

```
200 PRINT:PRINT:PRINT"<6 SPACE>0 1 2 3 4 5 6 7"
205 PRINT "     <CTRL 9><C=64 A><15 SHIFT C><C=64 S><CTRL 0>";
210 L=G2-24
220 FOR I = 0 TO 127
230 IF I/8 <> INT(I/8) THEN 260
235 PRINT
240 L=L+24
250 PRINT SPC(-1*(I<100)-(I<10));I;"<CTRL 9><SHIFT B>
    <CTRL 0><15 SPACE><CTRL 9><SHIFT B><CTRL 0>";
260 POKE L,I
```

```
270 L=L+2
280 NEXT I
285 PRINT
290 PRINT "<5 SPACE><CTRL 9><C=64 Z><15 SHIFT C><C=64 X>
    <CTRL 0>";
295 RETURN
```

This subroutine uses one program constant, G2, and one temporary variable, L. I am depending on G2 to remain constant throughout the program, but I must be able to change this location with every byte of a character. Therefore I created the temporary variable L. The next paragraph details why line 210 subtracts 24 from G2 only to have line 240 add it back. Character #0 will be POKEd to location L, character #1 to location L+2, character #3 to location L+4, and so on. Since L is being incremented in twos, we need to increase L by two after POKEing a character.

You should recall the little even/odd programming outlined in Chapter 5. I've used the trick here to print the characters out in a rows of eight. That is, I/8 equals INT(I/8) only when the loop variable (I) is 8 or a multiple thereof. When I is 8 or one of its multiples, we want to increment our POKE location by a whole screen line (which turns out to be 24 in this program) and print the value of I and our uprights at either side of the matrix.

This presents a problem because 0 is one of the values that generates a −1 or "true" in this situation. But we don't really need to add 24 to our screen location variable when I is zero—it's already right. This problem is cleanly avoided by subtracting 24 from the location before starting the whole loop—zero is no longer a special case!

These characters will be framed in an attractive border made of graphics characters. But a problem arose in printing the display codes for the character outside the frame. The display codes range from 0 to 127, and if they are to be printed neatly to the screen we must find some way to compensate for the difference in the lengths of the numbers.

Basic will print one space in front of the number, so in case the number is negative there will be a place for the sign. We have no way of telling Basic that there won't be negative numbers here. The solution I chose is to use a Boolean calculation. Line 250 uses this trick to print two spaces when I is both less than 100 and less than 10, one space when I is less than 100, and no spaces when it is greater than 100. This organizes everything quite nicely.

It should be noted here that this is only one solution to the problem. Until quite recently, I would have used string concatenation, which looks something like this:

```
222 I$=VAL(I)
223 IF LEN(I$)<3 THEN I$=" "+I$:GOTO 223
250 PRINT I$;"...                          (the rest is the same)
```

There's nothing wrong with the string approach except that it uses up more memory and probably more time, neither of which is very important here. The major advantage of the Boolean approach is that it is elegant. Professional programmers often talk about elegance in programs and while it's hard to define an elegant algorithm, clearly a major consideration is economy of both time and memory balanced with clarity. The Boolean solution, in this case, provides both.

If you want to try out your subroutine, add the following lines:

```
20 O1=227:O2=166:SCRN=1024:CRAM=55296
30 CHAR=14336:G1=SCRN+O1:C1=CRAM+O1:G2=SCRN+O2:C2=CRAM+O2
90 GOTO 1000

100 POKE 56334,PEEK(56334) AND 254
110 POKE 1,PEEK(1) AND 251
120 FOR I = 0 TO 2047:POKE CHAR+I,PEEK(53248+I):NEXT I
130 POKE 1,PEEK(1) OR 4
140 POKE 56334,PEEK(56334) OR 1
150 POKE 53272,(PEEK(53272) AND 240) OR 14

190 RETURN

1000 PRINT "<SHIFT CLR/HOME>";
1010 IF PEEK(14336)<>60 THEN GOSUB 100
1020 GOSUB 200
```

You'll note that lines 100 through 150 are identical to those found in the same listing in Chapter 6. This might be a good time to try out the merge process described in Appendix A.

The variables that haven't yet been described will be shortly—they deal with the rest of the screen. Note that the screen location has been made a variable (SCRN) so that there is enough flexibility to allow relocation of the screen or the character set without too much angst. People new to programming often use too many in-line constants, but it's better to make values that might change variables.

Besides, Commodore Basic executes faster when numerics have been defined this way; this is because numbers in program lines must be coded and decoded several times, while variables have already been coded and are ready to use. However, for this program time considerations are purely academic—such choices are more a matter of programming style.

Incidentally, line 1010 is there to prevent the lengthy transfer from ROM to RAM from occurring. The check I've done is rather primitive—if the top row of character 0 (the @) has been edited, this test will fail. You might want to devise a more sophisticated test by checking several characters for changes rather than just one.

Now we have a very nice character display but the edit grid remains to be created. It should be large because individual pixels are too small to really see well. We'll use one character to represent one pixel, which will result in an eight by eight character grid. Since the editor, as it is being developed, will deal with characters in the lower end of the character set, the grid will be composed of characters from the upper set—the inverse set.

The following subroutine will add the edit grid to your display:

```
50 EC=128:TL=1134
60 CC=1

300 GR$="<27 CRSR RIGHT><CRTL 9><8 SHIFT W><CTRL 0>"
310 PRINT"<CLR/HOME><4 CRSR DOWN>"
320 FOR I = 1 TO 8:PRINT GR$:NEXT I
330 POKE C1,CC
340 POKE TL,32
390 RETURN

1040 GOSUB 300
```

This is a very straightforward subroutine. The edit grid itself is defined as GR$, which, in addition to the cursor positioning instructions, consists of eight open circles (the <SHIFT W>). GR$ is printed eight times. CC stands for character color and EC stands for edit character. We'll deal more with these shortly.

That's our screen. Later, we'll add a list of commands to the screen to provide help, but for now let's decide how the editor should work.

Editing Modes

There must be two modes: one to select the character to be edited and one to do the actual editing. Both activities can be done by providing one cursor for each mode. Because we are working within a program, we cannot use the normal Commodore 64 cursor.

In the first mode, the cursor will move in the large character grid; in the second mode, the cursor will move in the smaller edit grid. Since we know

where both grids start, it is easy to determine the cursor locations. But what kind of cursor to use? It must clearly indicate the character we are selecting, but must also allow us to see the character.

In pondering this design question, Commodore's interesting color selection method kept piquing my curiosity. Why not use color memory to make the cursor? In this way, all that is on the screen remains visible; only its color changes. Since color RAM perfectly mirrors screen RAM, there is no problem maintaining our position in both areas.

We'll need at least four colors: background color, normal text color, cursor color, and a plot color for the edit grid. I decided to use Commodore's default colors for the background and normal text color: dark blue for the background and light blue for the text (colors 6 and 14 respectively).

I chose white (color 1) for the cursor and yellow (color 7) for the plotting points. It's easy enough to change these if another aesthetic strikes your fancy, but be careful that your choices are clear. Some color combinations are less than ideal.

Now it is necessary to make it work in the program. To do this, we must know how the cursor will move. Since the Commodore 64 has cursor control keys, I chose to use them. They will function as they do when you are editing program lines—you'll have to shift to move the cursor left. If you find this arrangement awkward, customize the keys to your liking. I plan to change to I, J, K, and M for up, left, right, and down respectively.

The following subroutine will accomplish the cursor movements. This whole program section should seem familiar to you because there's a lot of the random ball program that we saw in Chapters 3 and 4 here.

```
40 SX=0:X1=7:X2=14:Y1=7:Y2=15:SY=0:BY=Y1:BX=X1: REM BOUNDARIE
60 CC=1:TC=14:BC=6:PC=7: REM COLORS
80 POKE 53280,PC:POKE 53281,BC:POKE 646,TC

400 GET A$:IF A$="" THEN 400
410 IF A$="<CRSR UP>" THEN Y=Y-1
412 IF A$="<CRSR DOWN>" THEN Y=Y+1
414 IF A$="<CRSR LEFT>" THEN X=X-1
416 IF A$="<CRSR RIGHT>" THEN X=X+1
420 IF X<SX THEN X=BX
422 IF X>BX THEN X=SX
424 IF Y<SY THEN Y=BY
426 IF Y>BY THEN Y=SY
430 POKE (C2+X)+40*Y,CC:POKE (C2+OX)+40*OY,TC
490 OX=X:OY=Y
495 GOTO 400
```

```
1030 BY=Y2:BX=X2:X=SX:Y=SY:OX=X:OY=Y
1040 GOSUB 300
1050 GOSUB 400
```

Try it out. There's no editing here, but you should be able to move around the large character grid. Notice especially how I've used variables: SX and BX (for small X and big X respectively) to check for out-of-bounds. In this way, I can use the same loop for both the character and the edit grids by changing the appropriate variables. The boundary variables contain a 2 if they refer to the large character grid and a 1 if they refer to the small edit grid.

Now we need a way to tell the program that we want to edit the character that we've selected. Since there is a set of underused function keys on the computer, we'll use them for major commands: EDIT/FORGET, KEEP CHAR, SAVE SET, and LOAD SET. Four commands; four keys.

I'm bad with shifted function keys because I always forget whether to shift or not. So we'll relegate the other editor commands to the regular keyboard. The other functions will be insert, delete, and point plot. Since you're involved in the design of the editor, you can implement whatever additional commands you find you need.

It's not hard to add the additional commands to the command section of the program—just add the proper IF statements. Implementing the commands is another matter.

Program Commands

The first problem is getting the current character—the one highlighted in the character grid—onto the edit grid. We must find some way to translate the bits that represent each dot in the chosen character into the individual characters of the edit grid.

When a bit is on (or is a 1) the appropriate character of the edit grid should change to yellow (or whatever color the PC—plot color—is). Wherever a bit is off (or is a 0) the character in the edit grid should stay in the character color (CC). It's not hard to find the character that we have chosen; it's on the screen so all we have to do is PEEK it.

Getting the eight bytes that represent the screen design of the character is no problem, either. Since each character is eight bytes long, our chosen character is 8 times the character code into the character set. Fetching the next seven bytes will get the whole eight by eight character design. Sometime

after the first fetch and before the next fetch, we should decode the byte into individual bits and somehow get those that are on onto the edit grid in yellow. Here's how:

```
500 C=PEEK((G2+X)+40*Y):REM GET CHARACTER CODE FROM SCREEN
510 LOC=C1:REM START OF EDIT GRID
520 FOR I = 0 TO 7
530 T=PEEK(CHAR+(C*8+I)):REM GET BYTE (ROW) OF CHARACTER
550 FOR J= 7 TO 0 STEP -1
560 IF T=>2^J THEN POKE LOC+(7-J),PC:T=T-2^J:C(7-J,I)=PC
570 NEXT J
580 LOC=LOC+40:REM GO TO NEXT LINE OF EDIT GRID
590 NEXT I
595 RETURN
```

And, of course, add the command to the main command loop:

```
405 IF A$="<F1>" THEN RETURN
```

And add the necessary initialization to the main program:

```
1060 GOSUB 500
1070 BY=Y1:BX=X1:BY=Y2:X=SX:Y=SY:OX=X:OY=Y
1080 GOSUB 400
```

Since we're using the main loop for both the edit grid and the character grid, we need some way to differentiate when we are editing a character in the former and when we are selecting a character from the latter. A new variable, E, will provide us with this information. E will be a Boolean variable with a value of -1 when we are editing and 0 when we are selecting a character. Several lines must be updated:

```
430 IF NOT E THEN POKE (C2+X)+40*Y,CC:POKE (C2+OX)+40*OY,TC:
    CX=X:CY=Y:GOTO 490

1030 E=0:BY=Y2:BX=X2:X=SX:Y=SY:OX=X:OY=Y
1070 E=-1:BY=Y1:BX=X1:BY=Y2:X=SX:Y=SY:OX=X:OY=Y
1080 GOSUB 400
1090 GOSUB 300
1100 E=0:BY=Y2:BX=X2:X=CX:Y=CY:OX=X:OY=Y
1110 GOTO 1040
```

I slipped in a new variable here. To be fair, this didn't occur to me until the whole program was working but the cursor in the character grid was returning to character zero after each edit. This seemed wrong. After all, it's not uncommon to be editing a group of characters that are bunched together,

and having to move back down from character zero is unacceptable to me.

So two new variables were born—CX and CY—for character X and character Y. These variables get assigned only when one is not editing (thanks to the revision of line 430 above). After pressing F1 for edit mode, Line 1100 assures that these variables retain their values and allow us to return to the current character. It's tough to be elegant in Basic. But Basic does get the job done.

The Byte Decoder

The byte decoder is elegant. Since each bit in a byte represents another power of two, we can check each bit by going backwards from 255. We want to check each bit, so it will be necessary to subtract the appropriate power of two for any bit we find on (or set to 1). The loop at lines 550 to 570 is the key to the whole thing. It bears some study.

Let's imagine that T, the byte from the character set, is 128. In binary this is 10000000; in character design this represents a single dot in the upper lefthand corner of the character. In the loop with J as 7, T is equal to or greater than 128 (7^2). Therefore, we POKE into the color RAM a yellow color and subtract 128 from T; 128 − 128 = 0. The IF statement fails on all other passes, leaving us with a single yellow character in the upper left corner.

Similarly with 192 (in binary 11000000), the first pass is as before except that 192 − 128 = 64. So with T as 64, we repeat the loop. At the IF statement T is greater than or equal to 64, and J is now 6^2, so another character becomes yellow. T, which is now 64, subtracted from J^2, which is now 2^6 or 64, becomes 0 and the IF statement fails on all additional passes.

You might think a mathematical genius devised this process, but it's just logic, clear design, and a lot of debugging. It didn't work the first time.

Try the loop with other numbers. Hand simulate what the computer is going through—and you'll see that the algorithm works. You may understand how I isolated the bits but maybe the POKE LOC+ (7-J) is not clear. Here again logic must prevail.

The first working program produced some interesting backwards characters, which looked okay but had to be edited backwards. It never occurred to me that I was displaying the characters backwards. By this I mean that J is looping with a step of −1; just POKEing LOC+J produced strange results. We want the rightmost bit—the 128th bit (10000000)—to be at LOC, not at LOC+J! So subtract J from 7 to get the bit position.

It's complicated, but it is also logical and reasonable.

So far we've learned how to select a character from the character grid and move the cursor around the edit grid, but not much more. We need to add the following editing commands to our command loop: plot a point, continuous plot, and continuous delete. For these commands I've chosen the period, I, and D keys. If you'd rather have another set of keys—change 'em!

```
406 IF A$="I" THEN P=-1:D=0
407 IF A$="D" THEN P=0:D=-1
408 IF A$="." THEN P=0:D=0:P2=-1

440 IF P OR P2 THEN GOSUB 700
450 IF D THEN GOSUB 800
```

Note that P, D, and P2 are Boolean values. Each of the commands—plot a point, plot continuous points, and erase continuous points—should be unto itself. That is, when in plot a point, erase must be turned off. The three flags take care of this: when plotting is on, P has a value of − 1 (true) but when we switch to erase or point plotting mode, P is set equal to 0.

Now we must implement the individual commands in edit mode.

Insert and Point Plot

Since insert and plot point are really the same function, we'll construct both at the same time. These commands allow us to add dots to a character. As our cursor moves around the edit grid, it is really moving around the bits of the eight bytes that comprise the character being edited. We know which byte we're on—it's the same as the Y coordinate of the edit grid cursor. Similarly, the bit we are plotting is identical to the X coordinate of the edit grid cursor.

To turn the dot on, POKE the new byte value into the correct byte of the character set. We know what character is being edited—the subroutine at line 500 has assigned it to the variable C. Armed with this information, our insert subroutine looks like this:

```
700 REM
710 Z=PEEK(CHAR+(C*8+Y)):Q=(2^(7-X))
720 POKE CHAR+(C*8+Y),Z OR Q
730 P2=0
790 RETURN
```

The basic algorithm at lines 710 and 720 is the usual one to turn a bit on without changing the rest of the byte. The variables Q and Z, purely arbitrary names, are temporary variables to hold the values required. Z holds the current value of the byte and Q the bit value.

Line 730 provides the difference between insert and point plot commands. Point plot is good for only one bit whereas insert plots continuously. Line 730 turns off plotting by restoring P2 to zero.

Delete Mode

Delete mode turns out to be quite similar to plot mode. The only difference is the type of operation we need to perform on the byte. Since we are removing a bit, we want to AND the offending bit with a zero while not disturbing the rest of the byte.

This subroutine should look familiar to you:

```
800 REM
810 Z=PEEK(CHAR+(C*8+Y)):Q=(2^(7-X))
820 POKE CHAR+(C*8+Y),Z AND Z-Q
895 RETURN
```

Undo

All of this sounds fine, but there is a disadvantage to these subroutines. Since they alter the actual bytes in the current character set, you have no chance to change your mind and abort the edit. Instead, I'd rather edit a temporary character, and when I am happy with the edit, change the actual character when leaving edit mode.

We could select any character to be the temporary edit character. But, since this editor, as written, edits only the lower 128 characters of a character set, let's use character 128. Character 128 is the first character of the inverse set—the inverse commercial "at" sign. Further, since we might want to adapt the editor to edit different portions of the character set, we might want to change this character. So we'll make it a variable (EC).

If we do this, we will not see the effects of our edits, because character 128 is not displayed in our character grid. So we'll print it at a temporary location (TL).

The following changes will accomplish this:

```
50 TL=1134:EC=128

340 POKE TL,32:REM CLEAR TEMPORARY EDIT CHARACTER
480 POKE TL,EC:REM DISPLAY CURRENT EDIT IN LIFE SIZE

700 REM
710 Z=PEEK(CHAR+(EC*8+Y)):Q=(2^(7-X))
720 POKE CHAR+(EC*8+Y),Z OR Q
730 P2=0
790 RETURN

800 REM
810 Z=PEEK(CHAR+(EC*8+Y)):Q=(2^(7-X))
820 POKE CHAR+(EC*8+Y),Z AND Z-Q
895 RETURN
```

We need to add a line to the display subroutine to copy the character we are editing to our temporary locations. Since the subroutine (at line 500) is already pulling apart the character, byte by byte, all we need is a simple POKE:

```
540 POKE CHAR+(EC*8+I),T
```

And let's see what it looks like:

```
592 POKE TL,EC
```

Finishing Up

Our editor is almost complete but there's one more problem. Nothing happens in the edit grid when you are editing!

The following lines will help:

```
470 POKE (C1+X)+40*Y,CC:POKE (C1+OX)+40*OY,TC
480 POKE TL,EC
```

Line 470 does the work by putting the cursor color (CC) into color RAM at the current X and Y and then erasing the old cursor position by POKEing the text color (TC) into the old X and Y locations. But wait a minute! Do we always want the edit dot to return to the text color? No. Only if there is no pixel in that position. If there is a pixel, it should turn yellow (or whatever color is stored in PC). Time to revise again!

```
470 POKE (C1+X)+40*Y,CC:POKE (C1+OX)+40*OY,C(OX,OY)
```

Rather than try to find a clever relationship among various colors, it is easier and clearer to create a small array (C( )). This array will hold the color of the various dots on the edit grid. We must initialize the array (in the subroutine at line 300) and update it everytime we plot or erase a pixel on the grid:

```
350 FOR I = 0 TO 7:FOR J = 0 TO 7:C(I,J)=TC:NEXT:NEXT
360 FOR I = 0 TO 7:POKE (CHAR+(EC*8+I)),0:NEXT

700 C(X,Y)=PC:REM MAKE COLOR PLOTTING COLOR

820 IF C(X,Y)=PC THEN POKE CHAR+(EC*8+Y),Z AND Z-Q
830 C(X,Y)=TC
```

The editor is now almost completely functional—all that remains is a routine to move the newly edited character from its temporary home in EC to its final home in C:

```
409 IF A$="<F3>" AND E THEN E = NOT E:GOSUB 600:NEXT

600 FOR I = 0 TO 7
610 T=PEEK(CHAR+(EC*8+I)):POKE CHAR+(C*8+I),T
620 NEXT
630 RETURN
```

The new command at line 409 gives us a way to abort an edit. Line 405 accepts the F1 function key and returns to the main program at 1090. This effectively aborts the edit. To save the edited character, the F3 key must be pressed. Like F1, this will cause a return to the main program but will GOSUB to our new subroutine at line 600 to transfer the character.

The editor is completely functional but there is no way to permanently save your work to disk or cassette. We'll add these functions in a subsequent chapter. It would also be convenient to have instructions on the screen. In the full listing, which follows, you'll recognize the text positioning algorithm from Chapter 3.

```
5 REM CHARACTER SET EDITOR
10 POKE 52,56:POKE 56,56:CLR
20 O1=227:O2=166:SCRN=1024:CRAM=55296
30 CHAR=14336:G1=SCRN+O1:C1=CRAM+O1:G2=SCRN+O2:C2=CRAM+O2
40 SX=0:X1=7:X2=14:Y1=7:Y2=15:SY=0:BY=Y1:BX=X1
50 TL=1134:EC=128
60 CC=1:TC=14:BC=6:PC=7: REM COLORS
80 POKE 53280,PC:POKE 53281,BC:POKE 646,TC
90 GOTO 1000
100 POKE 56334,PEEK(56334) AND 254
110 POKE 1,PEEK(1) AND 251
```

```
120 FOR I=Ø TO 2047:POKE CHAR+I,PEEK(53248+I):NEXT
130 POKE 1,PEEK(1) OR 4
140 POKE 56334,PEEK(56334) OR 1
150 POKE 53272,(PEEK(53272) AND 240) OR 14
190 RETURN
200 PRINT:PRINT:PRINT"<6 SPACE>Ø 1 2 3 4 5 6 7"
205 PRINT "      <CTRL 9><C=64 A><15 SHIFT C><C=64 S><CTRL Ø>";
210 L=G2-24
220 FOR I = Ø TO 127
230 IF I/8 <> INT(I/8) THEN 260
235 PRINT
240 L=L+24
250 PRINT SPC(-1*(I<100)-(I<10));I;"<CTRL 9><SHIFT B>
    <CTRL Ø><15 SPACE><CTRL 9><SHIFT B><CTRL Ø>";
260 POKE L,I
270 L=L+2
280 NEXT I
285 PRINT
290 PRINT "<5 SPACE><CTRL 9><C=64 Z><15 SHIFT C><C=64 X>
    <CTRL Ø>";
295 RETURN
300 GR$="<27 CRSR RIGHT><CRTL 9><8 SHIFT W><CTRL Ø>"
310 PRINT "<CLR/HOME><4 CRSR DOWN>"
320 FOR I = 1 TO 8:PRINT GR$:NEXT I
330 POKE C1,CC
340 POKE TL,32
350 FOR I = Ø TO 7:FOR J= Ø TO 7:C(I,J)=TC:NEXT:NEXT
360 FOR I = Ø TO 7:POKE (CHAR+(EC*8+I)),Ø:NEXT
390 RETURN
400 GET A$:IF A$="" THEN 400
405 IF A$="<F1>" THEN RETURN
406 IF A$="I" THEN P=-1:D=Ø
407 IF A$="D" THEN P=Ø:D=-1
408 IF A$="." THEN P=Ø:D=Ø:P2=-1
409 IF A$="<F3>" AND E THEN E = NOT E: GOSUB 6ØØ:RETURN
410 IF A$="<CRSR UP>" THEN Y=Y-1
412 IF A$="<CRSR DOWN>" THEN Y=Y+1
414 IF A$="<CRSR LEFT>" THEN X=X-1+(E=Ø)
416 IF A$="<CRSR RIGHT>" THEN X=X+1+(-1*(E=Ø))
420 IF X<SX THEN X=BX
422 IF X>BX THEN X=SX
424 IF Y<SY THEN Y=BY
426 IF Y>BY THEN Y=SY
430 IF NOT E THEN POKE (C2+X)+4Ø*Y,CC:POKE (C2+OX)+4Ø*OY,TC:
    CX=X:CY=Y:GOTO 49Ø
440 IF P OR P2 THEN GOSUB 7ØØ
450 IF D THEN GOSUB 8ØØ
470 POKE (C1+X)+4Ø*Y,CC:POKE (C1+OX)+4Ø*OY,C(OX,OY)
480 POKE TL,EC
490 OX=X:OY=Y
495 GOTO 4ØØ
500 C=PEEK((G2+X)+4Ø*Y):REM GET CHARACTER CODE FROM SCREEN
```

```
510 LOC=C1:REM START OF EDIT GRID
520 FOR I = Ø TO 7
530 T=PEEK(CHAR+(C*8+I)):REM GET BYTE (ROW) OF CHARACTER
540 POKE CHAR+(EC*8+I),T
550 FOR J = 7 TO Ø STEP -1
560 IF T=>2^J THEN POKE LOC+(7-J),PC:T=T-2^J:C(7-J,I)=PC
570 NEXT J
580 LOC=LOC+4Ø:REM GO TO NEXT LINE OF EDIT GRID
590 NEXT I
592 POKE TL,EC
595 RETURN
600 FOR I = Ø TO 7
610 T=PEEK(CHAR+(EC*8+I)):POKE CHAR+(C*8+I),T
620 NEXT
630 RETURN
700 C(X,Y)=PC:REM MAKE COLOR PLOTTING COLOR
710 Z=PEEK(CHAR+(EC*8+Y)):Q=(2^(7-X))
720 POKE CHAR+(EC*8+Y),Z OR Q
730 P2=Ø
790 RETURN
800 REM
810 Z=PEEK(CHAR+(EC*8+Y)):Q=(2^(7-X))
820 IF C(X,Y)=PC THEN POKE CHAR+(EC*8+Y),Z AND Z-Q
830 C(X,Y)=TC
895 RETURN
900 H=23:V=17:GOSUB 99Ø:PRINT "F1 - EDIT/FORGET"
910 V=18:GOSUB 99Ø:PRINT "F3 - KEEP CHAR"
920 V=19:GOSUB 99Ø:PRINT "F5 - LOAD"
930 V=2Ø:GOSUB 99Ø:PRINT "F7 - SAVE"
940 H=4:V=22:GOSUB 99Ø:PRINT "<CTRL 9>I<CTRL Ø>NSERT<4
    SPACE><CTRL 9>D<CTRL Ø>ELETE<4 SPACE><CTRL 9>.<CTRL Ø>
    POINT PLOT"
950 V=23:GOSUB 99Ø:PRINT "<CTRL 9>CRSR<CTRL Ø> MOVE CURSOR";
980 RETURN
990 PRINT "<CLR/HOME>";:FOR I = 1 TO V:PRINT"<CRSR DOWN>";:
    NEXT:FOR I = 1 TO H:PRINT "<CRSR RIGHT>";:NEXT
995 RETURN
1000 PRINT "<SHIFT CLR/HOME>";
1010 IF PEEK(14336)<>60 THEN GOSUB 1ØØ
1020 GOSUB 2ØØ:GOSUB 9ØØ
1030 E=Ø:BY=Y2:BX=X2:X=SX:Y=SY:OX=X:OY=Y
1040 GOSUB 3ØØ
1050 GOSUB 4ØØ
1060 GOSUB 5ØØ
1070 E=-1:BY=Y1:BX=X1:X=SX:Y=SY:OX=X:OY=Y
1080 GOSUB 4ØØ
1090 GOSUB 3ØØ
1100 E=Ø:BY=Y2:BX=X2:X=CX:Y=CY:OX=X:OY=Y
1110 GOTO 1Ø4Ø
```

Finally, note the POKEs in line 10. If it's beginning to sound like these things are never done, you're getting into the swing of things. We are storing our character designs in the same memory area as Basic is using for itself, so we need a way to protect the bit maps from Basic's good intentions.

This statement lowers the top of string storage (location 52) and the top of Basic's memory (location 56) to 6510 page 56 or location 14336 (56 * 256 = 14336). Following this, CLR clears all existing variables.

In this way Basic will not attempt to store its data or programs in our character designs. But your program workspace has been reduced from the usual 39,000 to about 8,000 bytes. This program fits in the allotted space easily, but if you start to modify it and add functions, it may overflow the memory allotted to it.

There Are Sprites at the Bottom of My Garden

Character animation is useful for many purposes. It is conceptually simple and, on the Commodore 64, easy to do. But there are problems creating character animations. As we saw from the fish example in Chapter 5, large figures are slow. Once your design exceeds the eight by eight grid of a character, you begin to acquire overhead that slows your animation. Furthermore, if your animation is moving over a design, the figure destroys the background as it moves. Have hope, there's a solution: sprites.

Sprites are animated blocks built into the video display hardware of several personal computers. (On the Atari, they're called player missiles.) They differ from the normal characters or graphics produced by the computer in that they are created separately and then mixed into the video signal before it is sent out of the computer.

In other words, sprites are superimposed on the screen, not a part of it. This solves the problem of animations destroying the backgrounds. The size/speed problem is also eased by sprites because they are generated by the VIC II chip, not by the main microprocessor. The VIC II chip creates the display screen only, so it can handle good-sized sprites—eight at a time—without slowing down appreciably. Unfortunately, if you have many complex moves in an animation, the sprite will still move slowly. The problem is not the VIC II chip or the sprite but rather the overhead of Basic as it computes the movement.

Designing a Sprite

Sprites, like characters, require a **bit-mapped** or dot by dot image within the current 16K VIC II address space. If your work calls for a custom character set and all eight sprites, that space will be crowded.

Sprites are 24 dots horizontally (three bytes worth of pixels) and 21 dots or lines vertically ($24 \times 21 = 504$ dots; $3 \times 21 = 63$ bytes). If a bit within this area is a 1, the VIC II displays a dot in the current sprite color at the current sprite location. If a bit is a zero, that area of the sprite is displayed as transparent; that is, with is the background showing through.

To design a sprite, draw the figure on graph paper using a 24×21 grid. Design in hand, each row must be split into bytes or eight dot chunks. Each byte-worth of dots, then, is converted to decimal. The resulting 63 bytes can then be POKEd into memory.

We'll go through the process for a sample sprite—a space ship. The first ten rows of the design is shown in Fig. 8.1. The rest of the sprite is all zeroes and so is transparent. As drawn here, it looks a little like a top. But remember that there is an aspect ratio to consider. In a design that is 320 horizontal pixels by 200 vertical pixels, each pixel is about one-third taller than it is wide. This will stretch the design horizontally and achieve the desired effect. You'll have trouble drawing a circle if you make it exactly symmetrical on graph paper. Instead make it a little wider than it is tall. On the screen, it will look like a circle.

Figure 8.1
Space Ship Sprite Design

Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø	Ø	1	1	1	1	1	1	1	1	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø	Ø	1	1	1	1	Ø	1	1	Ø	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø	1	1	Ø	1	1	Ø	1	1	Ø	1	1	Ø	1	1	Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	1	1	1	1	Ø	1	1	Ø	1	1	Ø	1	1	Ø	1	1	1	1	Ø	Ø	Ø
Ø	Ø	Ø	Ø	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø	Ø	1	1	1	1	1	1	1	1	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	1	1	1	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

Then we must split the design into bytes and convert to decimals (see Fig. 8.2). If you've forgotten how to convert binary to decimal, refer to Chapter 2.

Figure 8.2
Space Ship Design Separated into Bytes

00000000	00111100	00000000	(0)	(60)	(0)
00000101	11111111	11000000	(3)	(255)	(192)
00000101	11011011	11000000	(3)	(219)	(192)
00000110	11011011	01100000	(6)	(219)	(96)
00011110	11011011	01111000	(30)	(219)	(120)
00001111	11111111	11110000	(15)	(255)	(240)
00000011	11111111	11000000	(3)	(255)	(192)
00000000	11111111	00000000	(0)	(255)	(0)
00000000	00111100	00000000	(0)	(60)	(0)
00000000	00011000	00000000	(0)	(24)	(0)

Now we have designed a sprite. Since our sprite is only ten lines long, fill the rest of the definition with zeroes. A program fragment to use these looks like this:

```
1000 LOC=832
1010 FOR I = LOC TO LOC+29:READ BYTE:POKE I,BYTE:NEXT I
1020 FOR I = LOC+30 TO LOC+63:POKE I,0:NEXT I
1090 RETURN
1100 DATA 0,60,0,3,255,192,3,219,192
1110 DATA 6,219,96,30,219,120,15,255,240
1120 DATA 3,255,192,0,255,0,0,60,0
1130 DATA 0,24,0
```

These lines define our sprite and POKE it into locations 832 and following. 832 is near the start of the cassette buffer. If you're using the Commodore cassette recorder, the sprite definition will be destroyed everytime you use the cassette. But our program doesn't use the cassette so this won't be a problem. Since 192 bytes of the buffer are usable for sprites, it's a handy place to tuck three sprites (192/64 = 3).

Pages

Why are only 192 bytes of the buffer usable? The answer lies in how we have to tell the VIC II where our sprites are. Commodore engineers allotted a byte for this information (very generous in comparison to the character sets!). One byte can hold any number up to 255, it is used to point to the "page" where the VIC II can find the sprite design data.

Since sprites use the VIC II chip, we are still limited to the VIC II's 16K space. 16K is 16,384 in decimal—divided by 64 bytes (the space needed to design a sprite) equals 256, or 255 if you're counting from zero as Commodore does. Does the logic become clear?

The 16K space of the VIC II, then, can be thought of as being divided into pages of 64 bytes. Each page can hold the design for one sprite. We need only POKE the page number into the correct sprite data pointer area and the VIC II can find the sprite design. This also limits where the sprites can be, because they must be placed where the starting location is evenly divisible by 64.832 is such a place. The cassette buffer starts at 828 so the first four bytes are not usable a sprite page.

Be careful here to keep the notion of "page" straight. Generally, a page for an eight-bit microprocessor (like the 6510) is 256 bytes long and there are 256 of them in the 64K address space (256 × 256 = 65,536). Most of the graphics for the Commodore 64 use page in reference to the VIC II chip, which has only a 16K space.

Now we need to POKE the sprite page into the correct location. Commodore took advantage of some unused space for these locations. Since the screen is only 1,000 characters long (25 × 40 = 1,000) and Basic doesn't start storing its program until the next even 6510 page number (location 2048), there are 24 free bytes that would not normally be used. These start at location 2024. The sprite pointers start at 2040 and go to 2047 . . . room for eight. Add these lines to your program:

```
1000 LOC=832:PAGE=LOC/64
1030 POKE 2040,PAGE
```

There are two more chores before the sprite can be displayed. We need to tell the VIC II first that we want to see sprite 0 and then where to put it on the screen.

Only one location controls which sprites the VIC II will display. Each bit of the one byte location is used. Turning bit zero on (that is, making it a one) will tell the VIC II to show sprite 0. When no sprites are being displayed, the location holds a 0. By manipulating this byte we can turn sprites on and off. We'll find out later that this is the key to multiframe sprite animation, but for now, let's just turn on sprite 0.

```
1040 POKE 53269,PEEK(53269) OR 1
```

The formula should look familiar to you, as we've already used it. The more general formula for sprites should be equally familiar:

```
POKE 53269,PEEK(53269) OR (2^N)
```

In this line, N represents the sprite we wish to see. We used the same Boolean trick to get our characters edited and we'll use it again when learning about high resolution graphics in Chapter 11.

The VIC II now knows to display our sprite, but we must tell it where to put our space ship. This will do it:

```
5 REM SPRITE DEMO 1
10 PRINT "<SHIFT CLR/HOME>"
20 GOSUB 1000
30 POKE 53248,100: REM SPRITE 0 X LOCATION
40 POKE 53249,100: REM SPRITE 0 Y LOCATION
990 END
```

When you run this program, you'll see a white top-like space ship on the screen. List the program—the sprite stays right on top of your program. Recall that the VIC II is now mixing two video planes: one from the standard text screen and one of the sprite planes. When I first worked with sprites several years ago on the Texas Instruments 99/4A computer, I was astonished by their power. And they're really quite easy to use, especially with the tools we're going to assemble.

To get rid of your sprite, you'll have to press <RUN/STOP RESTORE> or POKE 53269,0. Instead of doing this all the time, add these lines to your program:

```
890 STOP
900 POKE 53269,0
```

In this way, the program will stop with the sprite displayed. To make it disappear type CONT. Any special character sets will remain attached to the VIC II. When you press <RUN/STOP RESTORE>, the VIC II resets itself and any custom character sets will be lost.

Positioning Your Sprite

Three one-byte registers control each sprite's location. The horizontal coordinate is determined by location 53248 and the vertical coordinate by

location 53249. (The other sprites continue upward in memory in the same way.) We can use all 255 possible values that can fit in a byte. The sprite will move one dot each time a location register is changed. Try changing your program as follows:

```
40 FOR I = 0 TO 255
50 POKE 53279,I
60 NEXT I
```

You'll see a swiftly descending space ship. Crisp and flicker-free. If you were observant, you noticed that the ship doesn't appear right away and it is a short while before the BREAK IN 890 message appears. Why?

It is appropriate, for most animations, to have the sprite slide off the screen completely before it reappears on the other side of the screen. With traditional text screen animation, this capability must be programmed. (Our fish was split between the two sides of the screen for a few moves.) The sprite planes are separate from the regular video plane and are larger than the normal Commodore display screen. Therefore, if a sprite is too far to the left, right, top, or bottom, we won't see it. Further, you can gain more off-screen sprite room by changing the screen size from the normal 25 by 40 to 24 by 38.

Sprites are always positioned from the upper left-hand dot of the sprite design, even if this is a transparent dot, as in the case of our space ship. The space ship doesn't appear until I reaches 30 and starts to disappear when I is 249. If you want to confirm this, add this line to your program:

```
55 PRINT "<CLR/HOME>";I:FOR D = 1 TO 50:NEXT D
```

Something similar happens in the X direction. Edit your program as follows:

```
30 POKE 53249,100
50 POKE 53248,I
```

Now run this program. Strange. The beginning was just as expected, except that the sprite appeared sooner than it did in the Y direction. But it's still on the screen—about three-fourths of the way across. The problem with moving in the X direction is that there are over 255 visible positions. Commodore did deal with this, but, to my way of thinking, in an unsatisfactory way. However, we are stuck with it.

To move a sprite beyond position 255, we need to use an additional memory location. Rather than devoting a whole location to the additional X positions, Commodore engineers chose to be clever. The location 53264 controls whether the X location is to be interpreted as starting at screen position zero or at screen position 256. If the proper bit is set, the X location starts at 256 rather than zero. Add the following lines to your program:

```
70 POKE 53248,0
80 POKE 53264,PEEK(53264) OR 1
90 FOR I = 0 TO 63
95 PRINT "<CLR/HOME>";I;" ":FOR D = 1 to 50:NEXT D
100 POKE 53248,I
110 NEXT I
910 POKE 53264,0
```

This technique is no problem with one sprite, but keeping track of eight sprites is more difficult. It's just one more thing to slow down the sprite movement from Basic. Of course moving the sprites from assembly language is much quicker, but assembler is much trickier to use than Basic.

Diagonal movement, of course, requires both registers be changed:

```
50 POKE 53248,I:POKE 53249,I
```

With diagonal movement, it becomes particularly troublesome to determine when to switch the X overflow bit. Only careful storyboarding will enable you to effectively accomplish this.

Colorful Sprites

The sprite we've been playing with is white but we can make it any of the Commodore 64's 16 colors. One POKE takes care of this:

```
55 POKE 53287,2
```

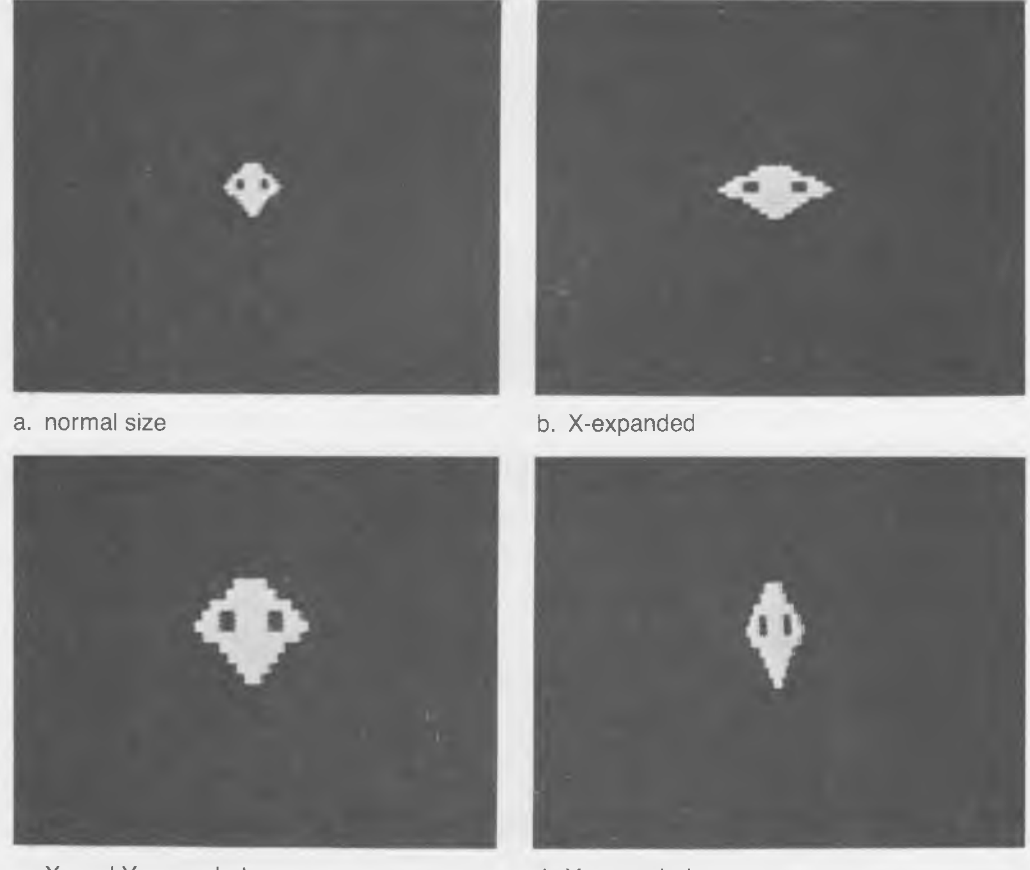
The number being POKEd is a Commodore 64 color (see Fig. 4.1) and the usual color effects are also enforced. A red sprite or character (color 2) on the standard blue (color 6) background is a bit fuzzy. For sprites, this combination can be used as an effect; for characters they are illegible. In this case the ship seems to stand out from the background a bit, a rather nice effect. The only way to be certain about how your background/sprite color

combinations will combine is to try them out. And be aware that the various colors look different on different types of CRTs and televisions.

Expanded Sprites

As if enabling smooth, colorful animation were not enough, sprites have another important capability. They can be expanded. On the Commodore 64, you have control over expansion in either the X direction or in the Y direction. This is a sublime capability.

Figure 8.3
Sprite Expansion



a. normal size

b. X-expanded

c. X- and Y-expanded

d. Y-expanded

The expansion of each sprite is controlled by two memory locations, one for X and one for Y. Location 53277 controls X expansion and location 53271 controls Y expansion. These locations are bit-oriented, similar to the sprite-enable location. Setting the proper bit in each location will expand the sprite in the expected direction.

To make our space ship a wider key:

```
POKE 53277,1
```

Since we only have one sprite, we needn't worry about the other bits. However if we had more than one sprite, we would have to preserve the previous contents of the location by using the usual algorithm:

```
PEEK 53277, PEEK(53277)OR(2^N)
```

Actually, I prefer this with some color flashes. Try the following:

```
45 POKE 53287,I
```

The image produced by the program now is close to what I had in mind when I designed the ship. But let's see what it looks like with the other axis expanded:

```
POKE 53271,1
```

Each pixel has been expanded in both the X and the Y directions. The ship now looks rather blocky. There's nothing you can do about this. Some designs look good expanded; others look clumsy. Experience will tell you how to design to get the effect you desire. We should look at the final permutation—expansion in only the Y direction:

```
POKE 53277,Ø
```

Expansion in the X direction is less distorting than expansion in the Y direction. This is due mostly to the aspect ratio of the Commodore 64 screen. However, there are times when Y expansion is necessary and the capability to do it is welcome.

The following data statements describe a car-shaped sprite that better illustrates the advantages of X and Y expansion. We'll replace the space ship in your current program:

```
1100 DATA 0,0,0,1,248,0,7,196
1110 DATA 7,199,128,63,255,192,127,255,192
1120 DATA 49,241,128,10,10,0,4,4,0
1130 DATA 0,0,0,0
```

Do try this! The appearance of the car can change from a tiny Volkswagen to a sleek racing model to a Checker by changing the expansions.

Expansion does affect the size of the off-screen locations. The larger the sprite, the fewer the off-screen dots. Fig. 8.4 details the size of the sprite planes with various expansions and screen modes.

Figure 8.4
Sprite Video Plane Sizes*

Mode	Expansion		Visible Positions*			
	X	Y	Min X	Min Y	Max X	Max Y
40 × 25	N	N	1	30	343	249
	Y	Y	489	9	343	249
38 × 24	N	N	8	34	334	245
	Y	Y	496	13	334	245

* Note that for X positions over 255, the actual value in the X position register must be X:255 and the proper bit in location 53264 must be set.

Sprite Priorities

When two sprites overlap on the screen, the VIC II has to make a decision about which sprite to display on top. The rule it uses is very simple, sprite zero has the highest priority and will cover any other sprite. Sprite one has next priority and so on. To demonstrate this, let's first get two sprites on the screen. We need not have two designs; instead we'll use identical bit images for both sprites. First, we must enable two sprites:

```
POKE 53269,3
```

Now, we'll set the pointers up:

```
POKE 2040,13
POKE 2041,13
```

Note that both POKEs indicate VIC page 13, location 832. You may or may not see anything at this point; we haven't specified the X and Y coordinates of the sprites.

```
POKE 53248,100:POKE 53249,100
POKE 53250,100:POKE 53251,120
```

You should be seeing two cars or space ships, depending on the bit image you last worked with. If you don't see this, rerun the sprite program, restoring the sprite designs to memory.

Now we'll superimpose the sprites:

```
POKE 53251,100
```

Sprite one should disappear; it's now underneath sprite zero. Let's expand sprite one to confirm the positioning:

```
POKE 53277,2
```

Suddenly, sprite one is visible behind sprite one. You might want to experiment with the other sprites. Sprite priority is very useful for complex animations so it's worth exploring.

The same decision is made about whether to display the sprites on top of the text screen or on the bottom. Normally, the sprites appear on top of the text screen, but they can be easily placed beneath. Scroll some text over your sprites and type:

```
POKE 53275,3
```

VIC II now places the text plane on top of both sprites. This location is another bit-determined location. Where the proper bit is set to zero, the sprite has priority; where the bit is set to one, the text has priority. To return the sprites to their normal priority, POKE a zero into 53275.

Multiple Frame Sprite Animation

Creating multiple frame sprite animation is almost as simple as creating single frame animation. In multiple frame animation, we must have two or more sprite designs—one for each frame of the animation.

The following program demonstrates how to accomplish multiple frame animation using sprites. The program is similar to the first sprite demonstration program; however, the subroutine at line 1000 has been rewritten slightly to allow it to use different locations for storing the sprite designs. LOC is now defined in the main program (lines 20 and 30).

After reading in the sprite designs and setting the correct pointers, both sprites are set to the same X and Y coordinates (lines 40 and 50). The last section of the main program is the animation loop (lines 60 to 110). This loop changes the sprite being displayed (location 53269) rather than the X and Y coordinates of the sprite.

At any specific instant, only one sprite is seen. The animation effect is achieved by alternating between the two designs rapidly. The alternation is accomplished by the even/odd algorithm we've used several times before.

```
5 REM SPRITE DEMO 2
10 PRINT "<SHIFT CLR/HOME>"
20 LOC=832:GOSUB 1000:POKE 2040,PAGE
30 LOC=896:GOSUB 1000:POKE 2041,PAGE:POKE 53288,1
40 POKE 53248,100:POKE 53249,100
50 POKE 53250,100:POKE 53251,100
60 FOR I = 1 TO 100
70 S=1
80 IF I/2=INT(I/2) THEN S=2
90 POKE 53269,S
100 FOR D = 1 TO 50:NEXT D
110 NEXT I
990 END
1000 PAGE=LOC/64
1010 FOR I = LOC TO LOC+26:READ BYTE:POKE I,BYTE:NEXT I
1020 FOR I = LOC+27 TO LOC+63:POKE I,0:NEXT I
1090 RETURN
1100 DATA 0,255,0,0,24,0,0,24,0
1110 DATA 12,63,192,15,127,224,7,255,224
1120 DATA 3,255,192,0,66,16,3,255,224
1200 DATA 0,0,0,0,24,0,0,24,0
1210 DATA 12,63,192,15,127,224,7,255,224
1220 DATA 3,255,192,0,66,16,3,255,224
```

By now you should have a good idea of how to control sprites (see Fig. 8.5 for a summary of all the important sprite locations).

Play with these PEEKs. If you haven't already noticed, the computer executes the instruction immediately. If a sprite is on the screen and you change the color byte, it changes color immediately. You can experiment with different designs and with different expansions without extensive programming. I've included additional patterns for the car and helicopter sprites, in Figs. 8.6 and 8.7. In the next chapter we'll build an editor to make designing these creatures easier.

Figure 8.5
Sprite Control Locations*

Sprite	Pointer	X Loc	Y Loc	Color	Bit Value*
0	2040	53248	53249	53287	1
1	2041	53250	53251	53288	2
2	2042	53252	53253	53289	4
3	2043	53254	53255	53290	8
4	2044	53256	53257	53291	16
5	2045	53258	53259	53292	32
6	2046	53260	53261	53293	64
7	2047	53262	53263	53294	128

* Number to POKE into single byte locations including: SPRITE ENABLE (53269), X OVER 255 (53264), Y EXPANSION (53271), X EXPANSION (53277) and SPRITE/BACKGROUND PRIORITY (53275). The preferable way to accomplish this is with the algorithm POKE xxx, PEEK (xxx) OR N, where N represents the sprite bit value and xxx represents the desired memory location.

Figure 8.6
Car Sprite Pattern

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

Figure 8.7
Helicopter Sprite Pattern

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

Building the Sprite Editor

If you paid attention in Chapter 7, you'll have no trouble here. The sprite editor that we build will be patterned after our character editor, so most of the work is already done. Recall the overall format:

```
10–99       program constants
100–999     subroutines including
        100–199  technical set up
        200–299  screen set up
        300–399  decode/restore for new design
        400–499  command loop
        500–599  decode as edit
        600–699  clear or fill edit square*
        900–999  instructions
1000–     main program
1500–     data*

* no analog in the character editor
```

There are so few sprites available that we won't waste one for test editing as we did for the character editor. Instead, the changes will affect the actual sprite. This decision eliminates the need for edit/abort and save character commands. It is simpler to select sprite by number. These changes make the whole command structure easier to work with because you don't have to switch repeatedly between a select character mode and an edit character mode.

Of course, the technical details of pulling apart the edit grid to find the value to POKE will differ from the character editor. The most important difference is that sprites are three bytes wide rather than one, like characters. Aside from this, the concept is identical. When you delete a pixel, flip the edit

grid dot to the off color and change the correct bit in the sprite from a 1 to a 0. When inserting pixels, flip the edit grid dot to the on color and change the correct bit in the sprite from a 0 to a 1.

We've already detailed the logic of the programming process in the character editor chapter, so now let's look at the sprite editor in subroutine segments.

```
10 DIM C(24,21)
20 O1=54:SCRN=1024:CRAM=55296
30 G1=SCRN+O1:C1=CRAM+O1
40 SX=0:BX=23:SY=0:BY=20
50 XS=31:YS=54
60 CC=1:TC=11:BC=15:PC=14
80 POKE 53280,0:POKE 53281,BC:POKE 646,TC
90 GOTO 1000
```

As we did for the character editor, we'll need to maintain an array of numbers representing the current edited state of the sprite. In this way, the screen can be updated correctly as the cursor moves over it. But here, the array must be dimensioned as a sprite design: 24 by 21 dots.

You've probably noticed that there are fewer constants here than in the character edit program. Because we had to maintain two screen grids—one for the character selection and one for the editing—there had to be an additional set of screen locations.

For the sprite editor, we need only the edit screen for both characters (SCRN) and color (CRAM); the VIC II chip takes care of displaying the sprites. Since the edit grid is positioned differently on the screen, the offset (O1) is different from the one in the sprite program. When added to SCRN or CRAM, O1 will assure that the cursor positioning will occur relative to the upper left-hand corner of the edit grid.

Similarly, the number of restraints on the X and Y cursor values are reduced by half. The smallest acceptable value for both X and Y is 0 (SX and SY). The largest acceptable value for X and Y are 23 and 20, respectively (BX and BY). Notice that these values are 1 less than the maximum sprite size—24 by 21—because we're counting from 0 rather than from 1 as we are accustomed to doing.

As in the character editor, we want the sprites to be shown as we are editing. To do this, XS and YS were created. These represent the X and Y offsets necessary to display a nonexpanded sprite at a given X and Y coordinate. These will be added to the actual X and Y coordinates later. The two-character limitation on variable names is a real challenge here; it's not so

easy to differentiate between SX—small X, and XS—sprite X, but with longer variable names ruled out, this seemed to be the best solution.

The next line (line 40) defines the color constants for the cursor, text character, background, and plot, respectively. These have been changed from the character editor to prevent boredom. The Commodore colors are so glorious that we might as well use them. I've selected these colors for visibility and contrast with the default sprite colors. You might prefer a different selection.

Finally, these colors are POKEd into their requisite locations. Note that the border (location 53280) is initially given a value of 0. This was my solution to the problem of telling which sprite you are editing. In the character editor, we used a second cursor. Here, because each sprite has a different default color, we could use the border color to indicate the sprite being edited.

When a sprite is selected for editing, the border will change to the color of that sprite. If you have changed the sprite colors that two sprites have the same color, you may have trouble telling which sprite you are editing. The use of the border color may not be the best solution for all uses of the editor. It might be interesting to add a color change option to the list of commands. If you do, you should also add a second cursor or another on-screen indication of which sprite you are editing.

Technical Set-Up

```
100 FOR I = 0 TO 7:READ SL(I):POKE 2040+I,SL(I):NEXT
110 FOR I = 53248 TO 53263 STEP 2
120 READ SX(J),SY(J):POKE I,SX(J)+XS:POKE I+1,SY(J)+YS
130 J=J+1:NEXT I
140 POKE 53269,255
190 RETURN

1500 DATA 248,249,250,251,252,253,254,255
1510 DATA 16,24,64,24,16,48,64,48
1520 DATA 16,72,64,72,16,96,64,96
```

The character editor had to PEEK the ROM character set and POKE it into RAM. There are no ROM sprites, so all we need to do here is POKE the sprite design pointers (location 2040 and following), then POKE the sprite X and Y locations into location 53248 and following, and then turn on the sprites by POKEing 255 into location 53269. No special tricks here except the STEP 2 used in the FOR/NEXT loop to accommodate the X,Y structure of the locations.

The locations chosen to stash the bit designs of the sprites are at the very top of the VIC II address limits. These addresses are good for our program. Should you add more capability to the editor, it is conceivable that you may run into memory conflicts. Your program or its variables can overwrite the sprites, or even worse, the sprites can overwrite your program or its variables. The latter is a nasty situation indeed.

The sprite positions are given as absolute values and read into an array. When these positions are POKEd into locations 2040 and following, offsets (XS and YS) are added to them. These offsets let the sprites appear where you want them to. This method may seem more complicated than necessary, but it allows for easy changes. If you want an editor that can show a double-sized sprite, you'll have to change only the offset values rather than all the positions.

Screen Set-Up

```
200 PRINT "<CLR/HOME>"
210 FOR I = 1 TO 21
220 PRINT "<14 CRSR RIGHT><24 SHIFT W>"
230 NEXT
290 RETURN
```

Much like the technical set-up, the screen set-up is simpler than that of the character editor. We need only display a grid of 24-by-21 dots.

Decode/Restore

```
300 S=VAL(A$)-1:POKE 53280,PEEK(53287+S)
310 L=C1:C=0:M=0:N=0
320 FOR I = SL(S)*64 TO SL(S)*64+62
330 T=PEEK(I)
340 FOR J = 7 TO 0 STEP -1
345 IF T=>2^J THEN CT=PC:T=T-(2^J):GOTO 355
350 CT=TC
355 POKE L+(7-J),CT:C(N,M)=CT:N=N+1
360 NEXT J
370 L=L+8:C=C+1:IF C/3=INT(C/3) THEN L=L+16:C=0:M=M+1:N=0
380 NEXT I
385 X=0:Y=0
390 RETURN
```

This subroutine is significantly different from the character editor, yet it uses an identical algorithm to decode the sprite into its edit screen representa-

tion. The difference is that a sprite is spread out over three horizontal bytes rather than just one.

First we must discover which sprite the user wants to work with. This is accomplished by line 300. A$ represents the user input. When a person types 1, for the first sprite, line 300 subtracts one from it to arrive at sprite 0. This makes more sense than starting with 0, given the way the keys are laid out on the Commodore 64 keyboard. Next, line 300 POKEs the color value into the background color register.

The color is obtained by PEEKing the value in the sprite color registers starting at 53287 and following. It would have been much cleaner just to POKE the sprite number + 1, but sprite number 7 has a color of medium grey (color 12) rather than orange (color 9). Furthermore, since you may be adjusting the colors of the sprites, it seems best to pick out the color from the register.

Line 310 sets the variables to be used in the subroutine. L holds the character position in the edit grid. C is a simple counter, which will be explained shortly. M and N are index variables to be used to update our array of colors. This is the subroutine that will set a specific cell of this array to either the plotting color or the text color.

The FOR/NEXT loop in lines 320 to 380 is the workhorse. SL( ) has been set to the sprite pointer. Recall that the actual address of the sprite design is at SL ( ) *64 from the start of the VIC II addresses. If you want to adapt this editor to be used with sprites in other memory areas, you'll need to add the VIC II offset so that the right area of memory is used. I, then, will hold the address of each byte of a specific sprite.

Line 330 PEEKs the value of the selected sprite byte.

The FOR/NEXT loop in lines 340 to 360 decodes the byte into individual bits using the algorithm we've already seen. If a bit is set, the IF statement in line 345 will be true. The temporary color (CT) will be set to the plotting color (PC) and T will be reduced by the correct power of two. Basic will then branch to line 355.

If the IF statement is false, the temporary color will be set to the text color (TC).

Line 355 POKEs the color into the edit grid and sets the appropriate cell of the C( ) array to that color. N is also incremented as we now must progress to the next bit of the byte.

After a whole byte has been decoded, Basic falls through the FOR J loop and executes line 370. First, L is incremented by a value of eight for a new byte. L, you'll remember, is the POKE location into the edit grid. C is also

incremented and is counting the horizontal bytes. Since there are three bytes to each horizontal line, C must be restricted to values between zero and two. C starts at one and increments each time line 270 is reached.

The IF statement in line 270 assures that when C equals three, it is returned to a value of one. L must also be incremented by a value of 16—the amount necessary to get from the rightmost character on the edit grid to the leftmost character on the next line. M is incremented since we have now moved to a new screen row, whereas N must be returned to zero. As these variables are now reinitialized, line 380 ends the outer loop.

It would be worthwhile for you to spend some time following the logic of this loop, as it is critical to your understanding of the program.

Before returning to the caller (the command loop), the X and Y coordinates of the cursor are set to zero so that each edit will start the same. If you'd rather start each edit with the cursor unmoved, remove line 385.

Command Loop

```
400 GET A$:IF A$="" THEN 400
402 IF A$="<F1>" THEN CT=0:GOSUB 600:P=0:D=0
405 IF A$=>"1" AND A$<"9" THEN GOSUB 300:P=0:D=0
406 IF A$="I" THEN P=-1:D=0
407 IF A$="D" THEN P=0:D=-1
408 IF A$="." THEN P=0:D=0:P2=-1
409 IF A$="<F3>" THEN CT=255:GOSUB 600:P=0:D=0
410 IF A$="<CRSR UP>" THEN Y=Y-1
412 IF A$="<CRSR DOWN>" THEN Y=Y+1
414 IF A$="<CRSR RIGHT>" THEN X=X+1
416 IF A$="<CRSR LEFT>" THEN X=X-1
420 IF X<SX THEN X=BX
422 IF X>BX THEN X=SX
424 IF Y<SX THEN Y=BY
426 IF Y>BY THEN Y=SY
440 IF P OR P2 THEN GOSUB 500:P2=0
450 IF D THEN GOSUB 500
470 POKE (C1+X)+40*Y,CC:POKE(C1+OX)+40*OY,C(OX,OY)
480 OX=X:OY=Y
490 GOTO 400
```

It is amazing how similar this command loop is to the loop in the character editor. In programming, there are usually many ways to accomplish a given task, and some are more efficient or clearer than others. Since we already had a working loop, it seemed foolish to invent another one because the tasks are so similar. Speed and efficiency could, of course, improve this program, but the command loop really is not where optimization is needed.

There are some changes. Since there is no separate mode to select a character, the variable E from the character edit is not needed here. In fact, as it stands, we don't return to the main program at lines 1000 and following until the save and load options have been added—but, for now, there is no return.

Line 402 is a new option, clear sprite. The characters were small, so it wasn't much of a chore to clean out a grid. Sprites are another story. Note that CT is set to zero before the subroutine call. Upon RETURN, the plotting and deleting variables are set back to zero.

The next line deals with sprite selection. If a number has been typed, we GOSUB to the routine at line 300, which reads and decodes the sprites and puts the correct dots on the edit grid. This routine also resets the plotting and deleting variables to zero.

The next three lines are identical to their counterparts in the character editor. They define the plotting and deleting commands.

Line 409 defines the reverse of the F1 command. When F3 is pressed in the sprite editor, the grid is filled; that is, all bits in the sprite are converted to 1s. The variable C takes on a value of 255 before branching to the subroutine at line 600. Plotting and deleting are turned off upon return.

The next eight lines are also identical to their counterparts in the character editor. These define cursor movement and assure that it is kept within legal bounds. Since the cursor positions are also used to POKE sprite dot values, it is very important to maintain the cursor position within legal limits.

The next two lines, 440 and 450, execute the plotting and deleting commands by branching to a subroutine at line 500. This program was written after the character editor. It occurred to me that there were great similarities in the plotting and deleting commands. In fact, they only differ in their Boolean function—an OR for plotting and an AND for deleting. Why not combine the separate subroutines?

Rather than rewriting the character editor, I thought it might be good to show the refinement process that programs go through. Programs are like any creative work—they continue to grow and evolve. The successful programmer, however, finally says "enough is enough" and releases his or her work.

The remainder of the subroutine is identical to the character editor, except that the latter had to update the temporary character that represents the character being edited. Here the sprites change as they are being edited.

Decode

```
500 B=(-1*(X>7))+(-1*(X>15))
510 IF X<8 THEN BIT=2^(7-(X AND 7)):GOTO 520
512 IF X<16 THEN BIT=2^(7-((X-8) AND 7)):GOTO 520
514 BIT=2^(7-((X-16) AND 7))
520 LOC=SL(S)*64+(Y*3+B)
530 IF P THEN 570
540 POKE LOC,PEEK(LOC) AND PEEK(LOC)-BIT
550 CT=TC
560 GOTO 590
570 POKE LOC,PEEK(LOC) OR BIT
580 CT=PC
590 C(X,Y)=CT
595 RETURN
```

The hard part is figuring out which byte you are on. We know only the X coordinate. Fortunately, it's not hard to figure out which X coordinates belong in which bytes. This is accomplished by line 500. B is set to 0, 1, or 2 depending on whether the X coordinate is smaller than 8, larger than 8 but smaller than 16, or larger than 16. Try it yourself by hand—substitute values for X and see which bytes they turn out to be. Remember that the maximum value for X is 23.

Knowing what byte we are in is most of the problem. It's a simple matter to multiply by the correct power of two, as done in the IF statements and single line at 510 to 514. The algorithm is tried and true; the only subtlety is subtracting the correct value from X so that it calculates the correct power of two.

That accomplished, we must locate our byte in sprite design memory. Unlike our artificial three byte by 21 line array on the screen, the sprite bit maps are continuous. However, we know only where the sprite data starts (SL( )) and where our X and Y coordinates are located on the test screen. Multiply by 3 and we get the real row. To this row, add the column, or the previously calculated Y value. We've done this before, but not quite this way.

This process reveals a rather interesting and important concept of computer science. The bytes in memory are continuous, and it is up to the programmer to impose a structure. Many programming languages offer sophisticated data structures to eliminate the types of manipulations we've had to do. Commodore Basic however, does not, so we must create or build our own.

So far, the calculations are identical whether you are plotting or deleting.

Now the paths diverge. If we are plotting, we want to OR the value in BIT, our power of two based on the bit value of the dot we want to turn on. If we are deleting, we want to AND the value in BIT with the reverse image of BIT. For example, if BIT is 128 or 1000 0000, we want to AND with 0111 1111. The paths continue to diverge thanks to the IF statement at line 530. The appropriate colors for C( ) are assigned to CT, just as in the routine at line 300 and are assigned to C( ) before returning to the main command loop.

Clear or Fill Edit Square

```
600 V=22:H=4:GOSUB 990:PRINT "<35 SPACES)"
610 V=22:H=4:GOSUB 990:PRINT "ARE YOU SURE";:INPUT A$
620 IF LEFT$(A$,1)<>"Y" THEN 680
630 FOR I = SL(S)*64 TO SL(S)*64+62
640 POKE I,CT
650 NEXT
660 GOSUB 200
670 GOSUB 310
680 GOSUB 900
690 RETURN
```

This is a new function mandated by the size of a sprite. It is quite tedious to drag the cursor around the entire edit grid, even with the autorepeat cursor control keys. Besides, the change in sprite selection and edit mode freed up two function keys that can be used.

The start of this subroutine is an attempt at user friendliness. Since either action—filling the edit grid with 0s or 1s—would wipe out your sprite, I thought it would be good to ask if you were sure. Any answer that starts with "Y" will be taken for yes, thanks to the LEFT$ function in line 620.

Next, we simply POKE the appropriate value (held in CT as is customary for our color temporaries) into the 63 consecutive bytes that comprise the sprite in question.

The sprite changes immediately, but we need to update the edit grid to reflect the sprite's new condition. Therefore, we GOSUB to the subroutine at line 300 to restore the sprite, except that we must enter it at 310 to avoid changing the sprite number, and restore the screen display by the GOSUB to line 950.

Instructions

```
900 H=2:V=16:GOSUB 990:PRINT "F1 - CLEAR"
910 V=17:GOSUB 990:PRINT "F3 - FILL"
920 V=18:GOSUB 990:PRINT "F5 - LOAD"
930 V=19:GOSUB 990:PRINT "F7 - SAVE"
```

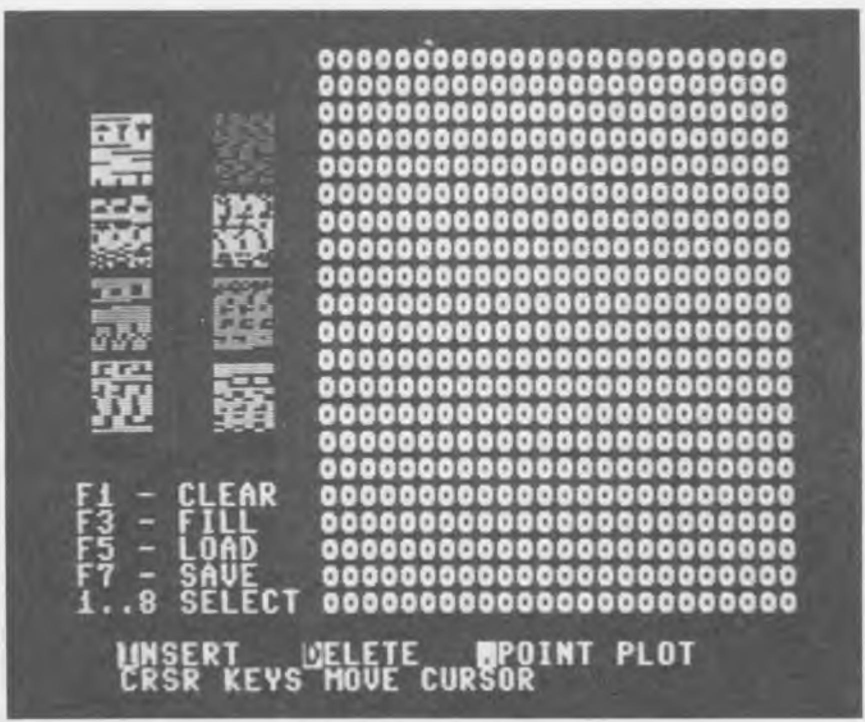
```
94Ø V=2Ø:GOSUB 99Ø:PRINT "1..8 SELECT"
95Ø H=4:V=22:GOSUB 99Ø:PRINT"<CTRL 9>I<CTRL Ø>NSERT<4 SPACE>
    <CTRL 9>D<CTRL Ø>ELETE<4 SPACE><CTRL 9>.<CTRL Ø>POINT PLOT"
96Ø V=23:GOSUB 99Ø:PRINT "CRSR KEYS MOVE CURSOR"
97Ø RETURN
99Ø PRINT "<CLR/HOME>";:FOR I = 1 TO V:PRINT "<CRSR DOWN>";:
    NEXT:FOR I = 1 TO H:PRINT "<CRSR RIGHT>";:NEXT
995 RETURN
```

This is pretty straightforward. The subroutine in line 99Ø is taken directly from Chapter 3.

The Main Program

```
1ØØØ PRINT "<SHIFT CLR/HOME>";
1Ø1Ø GOSUB 1ØØ:REM SET UP SPRITES
1Ø2Ø GOSUB 2ØØ:REM SET UP SCREEN
1Ø3Ø GOSUB 9ØØ:REM PRINT INSTRUCTIONS
1Ø4Ø GOSUB 4ØØ:REM START COMMAND LOOP
```

Figure 9.1
The Sprite Editor



The main program calls in the necessary subroutines at 100, 200, and 900 then GOSUBs to the command loop. We remain in the command loop, unlike the character editor. With the load and save options, to be added in Chapter 13, you now possess a full-featured sprite editor. Figure 9.1 shows the screen as it appears when you first run the program. The eight sprites are drawn with the bits that happened to be in memory when the photo was taken. Your screen may show different patterns. As you edit the sprites, of course, these shapeless blocks will take on meaningful designs.

A full program listing follows:

```
5 REM SPRITE EDITOR
10 DIM C(24,21)
20 O1=54:SCRN=1024:CRAM=55296
30 G1=SCRN+O1:C1=CRAM+O1
40 SX=0:BX=23:SY=0:BY=20
50 XS=31:YS=54
60 CC=1:TC=11:BC=15:PC=14
80 POKE 53280,0:POKE 53281,BC:POKE 646,TC
90 GOTO 1000
100 FOR I = 0 TO 7:READ SL(I):POKE 2040+I,SL(I):NEXT
110 FOR I = 53248 TO 53263 STEP 2
120 READ SX(J),SY(J):POKE I,SX(J)+XS:POKE I+1,SY(J)+YS
130 J=J+1:NEXT I
140 POKE 53269,255
190 RETURN
200 PRINT "<CLR/HOME>"
210 FOR I = 1 TO 21
220 PRINT "<14 CRSR RIGHT><24 SHIFT W>"
230 NEXT
290 RETURN
300 S=VAL(A$)-1:POKE 53280,PEEK(53287+S)
310 L=C1:C=0:M=0:N=0
320 FOR I = SL(S)*64 TO SL(S)*64+62
330 T=PEEK(I)
340 FOR J = 7 TO 0 STEP -1
345 IF T=>2^J THEN CT=PC:T=T-(2^J):GOTO 355
350 CT=TC
355 POKE L+(7-J),CT:C(N,M)=CT:N=N+1
360 NEXT J
370 L=L+8:C=C+1:IF C/3=INT(C/3) THEN L=L+16:C=0:M=M+1:N=0
380 NEXT I
385 X=0:Y=0
390 RETURN
400 GET A$:IF A$="" THEN 400
402 IF A$="<F1>" THEN CT=0:GOSUB 600:P=0:D=0
405 IF A$=>"1" AND A$<"9" THEN GOSUB 300:P=0:D=0
406 IF A$="I" THEN P=-1:D=0
407 IF A$="D" THEN P=0:D=-1
408 IF A$="." THEN P=0:D=0:P2=-1
409 IF A$="<F3>" THEN CT=255:GOSUB 600:P=0:D=0
```

```
410 IF A$="<CRSR UP>" THEN Y=Y-1
412 IF A$="<CRSR DOWN>" THEN Y=Y+1
414 IF A$="<CRSR RIGHT>" THEN X=X+1
416 IF A$="<CRSR LEFT>" THEN X=X-1
420 IF X<SX THEN X=BX
422 IF X>BX THEN X=SX
424 IF Y<SX THEN Y=BY
426 IF Y>BY THEN Y=SY
440 IF P OR P2 THEN GOSUB 500:P2=0
450 IF D THEN GOSUB 500
470 POKE (C1+X)+40*Y,CC:POKE(C1+OX)+40*OY,C(OX,OY)
480 OX=X:OY=Y
490 GOTO 400
500 B=(-1*(X>7))+(-1*(X>15))
510 IF X<8 THEN BIT=2^(7-(X AND 7)):GOTO 520
512 IF X<16 THEN BIT=2^(7-((X-8) AND 7)):GOTO 520
514 BIT=2^(7-((X-16) AND 7))
520 LOC=SL(S)*64+(Y*3+B)
530 IF P THEN 570
540 POKE LOC,PEEK(LOC) AND PEEK(LOC)-BIT
550 CT=TC
560 GOTO 590
570 POKE LOC,PEEK(LOC) OR BIT
580 CT=PC
590 C(X,Y)=CT
595 RETURN
600 V=22:H=4:GOSUB 990:PRINT "<35 SPACES)"
610 V=22:H=4:GOSUB 990:PRINT "ARE YOU SURE";:INPUT A$
620 IF LEFT$(A$,1)<>"Y" THEN 680
630 FOR I = SL(S)*64 TO SL(S)*64+62
640 POKE I,CT
650 NEXT
660 GOSUB 200
670 GOSUB 310
680 GOSUB 900
690 RETURN
900 H=2:V=16:GOSUB 990:PRINT "F1 - CLEAR"
910 V=17:GOSUB 990:PRINT "F3 - FILL"
920 V=18:GOSUB 990:PRINT "F5 - LOAD"
930 V=19:GOSUB 990:PRINT "F7 - SAVE"
940 V=20:GOSUB 990:PRINT "1..8 SELECT"
950 H=4:V=22:GOSUB 990:PRINT"<CTRL 9>I<CTRL 0>NSERT<4 SPACE>
    <CTRL 9>D<CTRL 0>ELETE<4 SPACE><CTRL 9>.<CTRL 0>POINT PLOT"
960 V=23:GOSUB 990:PRINT "CRSR KEYS MOVE CURSOR"
970 RETURN
990 PRINT "<CLR/HOME>";:FOR I = 1 TO V:PRINT "<CRSR DOWN>";:
    NEXT:FOR I = 1 TO H:PRINT "<CRSR RIGHT>";:NEXT
995 RETURN
1000 PRINT "<SHIFT CLR/HOME>";
1010 GOSUB 100:REM SET UP SPRITES
1020 GOSUB 200:REM SET UP SCREEN
1030 GOSUB 900:REM PRINT INSTRUCTIONS
1040 GOSUB 400:REM START COMMAND LOOP
1500 DATA 248,249,250,251,252,253,254,255
1510 DATA 16,24,64,24,16,48,64,48
1520 DATA 16,72,64,72,16,96,64,96
```

10 The Rainbow Revolution

So far we've seen only limited color. We were able to get borders and backgrounds in all 16 colors and text in any of the 16 colors. Our sprites were all one color. But your Commodore 64 is also capable of combining these colors within a character or sprite to yield a multicolored figure.

There is, of course, a penalty for this power: resolution. The more information a bit map needs to convey, the lower the resolution.

Multicolor Mode

Until now, our bit maps or character designs had to convey only whether a dot was to be on or off. In the multicolor mode, the bit map must also tell the VIC II what color to use for each dot. There are two ways to deal with this. One solution is to increase the size of the bit map or character design block. However, we've seen time and again that Commodore engineers favor economical memory usage.

A character set fits nicely into a 1,024-byte block of memory; a sprite fits into a 64-byte block. To expect the hardware (VIC II) chip to address different-sized chunks for multicolor mode would be a more difficult alternative than using the same-sized blocks and sacrificing design.

In multicolor mode, horizontal resolution is decreased by one-half, to 160 by 200 dots. The VIC II interprets pairs of bits as a pixel rather than as single bits. Up until now, I have been using the terms bit and pixel somewhat interchangeably, but in multicolor mode, the important technical distinction between them becomes clear.

A **bit** is a physical entity, a gate in a memory circuit that is either on or off, whereas a **pixel** is a logical creature. The number of pixels produced by a bit or a byte depends on how the information they represent is interpreted by

the computer hardware and software. To convince you of this, list a program or otherwise get a few lines of text on the screen and then try the following:

```
POKE 53270,PEEK(53270)OR 16
```

Pressing <RUN/STOP RESTORE> will reset the display to a readable form. This POKE turned on multicolor mode. The character bit map designs that once produced legible characters suddenly produce illegible ones. The reason lies in the amount of information that can be conveyed in a bit.

How much information can a pair of bits hold? A single bit can be either on or off, but a pair of bits can have the four arrangements: 00, 01, 10, and 11 in binary (0, 1, 2, and 3 in decimal notation). This will give us the ability to make a pixel any one of four colors, but a byte will now define only four pixels rather than eight.

The VIC II determines what colors these pixels represent by using color registers rather than "hardwiring" a pre-set color combination. The four colors can be any of the 16 available ones, depending on the contents of the registers, according to the scheme in Fig. 10.1.

Figure 10.1
Color Register Usage for Multicolor Characters

Bits	Location	Name of Register
00	53281	Background #0
01	53282	Background #1
10	53283	Background #2
11	55269+ screen position	Color RAM

There are many advantages to using color registers rather than predefined colors. The most important advantage is that it allows you to form a whole screen invisibly by setting all registers to the background color, drawing the screen, and then resetting the registers to the desired colors.

Masks

Since the characters are nearly unrecognizable when multicolor is turned on, we need to create a character set that will work in this mode. It is almost impossible to do so; the resolution is so poor that characters like Ms, Ns, and

Hs become nearly identical. But it can be done. In fact we already have a tool to do most of the work for us. The following subroutine from the character editor takes the pain out of reading your multicolor screen:

```
30 CHAR=14336
70 MASK=170
100 POKE 56334,PEEK(56334) AND 254:POKE 1, PEEK(1) AND 251
110 FOR I = 0 TO 2048:POKE CHAR+I,PEEK(53248+I)AND MASK:NEXT
120 POKE1,PEEK(1)OR 4:POKE 56334,PEEK(56334) OR 1
130 POKE 53272,31
140 POKE 53270,PEEK(53270) OR 16
```

If you've forgotten what lines 100-130 do, you might want to check back to Chapter 7. Line 140 instructs the VIC II to enter multicolor mode. If the characters are not readable, change their print color to white by pressing <CTRL 2>. I wouldn't want to work for a long time with these characters, but at least they can be read. Why?

The difference lies in line 110, which is slightly modified from the original. Instead of just transferring the exact bit map from the character ROM, the AND MASK assures that only certain combinations of bits get through. Fig. 10.2 shows a graphic example of this process.

Figure 10.2
The Process of Masking

ROM image for the center bar of the letter A		0	1	1	1	1	1	1	0
Mask (170)	**AND**	1	0	1	0	1	0	1	0
Result		0	0	1	0	1	0	1	0
Bit Pairs Before AND	0 1	1	1		1	1		1	0
Bit Pairs After AND	0 0	1	0		1	0		1	0

Masking is very important because it filters out the unwanted bits—those bits that were causing the colors to interfere with legibility. Since the basis for the operation was the original ROM character sets, the basic letter design is preserved even if the resolution is decreased. There are obviously four possible masks: one for each color/bit combination.

The first of these—00000000—is really useless because it will filter out all bits and leave you with an eight by eight pixel-less image. The next two—01010101 (decimal 85) and 10101010 (decimal 170)—will force your characters to take on the colors in background register 1 or 2, respectively.

The last mask, like the first one, is not very useful; 11111111 will leave you with an eight by eight solid block whose color will be dependent upon the value held in color RAM at its screen location.

With this drastic lack of resolution, why would multicolor characters be of any use? Many games or educational programs that use character graphics could benefit from a more colorful screen display—those with animated characters especially; games like *Space Invaders* spring immediately to mind. Using multicolor characters, it would be easy to create bloodthirsty red jaws on a blue space invader.

However, the effectiveness of such games depends on their having readable characters available for instruction and scoring screens. Straight-text characters are not too readable. But there is a solution to this quandary. The VIC II chip interprets characters as multicolor only if the color RAM underneath it has its third bit set. This includes all numbers from eight to 15 and many above 15. So POKEing the top two lines of color RAM with a number less than eight will cause the characters printed there to be displayed as regular old-fashioned characters that are less colorful but more readable. Try the following:

```
FOR I = 53269 TO 53269+79:POKE I,1:NEXT
```

If you saw no change in the characters, you still have the masked character set. To get the regular set, press <RUN/STOP RESTORE> and run 140, assuming the program fragment is still in memory. Unfortunately, you must type the line above in the dark, since the characters are unmasked and illegible. You also need to have something in the top of the screen. If all of these conditions are met, the unreadable multicolor characters will change into clear, white characters.

This is a lot of control to have over your character graphics. With 255 characters in a set, you can design many multicolored figures and never touch the basic alphabet.

Character Editor Modifications

The character editor from Chapter 7 can be easily converted for use in multicolor mode. The following additions:

```
70 M=85
105 POKE 53270,PEEK(53270) OR 16:REM ENTER MULTICOLOR MODE
```

and this change to line 110:

```
110 FOR I = 0 TO 2048:POKE CHAR+I,PEEK(53248) AND M:NEXT
```

will do it. Line 70 sets up a color mask for one of the available colors (see Fig. 10.2). The mask filters out the inappropriate columns of the character design, assuring that all characters are composed of 01 bit pairs. The mask is active only during the copying of characters from the ROM set. You are free to create a character with any or all of the bit pair combinations.

Multicolored Sprites

As you may have suspected, characters are not the only graphic blocks that can be displayed in multicolor. Sprites, too, have a multicolor mode.

To display a sprite in multicolor mode, POKE the sprite number into location 53276 using our familiar algorithm:

```
POKE 53276,PEEK(53276) OR (2^N)
```

Remember that N represents the sprite number. Unlike character sets, an individual sprite can be displayed in multicolor mode. To demonstrate multicolor sprites, load the original sprite demonstration program (from Chapter 8), which displayed the space ship, and add one additional line:

```
POKE 53276,PEEK(53276) OR (2^1)
```

When you run this, you should see an ominous black space ship with white dots on it. Fig. 10.3 shows the space ship design split into bit pairs.

Figure 10.3
The Space Ship Design Revisited

00 00 00 00	00 11 11 00	00 00 00 00	(0)	(60)	(0)
00 00 00 11	11 11 11 11	11 00 00 00	(3)	(255)	(192)
00 00 00 11	11 01 10 11	11 00 00 00	(3)	(219)	(192)
00 00 01 10	11 01 10 11	01 10 00 00	(6)	(219)	(96)
00 01 11 10	11 01 10 11	01 11 10 00	(30)	(219)	(120)
00 00 11 11	11 11 11 11	11 11 00 00	(15)	(255)	(240)
00 00 00 11	11 11 11 11	11 00 00 00	(3)	(255)	(192)
00 00 00 00	11 11 11 11	00 00 00 00	(0)	(255)	(0)
00 00 00 00	00 11 11 00	00 00 00 00	(0)	(60)	(0)
00 00 00 00	00 01 10 00	00 00 00 00	(0)	(24)	(0)

You can now see the pairs of bytes that make up a pixel in multicolor mode. The POKE numbers for the sprite bit map or design remain unchanged; only their interpretation differs. The color registers used for multicolored sprites differ from those used for the characters. The details are presented in Fig. 10.4.

Figure 10.4
Color Register Usage for Multicolor Sprites

Bits	Location	Name of Register
00	53281	Background #0
01	53285	Sprite multicolor #0
10	53287 + N*	Sprite color
11	55286	Sprite multicolor #1

* N represents the sprite number

It's easy to see why the ship is black—most of it is composed of pairs of 1s. Sprite multicolor register #1 is set to 0 or black. Changing this register produces a more colorful ship:

```
POKE 53286,2
```

Bingo! All the pairs of 1s now display as red. Sprite multicolor register #0 and the regular sprite color register control the other dots. The sprite color register is set to 1 and the multicolor register is set to 244, which will give purple. The color values are so similar that the dots look white. Both of these must be changed to create a good-looking space ship. This is due to the design of the ship, because of its symmetry.

Since the background register is used to determine the color of all the 00 pairs, you can have a maximum of three different colors in a multicolored sprite. Two of these colors—those held in the two multicolor registers—are the same for all sprites that are displayed in multicolor mode, while one (the regular sprite color register) can be unique for each sprite. This provides a great deal of flexibility, especially if you combine multicolored sprites with multicolored characters.

A multicolored sprite is identical to a regular sprite in all respects but color. There are half the number of pixels in a multicolored sprite compared to a normal sprite, so multicolored sprites produce very blocky designs when they are expanded. Assuming you've modified the space ship program and run it, the following may help you visualize some of these ideas:

```
POKE 53277,1: POKE 53271,1
```

Now you should have a big black space ship somewhere on the screen. The white (10) pixels and the purple (01) pixels should be clearly visible. Let's play with the bottom of the ship, which looks like this:

00 00 00 00	00 01 10 00	00 00 00 00	(0)	(24)	(0)

The difference in the color registers is evident here; what was two white dots in regular color mode becomes two dots of different colors. We can change the design to achieve two dots of the same color:

```
POKE 924,40
```

00 00 00 00	00 10 10 00	00 00 00 00	(0)	(40)	(0)

As an alternative we can change them to purple with:

```
POKE 924,20
```

Why 20?

00 00 00 00	00 01 01 00	00 00 00 00	(0)	(20)	(0)

As a third alternative, we can reduce it to a single pixel:

```
POKE 924, 16
```

00 00 00 00	00 01 00 00	00 00 00 00	(0)	(16)	(0)

The problem with the last alternative is that the bottom of the ship is now lop-sided. We'd have to redesign the whole ship to achieve symmetry. There is a final alternative: change the sprite multicolor register #0 to white. But if we do this, we lose the multicolor advantage and are now reduced to only two colors in a sprite.

Remember—POKE 53269,0 will erase the sprites for you.

Fortunately, the sprite editor from Chapter 9 can handle multicolored sprites very simply. We need only make some modest changes, which, as you might suspect, involve masks. These lines should be added to the program:

```
70 M(0)=0:M(1)=85:M(2)=170:M(3)=255:REM COLOR MASKS
150 POKE 53276,255:REM ENTER MULTICOLOR SPRITE MODE
622 V=22:H=4:GOSUB 990:PRINT "<20 SPACEs>";
624 V=22:H=4:GOSUB 990:PRINT "COLOR (0,1,2,3)";:INPUT A$
626 A=VAL(A$)
628 IF A<0 OR A>3 THEN 622
```

Line 640 requires a small edit to use the masks:

```
640 POKE I,M(A)
```

As written, these changes use the default colors for sprite multicolor mode. You can POKE the sprite multicolor registers with the correct color values prior to running the program, if you require other colors. As a project, you might modify the program to allow changes of the color registers within the program.

Extended Background Color Mode

The extended background color mode works only for characters. It allows you to assign different colors to the foreground and background of a character. This mode differs from multicolor mode in that the character design remains intact. A 1 signifies a pixel in the character or foreground color and 0 signifies a pixel in the background color. The difference lies in where the VIC II finds the foreground and background colors. Rather than relying on the usual color RAM location, it uses four registers (see Fig. 10.5).

Figure 10.5
Extended Background Color Registers

Character Code	Location	Name of Register
0-63	53281	Background register #0
64-127	53282	Background register #1
128-191	53283	Background register #2
192-255	53284	Background register #3

The tradeoff here is that there are only 64 characters in extended background color mode. Only the first 64 character designs or bit maps are used; those maps with display values (POKE values, not ASCII or CHR$( ) codes) above 63 are ignored.

While it is fortunate that you can fit most of the necessary characters into a table of 64, this mode is very limiting. But with the range of applications possible, there may be many programs where extended background mode is just what the doctor ordered. To get into this mode, try this:

```
POKE 53265,PEEK(53265)OR 64
```

You'll notice little difference in most listings. However, if the screen includes any inverse characters such as those that result from the use of SHIFT or the C=64 key, these will have a strange and wonderful appearance. For example, the <SHIFT CLR/HOME> character—normally an inverse heart—displays as an S on a cyan background. This will give you some idea of the possible effects:

```
10 PRINT "<SHIFT CLR/HOME>"
20 FOR I = 1 TO 193 STEP 64
30 POKE 1024,I
40 GET A$:IF A$="" THEN 40
50 NEXT I
```

This simple program will POKE a 1,65,129, and 192 into location 1024. Pressing any key will advance to the next code. The codes all represent the letter A.

11 Hi-Res Hijinks

Until now we have dealt with the computer's screen as an array of characters—25 rows of 40 characters each, 1,000 in all. Although we have almost total control over what appears in each of the character positions, we cannot easily create characters "on the fly." We have no way of controlling individual pixels on the screen. In high resolution graphics (or **hi-res** for short) we control each of the 64,000 pixels on the screen.

About High Resolution

As you might imagine, much more memory must be devoted to the screen in hi-res mode than in character mode. In order to have a unique bit arrangement for every byte on the display, we must devote 8K (actually 8,192) bytes to hold the screen bit map. The reason for this is the way character graphics work.

A full character set, you'll recall, has 255 characters. The designs, or bit maps, of these characters occupy 2,048 bytes. You can have two of these in memory at a time. This gives us 4,096 bytes devoted to character maps. Remember, too, that the VIC II chip uses some electronic wizardry to overlay its ROM character set on the VIC II addressable RAM. So when using the normal Commodore 64 character sets, memory usage is not a problem.

In high resolution, every byte on the screen represents a unique character; it is as if we had four complete character sets in memory. As we draw on the high resolution screen, we will be continually changing the design of the characters to reflect our graphic image.

This view of high resolution graphics is unique to Commodore computers. Every other personal computer with high resolution graphics maps the high resolution screen as a continuous stream of bits rather than as an array of

characters. The advantage of this bit-stream approach is speed. It is much faster to find a bit and turn it on using this approach than with Commodore's character matrix approach.

The advantage of Commodore's approach is that it is compatible with all of the capabilities of text mode. From the programmer's standpoint, hi-res is difficult to use on a plain vanilla—unexpanded—Commodore 64. But with the utilities and alternative languages discussed in the final chapters of this book, you'll find hi-res as simple to implement on the Commodore 64 as it is on the Apple or Atari. Meanwhile, we can do some rather impressive graphics in Basic.

There are two forms of high resolution graphics: normal and multicolor. The usual difference in resolution between the two modes applies here. In normal high resolution mode, you control 200 lines of 320 pixels (64,000 dots); while in multicolor hi-res, you control 200 lines of 160 pixels (32,000 dots). In multicolor hi-res, just as in multicolor sprite and character graphics, each pair of bits controls a pixel, which divides the total number of pixels in half.

Regular High Resolution Mode

Regular high resolution mode allows 200 lines of 320 pixels in two colors. In order to use this mode, you must tell the VIC II where to find your 8K screen bit map by POKEing the VIC page number into location 53272. This is identical to what we did when we used custom character sets. The POKE, however cannot damage bits 4 through 7 of this VIC register. Furthermore, the map is 8K long, half of the VIC II's address range. We are more limited as to where it can be placed than we were with our custom characters. To place the map at 8192, try:

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

Having informed VIC II of the map's location, we must turn on hi-res mode by setting bit 5 of location 53265 with:

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

We can get out of hi-res mode by entering these statements:

```
POKE 53272,PEEK(53272) OR 4:POKE 53265,PEEK(53265) AND 223
```

Readers who want a refresher on this type of statement should look back to Chapter 2.

When you enter hi-res mode, the screen will be rather unspectacular. You'll see random blocks of color and, depending on what you were doing before entering the mode, scattered dots or characters. Since you've told the VIC II to use locations 8192 to 16383 as a bit map of the screen, whatever garbage was there is now displayed, dutifully, as a bit-mapped image.

If there were character sets in this space (we did store our custom characters here), you'll see them. In hi-res mode, however, you will not be able to control them. Their bit maps are stationary, frozen into RAM. All you are apparently able to control is huge blocks of color. Not very hi-res.

These blocks of color represent the text screen. Just as in the various character and sprite modes, the VIC II needs to know what color you intend to draw with. The colors in normal hi-res mode are derived from the text screen, because color RAM is not used in this mode. The leftmost nibble of a byte (the high nibble) controls the color of bits set to 1 in the bit map; the rightmost nibble of a byte (the low nibble) controls the color of bits set to 0 in the bit map. This makes for a complicated color scheme, but it is regular (see Fig. 11.1).

Figure 11.1
Normal High Resolution Color Values

Plot Color	POKE*	+ Background
0 Black	0	0
1 White	16	1
2 Red	32	2
3 Cyan	48	3
4 Purple	64	4
5 Green	80	5
6 Blue	96	6
7 Yellow	112	7
8 Orange	128	8
9 Brown	144	9
10 Pink	160	10
11 Grey 1	176	11
12 Grey 2	192	12
13 Lt Green	208	13
14 Lt Blue	224	14
15 Grey 3	240	15

* To achieve the desired combination, add the POKE value to the desired background color. For example, to plot in red on a white background, POKE the screen with 33 (32 + 1).

Because the color is determined by the 1,000-character text screen, we can have all 256 variations of plot color and background color on the screen at the same time. But—and this is a big but—for artistic purposes, each eight by eight dot hi-res block can have only one plot color and one background color. There's lots of planning involved, therefore, in using more than one color combination on the regular high resolution screen.

We're ready to set up a screen now:

```
100 POKE 53272,PEEK(53272) OR 8 :REM SCREEN AT 8192
110 POKE 53265,PEEK(53265) OR 32:REM TURN ON HIRES
120 FOR I = MAP TO MAP+7999:POKEI,0:NEXT:REM CLEAR BIT MAP
130 FOR I = SCRN TO SCRN+999:POKEI,124:NEXT:REM SET COLORS
140 POKE 53280,7:REM SET BORDER COLOR
150 RETURN
```

With the hi-res screen set up, we need to create a subroutine that will accomplish the plotting we want to do. Ideally, we should be able to send the subroutine an X and a Y coordinate and let the subroutine do the plotting. The subroutine turns out to be almost identical to the subroutine used in the character editor. This makes perfectly good sense, since we are, conceptually, editing a character set.

```
500 REM
510 ROW=INT(Y/8):CHAR=INT(X/8)
520 LINE=Y AND 7
530 BIT = 7-(X AND 7)
540 BYTE=MAP+ROW*320+CHAR*8+LINE
550 POKE BYTE,PEEK(BYTE) OR 2 ↑ BIT
560 RETURN
```

The major differences between this subroutine and the subroutine in the character editor are line 510 and the variables LINE and BIT. Since the character editor operated on eight by eight bit maps, there was a one-to-one correspondence between what we were editing and the data. On the high resolution screen, however, the X and Y coordinates can range between 0 and 319, or 0 and 199, respectively.

In order to put a dot at these coordinates, they must be converted to an equivalent byte address within an enormous 8K bit map (see Fig. 11.2). The screen is organized as if it were a grid of characters. To put a dot at location 0,1 we have to manipulate the second screen byte. Recall that when we wanted to put a character one vertical position down and in the first column, we added 40 to the starting screen location. To put a dot at 8,0 we need to manipulate byte number 9 since the first eight bytes are devoted to X coordinates 0 to 7, and Y coordinates 0 to 7.

Figure 11.2
Bit Map Organization of High Resolution Screen

	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	 X
0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
2	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
3	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
4	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
5	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
6	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
7	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
8	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
9	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	....
Y	....	....	....	....

Enlargement of upper left-hand corner of screen (30 bytes shown)

	0	8	16	
0	8192	8200	8208	....
1	8193	8201	8209	....
2	8194	8202	8210	....
3	8195	8203	8211	....
4	8196	8204	8212	....
5	8197	8205	8213	....
6	8198	8206	8214	....
7	8199	8207	8215	....
8	8512	8520	8528	....
9	8513	8521	8529	....
Y	....	....	....	....

Addresses or locations of bytes shown above, assuming bit map starts at 8192.

The algorithms in this subroutine take care of all the nasty details. First, the X and Y coordinates are divided by 8 to bring them within the 40 by 25 line range of the text screen (line 510). Then the variable LINE is calculated by ANDing Y and 7 (in binary, seven is 00000111). This will represent that line in the eight by eight character grid in which Y appears; it will always be a number from 0 to 7. Try it yourself with valid Y values (0 to 24):

```
FOR I = 0 TO 24:PRINT I AND 7,:NEXT
```

Then the bit or pixel represented by the X value is calculated (line 520). This, too, will always be a number between 0 and 7. Notice how the X value gets flipped by subtracting the ANDing value from 7. It must be flipped because the bit numbers run backwards on the screen. The first pixel, at 0,0, is represented by the seventh bit of the first byte of the screen: 1000 0000; not the first bit: 0000 0001. To see this in action, try entering:

```
FOR I = 0 TO 39:PRINT 7 - (I AND 7),:NEXT
```

Now that we know the LINE and the BIT we can compute the actual byte in line 540. ROW * 320 is the equivalent of ROW * 40 in the character set editor—in hi-res, the screen is 320 dots wide rather than 40 characters wide. On the vertical scale each jump must be by eight dots, so we must use CHAR*8. To this we can add LINE for the specific byte within the correct eight-byte vertical block. Finally, the whole thing gets added to our bit-map location, MAP. This yields an address within the bit map that holds the particular bit we wish to manipulate.

The final step, then, is to OR that bit value onto the screen (line 550). Of course the reason we must OR it with the current contents of the byte rather than just POKEing it in is that there might already be dots in the byte. A simple OR would erase the previously plotted points.

This is quite a chore just to plot a point! And, unfortunately, it's slow. That's the penalty exacted by Basic. If you're interested in speed, learn assembly language (ugh!) or experiment with one of the utility programs discussed later in this book.

Now that we have all the subroutines we need, let's finally make use of hi-res:

```
5 REM PROGRAM BOX
10 MAP=8192:SCRN=1024
20 GOSUB 100 :REM SET UP SCREEN
30 FOR I = 20 TO 50:X=I:Y=20:GOSUB 500:NEXT
40 FOR I = 20 TO 50:X=20:Y=I:GOSUB 500
50 Y=I:X=50:GOSUB 500:NEXT I
60 FOR I = 20 TO 50:X=I:Y=50:GOSUB 500:NEXT
90 GOTO 90
```

You're ready to run this program, but it will take some time to initialize the bit map and the screen, so be patient. You can use <RUN/STOP RESTORE> to return to the text page. Of course, we could have made a box with a custom character set. Save this program; you'll need it later. Try the following quick program:

```
5 REM PROGRAM SINE WAVE
20 GOSUB 300:GOSUB 100
30 FOR I = 1 TO 31.9 STEP .1
40 X=X+1:Y=100-SIN(I)*50
50 GOSUB 500
60 NEXT I
```

A Quick Clear Subroutine

The plot speed is certainly annoying, but the most troublesome pause occurs during the clearing of the bit map. 8,192 bytes is a lot to POKE. Below is a short assembly language subroutine that functions identically to the loop in line 120, but is much faster because it does not have Basic's overhead to contend with. Add the following to your program:

```
120 SYS 49152
300 FOR I = 0 TO 20
310 READ T:POKE 49152+I,T
320 NEXT I
330 POKE 251,0:POKE 252,32
340 RETURN
350 DATA 169,00,170,168,145,251,200,240,02,208,249,230,
         252,232,224,32,240
360 DATA 02,208,240,96
```

We can't delve into the innards of assembly language here. However, each number in the DATA statements is either a 6510 **op code** (instruction) or data for the preceding instruction. The program is tucked into the free RAM at 49152 but could be placed anywhere in memory and is therefore called relocatable. This is not always true of assembly language programs; many are permanently frozen at the location in memory where they were assembled.

The **SYS** statement in line 120 replaces the FOR/NEXT loop and calls the assembly language subroutine. When Basic gets to the SYS statement, it saves the current program pointer so that it can return to the line where it left, then jumps to the address following the SYS statement.

Assuming the assembly language code doesn't have serious bugs, the computer returns to Basic when it finds an **RTS** (ReTurn from Subroutine) op code. The numeric representation of RTS is 96. Note the last number in the data list—the RTS instruction. The computer is smart enough, because it saved some information before it jumped, to return to the Basic program wherever it left it.

The only other addition is the two POKEs in line 330. These are data used by my assembly language subroutine; specifically, they are the address of the bit map. 8192 is the 32nd page of 6510 memory (32 * 256 = 8,192). Location 251 should always be POKEd with a zero because bit maps must fall on even page boundaries. If you move the bit map the VIC II bank, you must change the POKE to 252 to reflect the change. Also remember that you tell the VIC II where the bit map is in VIC II pages of 1,024 bytes (8 * 1,024 = 8,192 as in line 100).

If you run this program now, you'll notice a dramatic increase in startup speed.

Artifacting

We are, of course, in regular hi-res mode, with one plotting color and one background color. But there's an interesting phenomenon called **artifacting** that occurs because of the way American color televisions work. Artifacting is the production of color by creating a video signal that is too narrow. By plotting single dots, or single lines, on the hi-res screen, we'll be able to observe this phenomenon. Alter your program to read:

```
30 FOR I = 10 TO 310 STEP 2
40 FOR Y = 10 TO 20
50 X=I:GOSUB 500
60 NEXT Y
70 NEXT I

130 FOR I=SCRN TO SCRN+999:POKE I,16:NEXT I
```

The change in line 130 assures a clear view of the artifacting process. 16 is supposed to yield white dots on a black background (Fig. 11.1), but instead, you will see a multicolor strip. The computer thinks it's generating a black and white display; but a television or composite monitor (like the Commodore 1701) has trouble producing this display.

If you run the same program with a STEP 1 (or with no STEP statement) in line 30, you'll get a white band! The colors are accidents of video technology and are commonly used on all computers that produce TV output to create more colorful displays. Artifacting is difficult to control and, at best, is a poor substitute for true multicolor displays.

Aspect Ratios

The square drawn by our first hi-res program wasn't very square. You might want to take another look at that program to see for yourself. There are 50 dots on each side, and the rectangular look of the square results from the aspect ratio of the pixels in the display. The pixels aren't square—they are about 20% taller than they are wide.

For most purposes, this difference in **aspect ratio** has little effect. However, if you are a stickler for accuracy or are merely trying to draw a circle that looks like a circle, you'll have to adjust the aspect ratio. Fix this by adding an aspect ratio corrector to the plotting subroutine:

```
500 Y=INT(Y*.8)
```

Since aspect ratios are so critical for circles, we might as well draw a circle. The circle program below uses the following formulas to calculate the correct X and Y coordinates:

```
X = XC + R * COS(A) and Y = YC + R * SIN(A)
```

where XC and YC are the center of the circle (X and Y coordinates), R is the radius of the circle, and A is the angle in radians. Schools don't deal much with radians, but the Basic in most microcomputers performs all trigonometric functions in radians rather than in degrees. A full 360-degree rotation is expressed in 6.283183 radians (that's 2 times pi). If the radius for X and the radius for Y are different, the same formulas calculate the plotting points for an ellipse.

If all this math worries you, relax. Take it easy and slow and experiment—your Commodore 64 doesn't mind a bug or two. Because we have so few tools in Commodore Basic, it is necessary to get a bit more mathematical than would be necessary if a CIRCLE command were available from Basic. But many others have trod this path before, so you don't have to. If you are interested in this type of graphics programming check the reading list in the appendix.

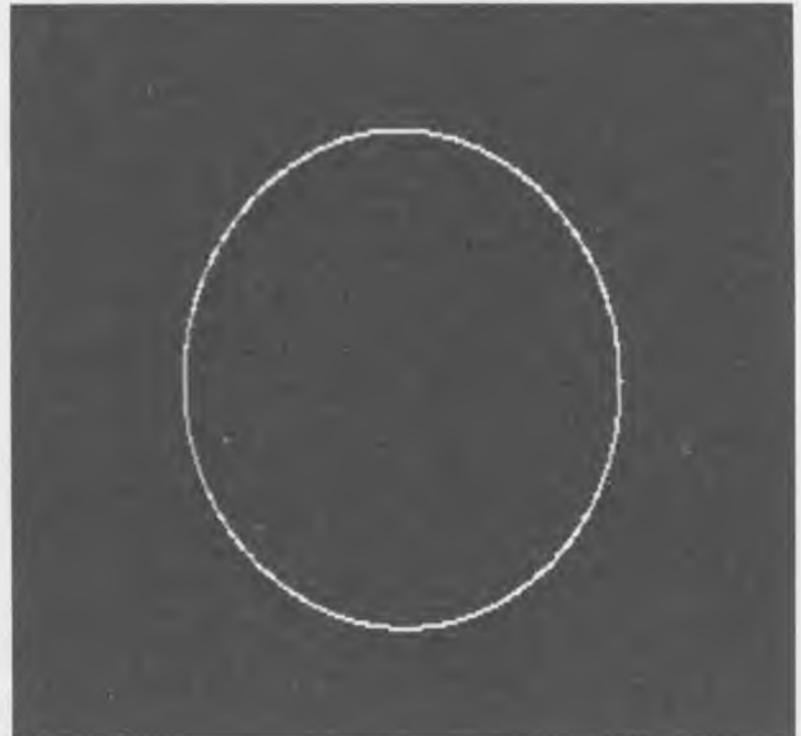
Here's the front end of the hi-res program that draws a circle; be sure to set the plotting, background, and border colors to a pleasing combination:

```
5 REM PROGRAM CIRCLE
10 MAP=8192:SCRN=1024
20 GOSUB 300:GOSUB 100
```

```
30 XM=320:YM=200:XC=160:YC=100+(100-100*.8):RX=90:RY=90
40 FOR A = 0 TO 2*<SHIFT UP_ARROW> STEP .01
50 DX=RX*COS(A):DY=RY*SIN(A)
60 X=XC+DX:Y=YC+DY
70 GOSUB 500
80 NEXT A
```

Your screen should look like Fig. 11.3.

Figure 11.3
Output of Circle Program



Aliasing

Notice the artifacting? You should also notice another computer graphics phenomenon—**aliasing**, also called "the jaggies." Because there are a limited number of points on the screen, lines that are not horizontal, vertical, or at a 45-degree angle must be approximated. Sometimes this approximation results in more than one pixel on a horizontal line (see Fig. 11.4).

Figure 11.4
Aliasing—Vertical or Horizontal Lines vs. Diagonals

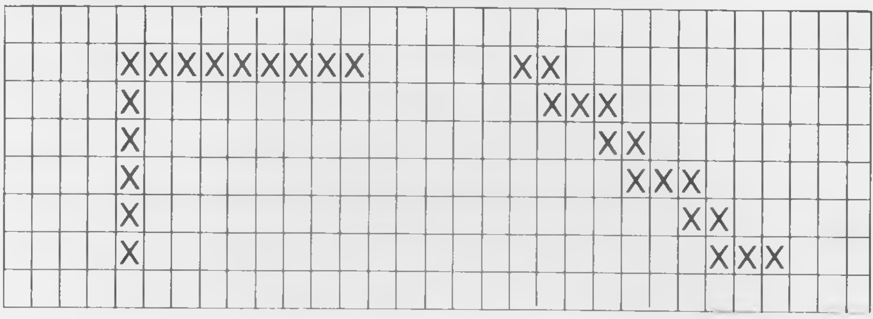
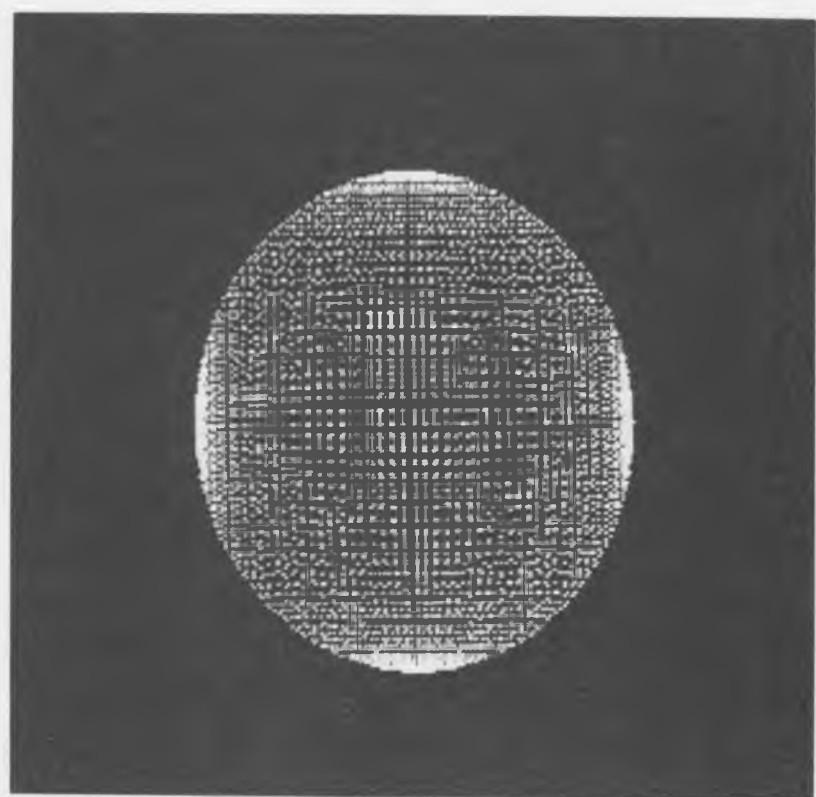


Figure 11.5
Output of Globe Program



Diagonal lines, or in this case, the curved portions of the circle, are not perfectly smooth. The simplest way to eliminate aliasing is to increase the resolution, an option that unfortunately is not available on the Commodore 64 because the resolution of the screen is fixed. It means, also, that in multicolor mode, the aliasing will be worse since the horizontal resolution will be halved.

You can have a lot of fun with the circle program by altering the values of the X and Y radii. Why not do it in another loop so that the program draws nested ellipses? Fig. 11.5 shows the result of my modifications as produced by the following program. The subroutines at 100, 300, and 500 are unchanged from our previous programs.

```
5 REM PROGRAM GLOBE
10 MAP=8192:SCRN=1024
20 GOSUB 300:GOSUB 100
30 XM=320:YM=200:XC=160:YC=100+(100-100*.8):RX=90:RY=90
35 FOR RX=0 TO 90 STEP .5
40 X=XC+RX:Y=YC:GOSUB 500
45 X=XC-RX:Y=YC:GOSUB 500
50 X=XC:Y=YC+RY:GOSUB 500
55 X=XC:Y=YC-RY:GOSUB 500
60 FOR A= 0 TO 2*<SHIFT UP_ARROW>/4 STEP .05
65 DX=RX*COS(A):DY=RY*SIN(A)
70 X=XC+DX:Y=YC+DY:GOSUB 500
72 X=XC-DX:Y=YC+DY:GOSUB 500
74 X=XC+DX:Y=YC-DY:GOSUB 500
76 X=XC-DX:Y=YC-DY:GOSUB 500
78 X=XC+DY:Y=YC+DX:GOSUB 500
80 X=XC-DY:Y=YC+DX:GOSUB 500
82 X=XC+DY:Y=YC-DX:GOSUB 500
84 X=XC-DY:Y=YC-DX:GOSUB 500
86 NEXT A
88 NEXT RX
90 GOTO 90
```

Multicolor Hi-res

Multicolor hi-res functions the same way that multicolor sprites and characters do: the four combinations of bits—00, 01, 10, and 11—result in different colors. Naturally, the color information is gained from different locations from those of multicolor sprites or characters (see Fig. 11.6).

The two mixed bit arrangements, 01 and 10, get their colors from the same locations as does the normal high resolution mode. In this mode, the normal background color register is active, as is color RAM. Since we have

Figure 11.6
Multicolor High Resolution Color Locations

Bits	Location	Name
00	53281	Background #0
01	upper 4 bits of screen memory	
10	lower 4 bits of screen memory	
11	55296 + loc*	Color RAM

* loc refers to the position of a dot in a standard 40 × 25 character matrix.

already devised algorithms to find a location on the character page that corresponds to a dot on the hi-res page and since color RAM has the same organization as normal screen RAM, it will be simple to deal with color RAM. In short, just about everything is in place for multicolor high resolution graphics.

First, let's modify the Program Box. Below is the whole thing in its entirety. Rather than type it all in again, recall your circle or box program and simply edit it to match this:

```
*  5 REM PROGRAM THREE BOXES
*  10 MAP=8192:SCRN=1024:CRAM=55296
   20 GOSUB 300 :GOSUB 100 :REM SET UP SCREEN
*  22 X1=20:X2=50:S=2:GOSUB 30
*  24 X1=71:X2=101:S=2:GOSUB 30
*  26 X1=120:X2=150:S=1:GOSUB 30
*  28 GOTO 90
*  30 FOR I = X1 TO X2 STEP S:X=I:Y=20:GOSUB 500:NEXT
*  40 FOR I = 20 TO 50:X=X1:Y=I:GOSUB 500
   50 Y=I:X=X2:GOSUB 500:NEXT I
*  52 IF S=2 THEN 60
*  54 FOR I = 20 TO 50:X=X1+1:Y=I:GOSUB 500
*  56 Y=I:X=X2+1:GOSUB 500:NEXT I
*  60 FOR I = X1 TO X2 STEP S:X=I:Y=50:GOSUB 500:NEXT
*  65 RETURN
*  70
   90 GOTO 90
   100 POKE 53272,PEEK(53272) OR 8 :REM SCREEN AT 8192
   110 POKE 53265,PEEK(53265) OR 32:REM TURN ON hi-res
*  115 POKE 53270,PEEK(53270) OR 16:REM TURN ON MULTICOLOR
   120 SYS 49152
*  130 FOR I = SCRN TO SCRN+999:POKEI,32+5:NEXT
*  135 FOR I = CRAM TO CRAM+999:POKEI,7:NEXT
   140 POKE 53280,7:REM SET BORDER COLOR
*  145 POKE 53281,0:REM SET BACKGROUND COLOR
   150 RETURN
   300 FOR I = 0 TO 20
   310 READ T:POKE 49152+I,T
```

```
320 NEXT I
330 POKE 251,0:POKE 252,32
340 RETURN
350 DATA 169,00,170,168,145,251,200,240,02,208,249,230,252,
         232,224,32,240
360 DATA 02,208,240,96
500 Y=INT(Y*.8):REM CORRECT FOR ASPECT RATIO
510 ROW=INT(Y/8):CHAR=INT(X/8)
520 LINE=Y AND 7
530 BIT = 7-(X AND 7)
540 BYTE=MAP+ROW*320+CHAR*8+LINE
550 POKE BYTE,PEEK(BYTE) OR 2 ^ BIT
560 RETURN
```

Changes from the previous version are marked with asterisks. Let's look at each of these changes. The new variable CRAM holds the location of color RAM. Because we want to plot three squares, one in each color since there's not much point plotting a square in the background color, I've made the plotting segment a subroutine but kept the same line numbers to minimize retyping.

All you need to do is edit the old lines to add the variables X1, X2, and S where shown. X1 is the starting point of the square and X2 is the end. S represents the STEP for use in the FOR/NEXT loop. Since we must plot 01s and 10s to get two of the colors, this STEP will accomplish the necessary skips in the FOR/NEXT loop. However, we don't want these skips in the final color, represented by the 11 plots. And we must broaden the uprights with lines 52 to 56. You can watch the color change as the plots are doubled. Judicious use of the <INST/DEL> key should allow these lines to be edited painlessly.

The screen set-up subroutine has been changed to reflect multicolor mode (line 115). Both screen memory and color RAM must be initialized. Screen memory is initialized to color 32 (red) for the 01 plots, and 5 (green) for the 10 plots (Fig. 11.1). Color RAM is initialized to 7 (yellow). Then the border is set to yellow and the background to black.

The program now draws three boxes—green, red, and yellow. You'll also notice a slight artifacting on the uprights; the yellow tends to white and the red tends to blue or purple. There's not much you can do about this.

Professional graphics programmers always plot upright, single-pixel lines double so that the colors are vivid. This problem is less severe on the Commodore 64 than on other personal computers and, depending on the background color chosen, may not be a problem at all. The green uprights, for example, look fine on a black background, but I'd plot the red twice.

Multicolor Sine/Cosine

The multicolor mode allows us to make the sine graph a little more impressive by adding a graph of the cosine. Retrieve your Three Boxes program and delete lines 22 through 65. Then add the following:

```
5 REM PROGRAM SINE/COSINE
25 X1=0:X2=1
30 FOR I = 0 TO 16 STEP .1
40 X=X1+2:X1=X:Y=100-SIN(I)*50:GOSUB 500
50 X=X2+2:X2=X:Y=100-COS(I)*50:GOSUB 500
60 NEXT
```

The subroutines are identical to those in the previous program, as are lines 10, 20, and 90. The only major change from our previous sine plotter is that another X variable has been added. Sine X1 starts at 0 and X2 starts at 1; they are always one plot position apart. Therefore, one plots 01s and the other plots 10s, which we see as different colors. The range of the FOR/NEXT loop was also reduced since multicolor has only half the resolution of regular hi-res.

Although the high resolution modes use a lot of memory, they are very flexible and allow for more sophisticated screen images than Commodore's character graphic mode. Like the character graphic modes, sprites can be superimposed over the hi-res screen to provide detailed backgrounds.

The lack of plotting support from Basic is unfortunate, but can be overcome by purchasing one or both of Commodore's Basic enhancements, "Simon's Basic" or the "Super Expander." With these cartridges you lose 8K of user space, but gain valuable Basic commands to ease manipulation of the high resolution screen. We'll take a closer look at Simon's Basic in Chapter 15. Meanwhile, you can create high resolution screens with the editor provided in the next chapter.

12 Screen Maker

It's not always smart to track every dot we want on the screen by using clever algorithms or long lists of data statements. The tactics described in the previous chapter work well for plotting sines and cosines but leave a lot to be desired if you want to create a high resolution image of a cityscape or autumnal scene.

Why not allow the computer to calculate the plotting points as we drive a cursor around the screen? We could then save the whole hi-res image so that we could incorporate it into other programs, or simply view it at a later date as a work of art.

We'll pattern our editor after the character and sprite editor and borrow several of the subroutines from the last chapter.

Building the Screen

We'll start by setting up the screen (including a fast wipe of the bit map).

```
100 POKE 53272,PEEK(53272) OR 8 :REM SCREEN AT 8192
110 POKE 53265,PEEK(53265) OR 32:REM TURN ON HIRES
120 SYS 49152
130 FOR I = SCRN TO SCRN+999:POKEI,PC*16+BC:NEXT
140 POKE 53280,PC:REM SET BORDER COLOR
150 RETURN
300 FOR I = 0 TO 20
310 READ T:POKE 49152+I,T
320 NEXT I
330 POKE 251,0:POKE 252,32
340 RETURN
350 DATA 169,00,170,168,145,251,200,240,02,208,249,230,252,
         232,224,32,240
360 DATA 02,208,240,96
```

These lines should look familiar, since they are nearly identical to our screen set-up subroutine from Chapter 11. The only difference here is in line 130, which introduces two variables, BC for background color and PC for pen color. The algorithm PC*16+BC assures that the correct nibbles of screen memory are set for the colors we want. In line 140, the border color is set to the pen color (refer to Fig. 11.1). We'll need to define these and other variables in our initialization section:

```
10 MAP=8192:SCRN=1024
30 PC=0:BC=1:REM PEN COLOR BLACK, BACKGROUND COLOR WHITE
90 GOTO 1000
1000 PRINT "<SHIFT CLR/HOME>"
1010 GOSUB 300
1020 GOSUB 100
```

Plotting

So far, so good. The plotting section looks quite different from its equivalent in the last chapter. This editor will operate much the same as the other editors, moving a cursor around the screen. Therefore, we must know our old X and Y positions (OX and OY) so we can erase the old cursor mark if we are not in drawing mode.

```
500 ROW=INT(OY/8):CHAR=INT(OX/8)
510 LINE=OY AND 7
520 BIT = 7-(OX AND 7)
530 BYTE=MAP+ROW*320+CHAR*8+LINE
535 T=PEEK(BYTE)
540 POKE BYTE,T AND (T-2^BIT)
550 ROW=INT(Y/8):CHAR=INT(X/8)
560 LINE=Y AND 7
570 BIT = 7-(X AND 7)
580 BYTE=MAP+ROW*320+CHAR*8+LINE
585 T=PEEK(BYTE)
590 POKE BYTE,T OR 2^BIT
595 RETURN
```

The plotting routine is duplicated, first for OX,OY and then for X,Y. The new temporary variable, T, was created to clarify the AND operation in line 540. Just as in the other editors, AND is used to restore the previous contents of the byte by erasing the bit that is set at OX,OY. After the old cursor mark is erased by the AND, the new one is made at X,Y by the OR operation in line 590. By this time you should be an old hand at this technique.

The Command Loop

The values of X and Y will be generated by a command loop, located at lines 400-499. First we need to set some constants. BX and BY—big X and big Y, which were used in the other two editors—are not useable here. Recall that Basic uses only two characters of a variable name. We're using BYTE in the plotting subroutine, so BY and BYTE would be perceived by Basic as being the same. BX would work but rather than have inconsistent variable names, we'll rename them XM and YM, for X max and Y max.

```
20 X=0:Y=0:XM=319:YM=199

400 GET A$:IF A$="" THEN 400
410 IF A$="<CRSR><UP>" THEN Y=Y-1
412 IF A$="<CRSR><DOWN>" THEN Y=Y+1
414 IF A$="<CRSR><RIGHT>" THEN X=X+1
416 IF A$="<CRSR><LEFT>" THEN X=X-1
418 IF A$="D" THEN D=NOT D
420 IF Y>YM THEN Y=YM
422 IF Y<0 THEN Y=0
424 IF X>XM THEN X=XM
426 IF X<0 THEN X=0
430 IF NOT D THEN GOSUB 500
440 IF D THEN GOSUB 550
490 OX=X:OY=Y
495 GOTO 400

1030 GOSUB 400
```

Note the subtle change in the section of this subroutine that detects cursor out-of-bounds. In the previous editors, the cursor wrapped around to the other side of the edit grid. This didn't seem like a good idea here because we will use this editor for detail work. If we hadn't adjusted the cursor out-of-bounds, the cursor would suddenly appear at the opposite side of the screen when we were drawing. So this editor **clips,** which is computer graphics jargon for "stops plotting when the value is out-of-bounds."

The only command, aside from the cursor control keys, is D for draw. This is implemented as a Boolean variable that flip-flops between a 0 and 1 every time the D key is pressed. When D is 0, as it is at the start of the program, pressing the D key makes the variable D = NOT D, or −1. The program will now draw. Pressing the D key again makes D = NOT D again and flips it back to a 0. When D is a 0, the IF condition in 430 evaluates as true and all of the plotting subroutine is executed.

Remember that this will cause the old cursor to be erased at OX,OY and a new cursor to be drawn at X,Y. When D is a 1, only the last half of the subroutine is executed—the half that plots at X,Y. The cursor never gets erased, so a point is plotted; the cursor is always one dot ahead of the plotting point. This may not be ideal, but it's simple and it works.

There are several additional commands that seem appropriate. First, the one-dot movement of the cursor was reasonable when we were doing character editing or using the sprite editor, but it doesn't work so well here; it takes too long to traverse the screen. The first set of additional commands either increases or decreases the number of pixels our cursor will move. Edit the following lines to add the required statements:

```
20 X=Ø:Y=Ø:XM=319:YM=199:INC=1

404 IF A$="+" THEN INC=INC+1:GOTO 4ØØ
4Ø5 IF A$="-" THEN INC=INC-1:GOTO 4ØØ
4Ø6 IF INC<1 THEN INC=1

41Ø IF A$="<CRSR><UP>" THEN Y=Y-INC
412 IF A$="<CRSR><DOWN>" THEN Y=Y+INC
414 IF A$="<CRSR><RIGHT>" THEN X=X+INC
416 IF A$="<CRSR><LEFT>" THEN X=X-INC
```

The new variable INC stands for increment and is changed when the plus or minus key is hit. It retains its value whether you are simply moving the cursor or drawing. Some very interesting dotted-line effects are possible. If you want dashed lines, you'll have to separate the X increment from the Y increment. I'll leave that to you.

Plotting Colors

We should also be able to change plotting colors in two ways. The first involves the entire screen—both background and plotting colors. The other involves only the current plotting point. Because we are in regular hi-res mode, the second type of change will also affect the surrounding eight by eight dot grid. We'll use the function keys for these changes.

Although I don't like shifted function keys, in this case, the function of the shifted key is so tied to the nonshifted function that it seems reasonable to use them. Therefore, F1 will change the background color at the current plotting position and F2 (shifted F1) will change the color of the whole screen. Similarly, F3 will change the plotting color at the current X,Y

location and F4 (shifted F3) will change the plotting color of the whole screen.

We must devise a way to select colors. It would be easy to use an INPUT statement to let you type in the color number you wanted. But this seemed a bit crude; it would mean that you would have to memorize the Commodore 64 color numbers or write them down. Why not show the colors? We have a border that is showing the current plotting color. Our color selection subroutine will cycle through the Commodore 64's 16 colors in the border. When the color of choice is shown, pressing <RETURN> will lock it in and continue with the subroutine.

This type of decision, and others such as which command keys to select, come under the general term "human engineering." Programs that are well engineered are said to be "user friendly." If you have used as much software as I have, you soon learn that everyone has his or her own definition of "well engineered." But, you have a big advantage here—you can tailor the programs in this book to your own style. Commercial software should let us do the same.

If you've been following my programming style, you now have an idea of how such tailoring can be accomplished economically. Here, we can write one subroutine to perform all of the color changes, and use flags to indicate what our command choice was.

```
402 IF A$="<SHIFT><F1>" THEN C=BC:GOSUB 600:GOTO 400
403 IF A$="<SHIFT><F3>" THEN C=PC:GOSUB 600:GOTO 400
408 IF A$="<F1>" THEN C=BC:GOSUB 600:GOTO 400
409 IF A$="<F3>" THEN C=PC:GOSUB 600:GOTO 400

600 POKE 53280,C
610 GET T$:IF T$<>" " AND T$<> CHR$(13) THEN 610
620 IF T$=CHR$(13) THEN 650
630 C=C+1:IF C>15 THEN C=0
640 GOTO 600
650 IF A$="<SHIFT><F1>" OR A$="<SHIFT><F3>" THEN 700
652 IF A$="<F3>" THEN TPC=C:TBC=BC:GOTO 660
654 TBC=C:TPC=PC
660 POKE 53280,TPC
670 ROW=INT(Y/8):CHAR=INT(X/8)
680 LOC=SCRN+CHAR+(40*ROW)
690 POKE LOC, TPC*16 + TBC
695 RETURN
700 IF A$="<SHIFT> <F3>" THEN PC=C:GOTO 710
702 BC=C
710 POKE 53280,PC
720 FOR I = SCRN TO SCRN+999:POKE I,PC*16+BC:NEXT
730 RETURN
```

The first section of this subroutine, lines 600–640, is a closed loop to handle the color selection. In the command loop, C is set to the current pen color or border color, whichever you want to change. Line 600 places this color in the border. Then there is a tight GET loop, which will stop only if T$ is equal to a space or to a <RETURN> (which is the same as CHR$(13)).

Next T$ is checked to see if it is a return; if it is, control is branched to line 650, where we'll sort out the various entry points to this subroutine. If T$ is a space, then C is incremented, and checked to make sure it is within the legal 0 to 15 range. Finally, line 640 loops back to 600, starting the process all over again.

Having selected a color by pressing <RETURN> when the desired color is visible on the border, we branch to 650. This line sorts out the shifted function keys from the unshifted ones. Since the initial loop in this subroutine uses T$ rather than A$, our original command is still saved in A$ and can be used as a basis for a branch. If the function key was shifted, control branches to line 700; otherwise, Basic continues to line 652.

Line 652 checks to see if you want to change the background or pen color. If you elect to change the latter, TPC (temporary pen color) is given the value of C, and TBC (temporary background color) is given the value of BC, the global background color. If you want to change the local background color, TPC takes on the value of the current global pen color, and BC becomes the color you selected from the border. Then the current TPC is POKEd into the border.

The next section of the subroutine (lines 660–680) is our old character-position locator that served so well in the character and sprite editors. It does the same thing here. Since the background color and the plot or pen color are both derived from the character screen in regular high resolution mode, this section of the subroutine will locate the byte we must change to alter the background and the pen color. Once located, we need only POKE the new TPC or TBC into that location. The branching back in line 652 assures that the colors are correct. Having done this we can now return to the main command loop.

The other alternative from line 550 is similarly structured. This time, however, we are actually changing the value of PC or BC so we must not toy with temporary variables here. The first decision is whether PC or BC will be changed, and line 700 takes care of this. If the IF test is true, the pen color, PC, will be changed. Otherwise, BC will be given the value selected from the border, C. Then the pen color is POKEd into the border and the long loop to change the values for the whole screen (line 720).

There are some problems with implementing the color commands this way. For one, TPC and TBC do not keep track of their previous values—should you attempt to change them twice within the same eight by eight dot block, they will revert to the global values of PC or BC. However, since it is impossible to have more than one pen color or background color within the block, this may not be so serious.

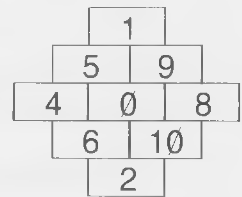
We've created all the basic drawing operations, and the program is functional. But we're not yet done with this editor. There are two more modifications to make. First, we'll implement a joystick drawing routine, since the keyboard may not be the ideal way to create a screen. Then, we'll incorporate the necessary changes to make the editor function in the multi-color high resolution mode.

About Joysticks

To make the following modifications, you must have a joystick attached to the second control port on the right side of your Commodore 64. Paddles will not work, although you might make a project out of converting the joystick routines to work with paddles.

Commodore documentation, however, warns you that reading the paddles from Basic may not be reliable. This is because paddles deliver an analog value—a number that varies from 0 to 255 depending on how the paddle is turned. Commodore joysticks, however, return nine different values depending on the position of the stick. These joysticks are digital, so these values are determined by whether any of four switches are open or closed (see Fig. 12.1 for these values).

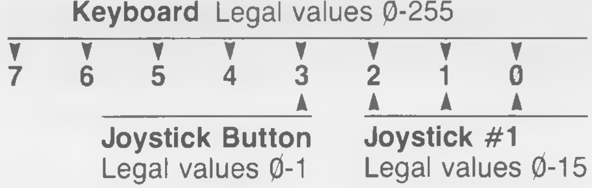
Figure 12.1
Joystick Values



The joystick is controlled, along with the keyboard, by the first CIA (Complex Interface Adaptor) of your Commodore 64. Like the VIC II, the CIA is controlled by POKEing locations in RAM registers located in high memory. The CIA #1 registers begin at location 56320 and extend to location 56335.

The joystick connected to control port 2 is read by PEEKing location 56320 (joystick #1 is controlled by location 56321). The location, 56320, also contains the values generated by the keyboard, but with a difference: the whole byte contains a numeric value related to the key being pressed, while only the lower five bits contain the joystick values. The fifth bit is set either on or off depending on whether the joystick button has been pressed. If pressed, the bit value is 0—off. This is counterintuitive but apparently didn't bother Commodore engineers, so we're stuck with it (see Fig. 12.2).

Figure 12.2
CIA Location 56321 Functions



PEEKing these locations will yield various values depending on which keys and buttons are being pressed, and how the joysticks is positioned. Using joystick 2 will eliminate interference from the keyboard. We must, therefore, isolate the lower four bits of this byte. Ditto for the button. This should look familiar:

```
60 DEF FN JOY(X)=15-(PEEK(56320) AND 15)
70 DEF FN BUT(X)=16-(PEEK(56320) AND 16)
```

The BUT function is fairly straightforward. To isolate the 16s bit (bit number 4), AND it with 00010000. The AND will fail with all bits except 16 since anything ANDed with a 0 is a 0. If the button is pressed, the function should return a 0; if the button is not pressed, the function should return a 16. We'll use the button to flip us in and out of draw mode.

The JOY function requires a little more thought, although it's not terribly difficult to understand. The joystick switches each throw a bit: bit 0 for up, bit 1 for down, bit 2 for left, and bit 3 for right. These switches work just like the buttons; they are a 1 if the switch is not being pressed and a 0 if it is.

Suppose we are pressing the joystick right or east. Only switch 3 will be thrown, giving a byte value of 0001 0111. If this is ANDed with 15 (0000

1111) the resulting value is 0000 0111, isolating the four bits that deal with the position of the stick. But this is 7 and the joystick is supposed to return a value of 8 if the right or east switch is thrown. Subtract the 7 from 15.

Subtraction takes care of Commodore's peculiar notion that a thrown switch should have a value of 0, not a value of 1. If you step through this sequence with any other legal combination of switch values, you'll see that the algorithm works. Some switch combinations make no sense—right and left, for example.

Drawing with Joysticks

Now that we have our joystick and button functions, let's integrate them into the command loop.

```
400 J=FN JOY(0):IF J=0 THEN GET A$:IF A$="" THEN 400
410 IF J=1 THEN Y=Y-INC:REM UP
411 IF J=9 THEN Y=Y-INC:X=X+INC:REM UP + RIGHT
412 IF J=2 THEN Y=Y+INC:REM DOWN
413 IF J=10 THEN Y=Y+INC:X=X+INC:REM DOWN + RIGHT
414 IF J=8 THEN X=X+INC:REM RIGHT
415 IF J=5 THEN Y=Y-INC:X=X-INC:REM UP + RIGHT
416 IF J=4 THEN X=X-INC:REM LEFT
417 IF J=6 THE Y=Y+1:X=X-1:REM DOWN + LEFT
418 IF FN BUT(0) THEN D=NOT D
```

And that's it. The program will now plot with the joystick. The added benefit of this approach is diagonal movement. We could have incorporated this in the keyboard version, but since the Commodore 64 does not have diagonal CRSR keys, I decided against fabricating new key functions.

I think you'll find the joystick version of this program much more fun than the keyboard version. Now for our final modification: multicolor.

Multicolor Modifications: Color

Changing the editor to function in multicolor mode is more complicated than incorporating a joystick. The main problem is in how the colors are generated. Recall that in multicolor mode the color of a particular pixel is determined by a pair of bits rather than by one bit. Each combination of the pair yields a different color: 00, 01, 10, and 11. The plotting routine must be modified to POKE one of these pairs rather than just flipping a bit.

Which of these pairs should be plotted? That depends on which color is currently selected, so our first step is to devise a way to allow colors to be selected. The selected color determines which bit pair to plot. Before we can write the command line, we need to decide how to store the numbers. Some methods of storing color values require a more programming than others.

Program listings in books and magazines often seem so technically polished, but I've opted here for clarity over polish. Even so, the listings may look imposing. But remember that these programs are the result of a lot of trial and error and hours of debugging. As you gain experience, most of this takes place before you actually start to write the program.

The clearest way to hold color choices is in an array. Each of the four color choices will have a value in the array. A new variable, CC, holds the color currently being used for drawing. But CC does not hold the actual color values; it is an index into the array. This method lets one statement assure that the border color reflects the pen color rather than four different statements. Compare:

```
POKE 53281,C(CC)
```

with

```
IF CC=1 THEN POKE 53281,C1
IF CC=2 THEN POKE 53281,C2
IF CC=3 THEN POKE 53281,C3
IF CC=4 THEN POKE 53281,C4
```

The second alternative is possible but not nearly as efficient or clear as the first. We now have enough information to write our color selection command and part of the initialization section.

Since you've already entered much of the code, start by loading either the regular version or the joystick version. The modifications will work with both.

```
10 MAP=8192:SCRN=1024:CRAM=55296
```

We must add CRAM—Color RAM—here to work with color 4 (bit pattern 11).

```
20 X=0:Y=0:XM=319:YM=199:INC=2
```

INC now becomes 2 since we must always plot pairs of pixels. This is only true in the X direction, so we'll also do a small rewrite in the command loop where INC is used.

```
50 C(1)=BC:C(2)=2:C(3)=6:C(4)=PC:CC=4
```

This statement initializes the array C, which holds the color values. Color 1 is black and is also the background, color 2 is red, color 3 is blue, and color 4 is white. The initial color (CC) is set to color 4. As in the regular hi-res editor, we'll be able to change these colors as we draw.

The subroutine that initializes the screen also needs some changing:

```
115 POKE 53270,PEEK(53270) OR 16:REM MULTICOLOR MODE
130 FOR I = SCRN TO SCRN+999:POKE I,C(2)*16+C(3):NEXT
135 FOR I = CRAM TO CRAM+999:POKE I,C(4):NEXT
145 POKE 53281,C(1)
```

Lines 130, 135, and 145 initialize our chosen colors. Line 130 handles both colors 2 and 3 using an algorithm to assure that the numbers correspond to those in Fig. 11.1. Line 135 POKEs color 4 into CRAM and line 145 puts the background color into the background color register.

Now we must modify the input loop to accommodate our new value for INC, which now represents movement only in the X direction. We will forego increments in the Y direction.

```
404 IF A$="+" THEN INC=INC+2:GOTO 400
405 IF A$="-" THEN INC=INC-2:GOTO 400
406 IF INC<2 THEN INC=2
```

The main reason INC was used in the original editor was that we need it in this multicolor modification. Without INC, we could not guarantee that our plotting would correspond to Commodore's multicolor pixels. This finishes off the INC modifications:

```
410 IF A$="<CRSR><UP>" THEN Y=Y-1
412 IF A$="<CRSR><DOWN>" THEN Y=Y+1
414 IF A$="<CRSR><RIGHT>" THEN X=X+INC
416 IF A$="<CRSR><LEFT>" THEN X=X-INC
```

Lines 414 and 416 are the same as before, but now we change Y by 1 rather than INC.

Finally, we come to the line that actually allows color selection:

```
419 IF A$=>"1" AND A$<"5" THEN CC=VAL(A$):POKE 53280,C(CC):
    GOTO 400
```

The AND in this IF statement assures that A$ is between 1 and 4. If true, CC is changed to the number typed, and the border register is POKEd with the value held in the color array. This way we always know the color our pen will draw in.

Multicolor Modifications: Plotting

We must now face the serious problem of how to plot the X and Y coordinates that our main command loop will give us. The X coordinate will be in the usual range of 0 to 319, but we can only plot 160 multicolored pixels because each pixel requires two bits.

You may think the solution is to divide X by 2, but this won't help—it will only restrict the plotting to half the screen. There are still 320 bits across the screen. By changing INC, we are assured of jumping across the screen in two-pixel moves. We must find a way to get our plotting routine to plot the correct bit arrangement. Once again, Boolean functions will come to our rescue.

We've done a lot of PEEKing and POKEing with Boolean functions, so you should be moderately comfortable with them. The cursor location is only part of the information needed to plot the pixel; we also must know the color. The color, not the location, determines what bits get flipped on.

This is different from the previous editor and it may seem that we need a whole new plotting routine. But we really don't. BIT, the variable that holds the pixel to be plotted in the normal hi-res editor, must be converted to two bits. Since the cursor is moving in two-bit increments, we want BIT to hold the current bit (at 7 − X AND 7) as well as its neighbor.

In binary, it's easy to calculate the neighboring bit—BIT/2. This is important, so let's look at an example. If the cursor is at 0,10, BIT would be 128 (7 − 0 AND 7 = 128). In binary notation, 128 is the seventh bit; the neighboring bit, the sixth, is 64. After all, 128/2 is 64. Try this for other bit combinations and you'll find that the algorithm works each time.

Now that we can get our two-bit pixel from the current X location, we need to somehow introduce color into the routine. After all, it is the color that determines what the actual status of the bits will be. All we've done so far is figure out which bits need to be changed.

As it stands, these bits would plot two pixels because the algorithm assures that the bits corresponding to the current X position are on, or set to a 1. We need to filter these to represent the current color status. An easy way to do this is with masks. Each color possibility requires a mask (see Fig. 12.3).

Figure 12.3
Color Masks

Mask	Decimal	Color
ØØØØØØØØ	Ø	1 (Background)
Ø1Ø1Ø1Ø1	85	2
1Ø1Ø1Ø1Ø	17Ø	3
11111111	255	4

Let's return to the example in which a pixel was located at 0,10. We've already figured out how to make BIT equal to 192 (128 + 64). In binary, 194 is 11000000. Fig. 12.4 shows the effects of ANDing this with each of the masks.

Figure 12.4
The Effects of ANDing 194 with the Color Masks

	11000000		11000000		11000000		11000000
AND	00000000	**AND**	01010101	**AND**	10101010	**AND**	11111111
	00000000		01000000		10000000		11000000

This isn't magic; it's just pure, cold Boolean logic. The resulting values represent the bit pattern we need to POKE to get the right color on the screen given a multicolor pixel at 0,10. Actually, the Y is irrelevant and it can be calculated exactly the same way as in the previous hi-res editors. Now all we have to do is get the bit pattern onto the screen without disturbing the other bits.

This is accomplished by the simple OR statement we used previously, but a problem remains: if we are tracing over a previously plotted point, nothing happens. The OR does exactly what it should and leaves points already on the screen intact. So we must first retrieve what is on the screen and unset the correct bits—make them 00s—before we can OR them onto the screen.

Here is the revised plotting subroutine:

```
55Ø ROW=INT(Y/8):CHAR=INT(X/8)
56Ø LINE=Y AND 7
57Ø BIT = 7-(X AND 7):BIT=(2^BIT/2)+(2^BIT):REM DOUBLE UP
58Ø BYTE=MAP+ROW*32Ø+CHAR*8+LINE
585 T=PEEK(BYTE):T=T AND (255-BIT):REM CONVERT 1s TO Øs
59Ø POKE BYTE,T OR (BIT AND M(CC)):REM PUT ON SCREEN
595 RETURN
```

A Sprite Cursor

Notice that the cursor erase routine, lines 500-540, is missing. Since multicolor is more difficult to deal with in terms of bits, a different type of cursor is called for here. And it's better to build on what you've already learned. The Commodore 64 provides an ideal graphics cursor—a sprite. Sprites don't affect bits you've already set, or destroy colors already displayed. They just glide over the screen.

I've chosen a cross-hair cursor, but if you'd prefer a different shape, change it. A small paint brush might be appropriate. Very few changes are required to add a sprite cursor. Sprite X and Y coordinates don't correspond directly to hi-res coordinates, so we need offsets to put the sprite where the hi-res X and Y are. We also must turn on sprites and read a sprite bit map into memory:

```
80 SXO=20:SYO=46:REM SPRITE OFFSETS

150 POKE 53269,PEEK(53269) OR 1:REM TURN ON SPRITE 1
160 POKE 53248,SXO:POKE 53249,SYO:REM STARTING POSITIONS
170 RETURN

200 FOR I = 0 TO 63
210 READ T:POKE 832+I,T
220 NEXT
230 POKE 2040,13:REM BIT MAP LOCATION VIC II PAGE 13
235 RETURN
240 DATA 8,0,0,8,0,0,8,0,0,8,0,0
250 DATA 247,128,0,8,0,0,8,0,0,8,0,0
260 DATA 8,0,0,0,0,0,0,0,0,0,0,0
270 DATA 0,0,0,0,0,0,0,0,0,0,0,0
280 DATA 0,0,0,0,0,0,0,0,0,0,0,0
290 DATA 0,0,0,0

430 GOSUB 500
440 REM

500 IF (X+SXO)>255 THEN POKE 53264,1:XS=X-255:GOTO 520
510 POKE 53264,0:XS=X
520 POKE 53248,XS+SXO
530 POKE 53249,Y+SYO
540 IF NOT D THEN RETURN

1010 GOSUB 200:GOSUB 300:REM SET UP SPRITES & QUICK CLEAR
```

That's it. The use of a sprite cursor required a small change to the drawing routine to show the cursor at all times, not just when we are not drawing. Line 430 had to be changed and line 440 removed. Actually, this logic could have been applied to the previous editors, but it's required here.

Color Changes

The final problem deals with color. As it now stands, the editor will draw, but color changes aren't recorded because we have never changed C( ). The color subroutines change only PC and BC.

```
402 IF A$="<SHIFT><F1>" OR A$="<SHIFT><F3>" OR A$="<F1>"
    OR A$="<F3>"THEN C=C(CC):GOSUB 600:GOTO 400
403 <RETURN>
408 <RETURN>
409 <RETURN>

600 POKE 53280,C
610 GET T$:IF T$<>" " AND T$<> CHR$(13) THEN 610
620 IF T$=CHR$(13) THEN 650
630 C=C+1:IF C>15 THEN C=0
640 GOTO 600
650 IF A$="<SHIFT><F1>" OR A$="<SHIFT><F3>" THEN 700
660 REM
670 ROW=INT(Y/8):CHAR=INT(X/8)
680 ON CC GOTO 682,684,686,688
682 POKE 53280,C:C(1)=C:GOTO 690:REM NO LOCAL CHANGE FOR 4
684 LOC=SCRN+CHAR+(40*ROW):POKE LOC,C*16+C(3):GOTO 690
686 LOC=CRAM+CHAR+(40*ROW):POKE LOC,C(2)*16+2:GOTO 690
688 LOC=CRAM+CHAR+(40*ROW):POKE LOC,C(4)
690 RETURN
700 C(CC)=C
710 ON CC GOTO 712,714,714,718
712 POKE 53280,C:GOTO 720
714 FOR I = SCRN TO SCRN+999:POKE I,C(2)*16+C(3):NEXT:GOTO 720
718 FOR I = CRAM TO CRAM+999:POKE I,C(4)
720 RETURN
```

Like the cursor subroutine, the new color subroutines are simpler. Essentially, the four colors are equivalent; we must POKE only the new values into the right locations. Because we are always altering C(CC), we can eliminate the extra function key statements in the command loop. Note that the overall structure of the subroutine is the same as before. The shifted function keys will make global changes, while the unshifted keys will make only local changes.

You should recognize the FOR/NEXT loops (as in 714). These are similar to the loops used in the character graphics sections. They should be, because the colors in multicolor hi-res mode are derived from the character screen and from color RAM.

There are two limitations to these routines as written. You cannot make a local change to the background color. Both the shifted and unshifted color

function keys work identically for color 1, because the background color is derived from the background color register only. This limitation is built into Commodore's multicolor hi-res mode.

You can easily get around the other limitation. When a local change is made to plotting colors 2 or 3, that change is immediately reflected on the screen in the correct eight by eight pixel box. So far so good, but when you try to change the other color, the first color will revert to the current global color (C( )). If this is a problem for you, it can be fixed by saving the colors in temporary variables as we did in the regular hi-res editor.

This is really only the beginning of a high resolution graphics editor. There are many useful functions that allow you to incorporate text on the graphics screen, enlarge portions of the drawing, provide box, frame, circles, or disk functions, and so on. Some of these functions could be added to our editor in Basic, but most require assembly language routines to accomplish the function with reasonable speed. Should you require a more complex editor, several commercial products may come to your rescue; see Chapter 16 for details.

```
5 REM SCREEN MAKER - MULTICOLOR JOYSTICK VERSION
10 MAP=8192:SCRN=1024:CRAM=55296
20 X=0:Y=0:XM=319:YM=199:INC=2
30 PC=0:BC=1
40 C(1)=BC:C(2)=2:C(3)=6:C(4)=PC:CC=4
50 M(1)=0:M(2)=85:M(3)=170:M(4)=255:REM COLOR MASKS
60 DEF FN JOY(X)=15-(PEEK(56320) AND 15)
70 DEF FN BUT(X)=16-(PEEK(56320) AND 16)
80 SXO=20:SYO=46:REM SPRITE OFFSETS
90 GOTO 1000
100 POKE 53272,PEEK(53272) OR 8 :REM SCREEN AT 8192
110 POKE 53265,PEEK(53265) OR 32:REM TURN ON HIRES
115 POKE 53270,PEEK(53270) OR 16:REM ENTER MULTICOLOR MODE
120 SYS 49152
130 FOR I = SCRN TO SCRN+999:POKEI,C(2)*16+C(3):NEXT
135 FOR I = CRAM TO CRAM+999:POKEI,C(4):NEXT
140 POKE 53280,PC:REM SET BORDER COLOR
145 POKE 53281,C(1)
150 POKE 53269,PEEK(53269) OR 1:REM TURN ON SPRITE 1
160 POKE 53248,SXO:POKE 53249,SYO:REM STARTING POSITIONS
170 RETURN
200 FOR I = 0 TO 63
210 READ T:POKE 832+I,T
220 NEXT
230 POKE 2040,13:REM BIT MAP LOCATION VIC II PAGE 13
235 RETURN
240 DATA 8,0,0,8,0,0,8,0,0,8,0,0
250 DATA 247,128,0,8,0,0,8,0,0,8,0,0
260 DATA 8,0,0,0,0,0,0,0,0,0,0,0
```

```
270 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
280 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
290 DATA Ø,Ø,Ø,Ø
300 FOR I = Ø TO 20
310 READ T:POKE 49152+I,T
320 NEXT I
330 POKE 251,Ø:POKE 252,32
340 RETURN
350 DATA 169,ØØ,170,168,145,251,200,240,Ø2,208,249,230,252,
         232,224,32,240
360 DATA Ø2,208,240,96
400 J=FN JOY(Ø):IF J=Ø THEN GET A$:IF A$="" THEN 4ØØ
402 IF A$="<SHIFT><F1>" OR A$="<SHIFT><F3>" OR A$="<F1>"
    OR A$="<F3>"THEN C=C(CC):GOSUB 6ØØ:GOTO 4ØØ
404 IF A$="+" THEN INC=INC+2:GOTO 4ØØ
405 IF A$="-" THEN INC=INC-2:GOTO 4ØØ
406 IF INC<2 THEN INC=2
41Ø IF J=1 THEN Y=Y-INC:REM UP
411 IF J=9 THEN Y=Y-INC:X=X+INC:REM UP + RIGHT
412 IF J=2 THEN Y=Y+INC:REM DOWN
413 IF J=1Ø THEN Y=Y+INC:X=X+INC:REM DOWN + RIGHT
414 IF J=8 THEN X=X+INC:REM RIGHT
415 IF J=5 THEN Y=Y-INC:X=X-INC:REM UP + RIGHT
416 IF J=4 THEN X=X-INC:REM LEFT
417 IF J=6 THEN Y=Y+1:X=X-INC:REM DOWN + LEFT
418 IF FN BUT(Ø) THEN D=NOT D
419 IF A$=>"1" AND A$<"5" THEN CC=VAL(A$):POKE 5328Ø,C(CC):
    GOTO 4ØØ
42Ø IF Y>YM THEN Y=YM
422 IF Y<Ø THEN Y=Ø
424 IF X>XM THEN X=XM
426 IF X<Ø THEN X=Ø
43Ø GOSUB 5ØØ
44Ø REM
49Ø REM
495 GOTO 4ØØ
5ØØ IF X+SXO>255 THEN POKE 53264,1:XS=X-255:GOTO 52Ø
51Ø POKE 53264,Ø:XS=X
52Ø POKE 53248,XS+SXO
53Ø POKE 53249,Y+SYO
54Ø IF NOT D THEN RETURN
55Ø ROW=INT(Y/8):CHAR=INT(X/8)
56Ø LINE=Y AND 7
57Ø BIT = 7-(X AND 7):BIT=(2^BIT/2)+(2^BIT)
58Ø BYTE=MAP+ROW*32Ø+CHAR*8+LINE
585 T=PEEK(BYTE):T=T AND (255-BIT)
59Ø POKE BYTE,T OR (BIT AND M(CC))
595 RETURN
6ØØ POKE 5328Ø,C
61Ø GET T$:IF T$<>" " AND T$<> CHR$(13) THEN 61Ø
62Ø IF T$=CHR$(13) THEN 65Ø
63Ø C=C+1:IF C>15 THEN C=Ø
64Ø GOTO 6ØØ
65Ø IF A$="<SHIFT><F1>" OR A$="<SHIFT><F3>" THEN 7ØØ
```

```
660 REM
670 ROW=INT(Y/8):CHAR=INT(X/8)
680 ON CC GOTO 682,684,686,688
682 POKE 53281,C:C(1)=C:GOTO 690:REM NO LOCAL CHANGE FOR 4
684 LOC=SCRN+CHAR+(40*ROW):POKE LOC,C*16+C(3):GOTO 690
686 LOC=CRAM+CHAR+(40*ROW):POKE LOC,C(2)*16+2:GOTO 690
688 LOC=CRAM+CHAR+(40*ROW):POKE LOC,C(4)
690 RETURN
700 C(CC)=C
710 ON CC GOTO 712,714,714,718
712 POKE 53281,C:GOTO 720
714 FOR I = SCRN TO SCRN+999:POKE I,C(2)*16+C(3):NEXT:GOTO
    720
718 FOR I = CRAM TO CRAM+999:POKE I,C(4)
720 RETURN
1000 PRINT "<SHIFT CLR/HOME>"
1010 GOSUB 200:GOSUB 300:REM SET UP SPRITES & QUICK CLEAR
1020 GOSUB 100:REM SET UP SCREEN
1030 GOSUB 400:REM COMMAND LOOP
```

13 Odds and Ends

You now have a set of very useful editors that will allow you to create custom character sets, sprites, and high resolution screens. But until now, you have had no way to use these outside of the editors within your own program. These editors all create a block of binary data in the computer's memory, but we need to create a routine that will allow us to load and save these data blocks from any program.

Many microcomputers feature BSAVE and BLOAD commands, which allow programmers to save or load data anywhere in the computer's memory. The Basic SAVE and LOAD commands on the Commodore are really a specialized form of BSAVE and BLOAD. Basic always knows where your program starts and how long it is.

When you save a program, Basic saves only a "snapshot" of memory, starting at the beginning of the program (normally 2048) and extending for its length. If we could find some way to change the pointers that tell Basic where the program starts and how long it is, the SAVE or LOAD would then work with the new location. This is how the PET and VIC loader programs for the Commodore 64 work.

This idea is fine but in practice it would not work. Basic would forget where the original program is and try to execute the newly loaded graphic data as a Basic program. It would behave similarly with a SAVE. The data might be a pretty picture or an elegant font but as a program it is garbage. Your Commodore 64 would probably lock up. To accomplish this, we must work in another language, one that can reside with Basic in memory. There's only one such language: machine language. Using the SYS command we can direct the computer to execute a machine language program and then return to Basic.

Don't panic. Machine language is no more difficult to learn than Basic—it simply requires more programming. One of the advantages of Basic (or the kernal) is that it does much of the hard work for you. Take a simple PRINT

statement. You need only say PRINT "HELLO", and you'll see HELLO on the screen. Basic must set up a loop to take the data character by character, find where on the screen a character should be placed, put it on the screen, tell the computer that it should update the cursor since one character has been printed, loop back, and get the next character. Basic is called a high-level language because you can spend more time thinking about what you want to print than about how to get the nitty-gritty details ironed out.

At the Heart of the Computer

Computers, at their hearts, can think only in terms of ones and zeroes. Machine language, therefore, is binary, and its commands are only numbers. But people don't like dealing purely with numbers, especially when context is important. For example, a person may confuse the ASCII code for the letter A with the machine language instruction EOR (Exclusive OR)—both are the number 65. The possibilities are endless. It is how the 65 is placed into memory and its relationship with other adjacent numbers that determine what the computer will do with it.

In the old days of computing, programmers had to deal with these binary numbers. Fortunately, it wasn't long before someone realized that this kind of head-scratching, number-shuffling stuff that had to be done to program a computer in binary is the ideal task for a computer to do itself, and assembly language was born.

Assembly language, like Basic, is not binary. There is one assembly language instruction for each category of machine language instruction. The abbreviation EOR, used in the example above, is easier to remember than 65. Most of the other instructions are equally easy to remember: ADC, add with carry; JMP, jump (like Basic's GOTO); BPL, branch on result plus (like Basic's IF/THEN); DEC, decrement memory (like A = A − 1); and so on.

Using Assembler

The program must be written with a text editor. Recall that the Commodore 64 runs Basic unless you specifically instruct it otherwise; assembly language commands would generate only syntax errors in Basic. After entering the program into a text editor, it is submitted to an assembler. The assembler takes all the DECs and BPLs and converts them to binary—the form the computer can understand.

This process is similar to what happens when you press <RETURN> after entering a line of Basic code; while Basic does not convert the line to machine language, it does convert it to a more memory-saving form. After assembly, the machine code can then be saved. If the program is intended to be a time-saving subroutine for a Basic program, it can be executed with a SYS command. Modern assemblers are quite complicated programs that relieve the programmer of many tedious tasks.

Debugging an assembly language program is quite a bit more involved than fixing a Basic program. This is because it is necessary to go back to the text editor to alter the assembly language code, then rerun the assembler to produce, with any luck, a working program. But, as we've seen with the hi-res bit-map clear routine, assembly language is very fast and it is the only way to milk your computer for all its capabilities.

It is interesting that many of the things we had to do with complicated algorithms in Basic can be very easily accomplished in assembly language. The BLOAD and BSAVE routines that follow are examples of operations that just can't be done in Basic.

The ultimate result of the routines that follow will be a subroutine that can be incorporated into your editors or programs to let you load in your pictures or character sets. But for now all we will do is go through the writing of the routine in assembly language. This should demystify the process and illustrate how you can use the Commodore 64 kernal routines to ease your entry into the strange and mysterious world of assembly language. If you'd prefer to bypass this section for now, just enter the finished subroutine that appears at the end of this chapter.

Registers

The 6510 (and 6502) microprocessors have four special memory locations that are not part of the 65535 locations in the total address space of the computer. These locations contain data that is being manipulated at the current moment. They are called the accumulator (or the A register), the X register, the Y register, and the status register.

Nearly everything the computer does happens through these registers. In order to make the kernal work for us, we must enter very specific information into these registers. Once the registers are correctly set up, the kernal will do our loads or saves.

The X, Y, and A registers are used like Basic temporary variables. All logical and arithmetic operations must occur through these registers.

The status register deserves special comment. Unlike the X, Y, and A

registers, the status register is automatically changed by the 6510 when certain events happen in the other registers. These events can include a value overflow—an addition that results in a number bigger than 255—or the results of a comparison. The status register is used to provide assembler's version of IF/THEN branching. Without it, assembly language programming would be difficult, if not impossible.

Op Codes

We'll need a small collection of assembly language instructions, also called op codes, to accomplish our program. As in Basic, most programs don't use all available instructions. Fig. 13.1 lists the major instructions we'll need.

Figure 13.1
Partial List of 6510 Assembly Instructions (Op Codes)

STA	Store Accumulator
LDA	LoaD Accumulator
LDX	LoaD X register
LDY	LoaD Y register
JSR	Jump to SubRoutine
RTS	ReTurn from Subroutine
BCC	Branch if Carry is Clear

Notice that the several of the instructions deal with loading a register. This should come as no surprise, becasue loading registers is the major task when using kernal routines. The question is where do we put the data so that it can be stored in the registers?

The 6510 is one of the most flexible microprocessors in this regard. There are at least eight ways to tell the 6510 where the number to load the accumulator (LDA) is stored. Not every instruction is this flexible. These methods of finding data, or addressing, are indicated by how we enter the values into the assembler. The syntax here has nothing to do with the 6510, only with the assembler. The assembler must choose among the legal techniques and decide which you want. From your assembly command, the assembler will produce the one-byte machine language instruction that will actually direct the microprocessor.

For our program we only need two methods. One instructs the computer that the next byte in the program is data. In Basic, this instruction is equivalent to LET X=10. The 10 is imbedded in the program, but Basic doesn't get confused; it knows that the 10 is data thanks to the LET

command. In fact, it knows this even when the LET is not there! The assembler syntax for immediate addressing (as this technique is called) is:

```
LDA #0
```

which would put a 0 in the accumulator.

The second technique is to tell the computer where the data is in memory by giving it an address. This is very much like LET A = B in Basic. Basic looks up the value it has for B and then assigns it to A. The computer does the same in absolute addressing, as this technique is called. It uses the two bytes following the op code as an address and finds the data at that address. The syntax for this is:

```
LDA 1024
```

where 1234 represents the byte of memory 1,234 bytes from the start of memory.

Addresses and Labels

Addresses are dealt with differently in assembly language from the way they are handled in Basic. Because a byte can hold a number only up to 255, two bytes are required to store an address. When sending an assembly language program an address from Basic, it must be divided into two numbers. The algorithms are:

```
LOBYTE=ADR-256*INT(ADR/256) and
HIBYTE=ADR/256
```

The hibyte is the page of 6510 memory, and the low byte is the actual address within that page. 1,024, the first byte of screen memory, becomes 0 and 4. This is to be expected because 1024 is the first byte on a new page of memory, the fourth page to be precise. The 6510 expects to find the low byte first, and then the high byte.

Fortunately, a good assembler allows us to give symbolic names to addresses and it takes care of putting it into the correct order. Assemblers can even be told that a symbol represents an address, and they will even chop it up for you. To do this, the usual assembler syntax is:

```
JUNK EQU 1024        assign a value to JUNK
     LDA   #<JUNK    load A with low byte of the address
     LDA   #>JUNK    load A with high byte of the address
```

So now we know how to get data and addresses into our assembly language program. There's another service assemblers can perform for you. In Basic, we can GOTO a line number; in assembler, we can jump or branch to a label. A label is like a line number, but it's made of letters rather than just numbers.

Because the assembler will look at the whole program and translate it for you, it's simple to use labels. The assembler keeps track of where the labels are. In this regard, assembly language is easier to use than Basic, with its tangles of line numbers.

Labels are generated in one of two ways. The assembler command EQU will set a label equal to a number. We saw this in the example above. Labels can also be used like line numbers in Basic, to provide convenient branching. The assembler will keep track of the name, just as Basic keeps track of your line numbers. This usage looks like:

```
JUNK LDA #0
```

Any further reference to JUNK would generate the address of this LDA. Conceptually, Basic does the same when it executes a GOTO 100—Basic finds where in memory line 100 is stored and transfers program control there.

The assembler translates the whole thing into a stream of binary values, where addresses are addresses and data are data. You cannot tell what the source, or text, code looked like from the object, or executable, code produced by an assembler. Labels, whether for variable assignment or branching, are for the programmer's convenience.

Branching in Assembler

Finally, we'll need a branch instruction, and here we will use the status register. For example, there is an INY intruction that increments the Y register by one—like Y = Y + 1 in Basic. But the Y register is only a byte big so what happens when it reaches 255 and is incremented again? It "rolls over" to zero like an odometer in a car. This rather important event is relayed to us by setting a bit in the status register. Our program will check one of these bits, for example, the carry bit, and branch (or GOTO) another part of the program if that bit is set. The assembler code for this example is BCS—Branch Carry Set.

The status register takes on further importance as the kernal sets certain bits when errors occur in its activities. By examinng the carry bit, we can determine if an error has occurred in our loads or saves.

Kernal Routines

Now we must find out how much work we have to do and how much we can rely on the kernal to do. *The Commodore 64 Programmer's Reference Guide* contains a section on useful kernal routines; these are a relief when you're programming in assembly language.

We will be opening a file, so we'll use two file set-up routines. The first, called SETLFS, will set up the logical file number. The second, SETNAM, will set up the file name. After these, we can use the kernal SAVE routine, which will do the hard work—open the file, save a chunk of memory, and close the file—all with one simple JSR (jump to subroutine). The register requirements are clearly described and we have the tools in assembly language to assure that the registers are working. Without these kernal subroutines, our programs would grow much longer.

Storage

What we don't have yet is a way for pertinent data from our Basic program to get into the assembler program. It would be foolish, not to mention wasteful, to imbed absolute addresses into the assembler code—one for sprite save, one for character set save, and so on. We need a more general-purpose way of getting the information across the langauge gap. The following information must be conveyed: the logical file number, the device (disk or tape), the starting memory location, the ending memory location, and the file name. This is at most 22 bytes, less with a file name shorter than 16 characters. Fortunately, there is a place in memory to POKE these data—locations 679 to 767 are unused.

We'll POKE the required information starting at 679 from Basic and then call the BSAVE subroutine. It will get the information to set the registers from our POKEs. There are fancier ways of doing this, but for our purposes this is clear and uncomplicated. The information area will be structured as shown in Fig. 13.2.

Figure 13.2
Storage for Assembler BSAVE and BLOAD Programs

Location	Contents
679	Device Number: 1 for Tape or 8 for Disk
680	Logical File Number (shouldn't already be in use)
681 & 682	Address of start of save in low byte/high byte format
683 & 684	Address of end of save in low byte/high byte format
685	Length of file name
686 & on	File name (ASCII characters) Max length 16

BSAVE

We can now set up most of our program. The following listing looks just as it would when entered into an assembler. Each line is divided into four fields: label, op code (or instruction), data (or operand), and comment. Only the op code field is required, the others may or may not be used depending on the particular instruction.

```
SAVE      EQU    65496     ;Address of kernal SAVE
SETLFS    EQU    65466     ;Address of SETLFS routine
SETNAM    EQU    65469     ;Address of SETNAM routine
INFO      EQU    679       ;Address of our data section
DEV_NO    EQU    INFO      ;Address of the device number
FILE_NO   EQU    INFO+1    ;Address of the logal file no.
START     EQU    INFO+2    ;Address of start of save
END       EQU    INFO+4    ;Address of end of save
LENGTH    EQU    INFO+6    ;Address of file name length
NAME      EQU    INFO+7    ;Address of file name
TEMP      EQU    251       ;Address used for temporary
                            storage
```

I've used more labels than most assembly language programmers would, but since there is no penalty—except in the size of the source code file—why not? If this is your first look at an assembly language program, it might seem strange, but it shows that assembly language is not only for the superprogrammers but for all of use who need extra power or speed.

```
LDX      DEV_NO     ;Get device no. into X register
LDA      FILE_NO    ;Get file no. into accumulator
LDY      #0         ;Set up Y with device command
JSR      SETLFS     ;GOSUB to SETLFS
```

```
            LDA     LENGTH      ;Get file name length
            LDX     #<NAME      ;Get low byte of name's address
            LDY     #>NAME      ;Get high byte of name's
                                 address
            JSR     SETNAM      ;GOSUB to SETNAM

            LDA     START       ;Get low byte of start
            STA     TEMP        ;Store it in TEMP
            LDA     START+1     ;Get high byte of start
            STA     TEMP+1      ;Store it in TEMP+1

            LDX     END         ;Get low byte of end into X
            LDY     END+1       ;Get high byte of end into Y
            LDA     #TEMP       ;Get low byte of TEMP into A

            JSR     SAVE        ;GOSUB to SAVE

            BCS     RPT_ERR     ;If carry bit of status
                                ;register is set, branch to
                                ;RPT_ERR -- error number is in
                                ;the accumulator
            LDA     #0          ;If no error, A contains last
                                ;data byte. This will clear
                                ;it.
RPT_ERR     STA     TEMP        ;Store A in TEMP.
            RTS                 ;Return to Basic
```

BLOAD

BSAVE's companion program, BLOAD, accomplishes the reverse function. Actually, is is possible to do a BLOAD directly in Basic. The syntax

```
LOAD "my data",8,1 for disk
LOAD "my data",1,1 for tape
```

will load the file to where it was saved from; the ",1" accomplishes this. However, you may not always want the data to be located where it was originally. You may, for example, want to use the sprite editor to create 16 sprites and load them into consecutive memory pages. With the Basic LOAD command, the second set of sprites would overlay the first because the editor uses only eight pages of sprite memory. A more flexible loader is necessary. It's remarkably like SAVE:

```
LOAD        EQU     65493       ;Address of kernal LOAD
SETLFS      EQU     65466       ;Address of SETLFS routine
SETNAM      EQU     65469       ;Address of SETNAM routine
INFO        EQU     679         ;Address of our data section
DEV_NO      EQU     INFO        ;Address of the device number
```

```
FILE_NO    EQU    INFO+1     ;Address of the logal file no.
START      EQU    INFO+2     ;Address of start of load
LENGTH     EQU    INFO+6     ;Address of file name length
NAME       EQU    INFO+7     ;Address of file name
TEMP       EQU    251        ;Address used for temporary
                              storage

           LDX    DEV_NO     ;Get device no. into X register
           LDA    FILE_NO    ;Get file no. into accumulator
           LDY    #Ø         ;Set up Y with device command
           JSR    SETLFS     ;GOSUB to SETLFS

           LDA    LENGTH     ;Get file name length
           LDX    #<NAME     ;Get low byte of name's address
           LDY    #>NAME     ;Get high byte of name's
                              address
           JSR    SETNAM     ;GOSUB to SETNAM

           LDX    START      ;Get low byte of start into X
           LDY    START+1    ;Get high byte of start into Y
           LDA    #Ø         ;Set up A for load

           JSR    LOAD       ;GOSUB to LOAD
           BCS    RPT_ERR    ;If carry bit of status
                             ;register is set, branch to
                             ;RPT_ERR -- error number is in
                             ;the accumulator
           LDA    #Ø         ;If no error, A contains last
                             ;data byte. This will clear
                             ;it.
RPT_ERR    STA    TEMP       ;Store A in TEMP.
           RTS               ;Return to Basic
```

Since the length of the file is stored with the file, we need not deal with our old end value—two fewer POKEs for us from Basic. Otherwise the routine is very similar. The SETLFS and SETNAM calls are identical. For a LOAD, the kernal requires the X and Y registers to contain the address at which the data is to start, and that the accumulator contains a zero. That's it—the kernal does all the hard work.

To enter these subroutines into our programs, we will write Basic subroutines that POKE the machine language into the Commodore 64 memory. Once installed, they will be called with the SYS instruction after having POKEd our data section (at byte 679 and following).

You must be extra careful in debugging because these programs work on your mass storage device; a minor typing error could cause the computer to wipe out a disk or cassette. I speak from experience. Use a freshly formatted disk that is not filled with programs that represent months of hard work. When you are sure that the routines work as expected, incorporate them into your other programs.

Storing the Routines

Finally, you must decide where to put the routines. When assembled, the BSAVE is 49 bytes long and the BLOAD 39 bytes. These, like the bit map clear routine, are relocatable. That is, they can go anywhere in memory where they will not conflict with Commodore's use of memory. Some assembly language programs incorporate absolute addresses internally and must be used only at the locations where they were created. We've put the clear routine at 49152; it would be consistent to put these routines there as well.

Another good reason for placing routines in this protected area of high memory is the <RUN/STOP RESTORE> wipes out most of low memory in its attempt to restore the computer to a pristine state. This key sequence does not disturb the area from 49152 to 53247. Therefore, we will put BSAVE at 49408 and BLOAD at 49664 (if these numbers sound odd to you, they are located on even pages of memory, that is, they're both divisible by 256 with no remainder—specifically pages 193 and 194).

Placing the subroutine on even page boundaries prevents certain difficulties. One of these involves the BCS instruction. If placed where the branch would go across a page boundary, it won't work. This is just one of the many technicalities that manage to creep in when you're working so close to the heart of the computer. If you want these someplace else, be sure to watch out for this problem.

A Basic Loader

You've seen the assembler source code. But, the programs cannot be used in this form. After writing them, they have to be assembled. The assembler produces machine language—numbers that represent the translation of the source code. However, we still cannot use the program. The following program is a Basic program that POKEs the machine language code into the correct memory locations. The process is similar to hand-designing sprites and creating a Basic program to POKE in the bit maps.

```
1 REM PROGRAM BLOAD/BSAVE LOADER
5 GOSUB 800
10 PRINT "<SHIFT><CLR/HOME>"
20 INPUT "FILE NAME ->";F$
30 INPUT "START OF SAVE ->";S
40 INPUT "END OF SAVE ->";E
50 INPUT "DISK (8) OR TAPE (1) ->";D$
60 INFO = 679
```

```
  70 POKE INFO,VAL(D$):              REM DEVICE NAME
  80 POKE INFO+1,1:                  REM LOGICAL FILE NUMBER
  90 POKE INFO+2,S-256*INT(S/256):   REM GET LOW BYTE
  100 POKE INFO+3,S/256:             REM GET HIGH BYTE
  110 POKE INFO+4,E-256*INT(E/256):  REM GET LOW BYTE
  120 POKE INFO+5,E/256:             REM GET HIGH BYTE
  130 POKE INFO+6,LEN(F$):           REM FILE NAME LENGTH
  140 FOR I = 0 TO LEN(F$)-1:POKE INFO+7+I,
      ASC(MID$(F$,I+1,1)):NEXT:      REM POKE FILE NAME
  150 SYS SA:                        REM CALL SAVE
  160 IF PEEK(251)=0 THEN 200        REM CHECK FOR ERROR
  170 PRINT "ERROR #";PEEK(251):     REM PRINT MESSAGE
* 200 CLOSE 15:OPEN 15,8,15:         REM OPEN STATUS CHANNEL
* 210 INPUT#15,A$,B$,C$,D$:          REM PRINT STATUS
  220 IF VAL(A$)<>0 THEN STOP:       REM STOP IF ERROR
  300 INPUT "FILE NAME ->";F$
  310 INPUT "WHERE TO LOAD ->";S
  320 INPUT "DISK (8) OR TAPE (1) ->";D$
  330 POKE INFO,VAL(D$):             REM DEVICE NAME
  340 POKE INFO+1,1:                 REM LOGICAL FILE NUMBER
  350 POKE INFO+2,S-256*INT(S/256):  REM GET LOW BYTE
  360 POKE INFO+3,S/256:             REM GET HIGH BYTE
  370 POKE INFO+6,LEN(F$):           REM FILE NAME LENGTH
  380 FOR I = 0 TO LEN(F$)-1:POKE INFO+7+I,
      ASC(MID$(F$,I+1,1)):NEXT:      REM POKE FILE NAME
  390 SYS LO:                        REM CALL LOAD
  400 IF PEEK(251)=0 THEN 420:       REM CHECK FOR ERROR
  410 PRINT "ERROR #";PEEK(251):     REM PRINT MESSAGE
* 420 CLOSE 15:OPEN 15,8,15:         REM OPEN STATUS CHANNEL
* 430 INPUT#15,A$,B$,C$,D$:          REM PRINT STATUS
  799 END
  800 SA=49408
  810 FOR I = SA TO SA+48:READ T:POKE I,T:NEXT
  830 DATA 174,167,2,173,168,2
  832 DATA 160,0,32,186,255,173
  834 DATA 173,2,162,174,160,2
  836 DATA 32,189,255,173,169,2
  838 DATA 133,251,173,170,2,133
  840 DATA 252,174,171,2,172,172,2
  842 DATA 169,251,32,216,255,176
  844 DATA 2,169,0,133,251,96
  850 LO=49664
  860 FOR I = LO TO LO+38:READ T:POKE I,T:NEXT
  870 RETURN
  880 DATA 174,167,2,173,168,2
  882 DATA 160,0,32,186,255,173
  884 DATA 173,2,162,174,160,2
  886 DATA 32,189,255,174,169,2
  888 DATA 172,170,2,169,0,32
  890 DATA 213,255,176,2,169,0
  892 DATA 133,251,96
```

*for disk users only; tape users replace with REMS

Typing in all these DATA statements is tedious, but it is the only way to get these routines into the computer initially. Once they are in the computer's memory, however, you can save them using the BSAVE routine and load them with the normal LOAD "name",8,1 command. They'll pop right back to their original location.

Before you test these routines, save the program to your regular work disk and then put in your test disk—the one you don't mind losing. If you're working with tape, save the program on a good tape and then put a new tape into the recorder. If you do this, you don't have to reenter the entire program again if you mistyped a data statement and wiped out the disk or tape; you could simply load it again and look for the typo. To test these routines, save a screen by entering the following to the program prompts:

```
FILE NAME ->? test screen
START OF SAVE ->? 1024
END OF SAVE ->? 2023
DISK (8) OR TAPE (1) ->?
```

After entering your response to the final query, the drive will spin or you'll be asked to turn the tape player on. If you made no typing errors in the program, the program will continue. If you made an error, one of several things might happen. The disk or tape may spin for a long time, or Basic will report an error, or your computer may lock up.

If the program works but the storage device reports an error, you'll see the status report—a number for tape users, and both the error number and messages for disk users. If this happens, check the drive or player and/or the media. If there are no errors, the program continues:

```
FILE NAME ->? test screen
WHERE TO LOAD ->? 1024
DISK (8) OR TAPE (1) ->?
```

Once again, the drive will spin or you'll be asked to turn the tape player on. You should see part of the screen erase, as the screen was saved before you were prompted for the load. If the load fails, you've made a typing error in the final DATA statements—check them.

These subroutines could be used for a variety of interesting effects. But there may be problems. If you want to reconstitute a screen that uses different colors, you must also restore color RAM (that's two files). If you are restoring a high resolution graphics screen, you must save screen memory, color RAM, and the bit map—three files for one picture! And for all saves, you must keep track of the values in the various color registers, or your images may not look the same when they are reloaded.

14 Introducing SID

Move over VIC, we're going to turn our attention to the second of the special integrated circuits in your computer. The sound interface device, SID, is to music what the VIC II is to Commodore 64 graphics. It endows your computer with the best sound capabilities of any computer on the market today.

SID is controlled by POKEing a value into a group of memory locations, as is the VIC II chip. In terms of the memory map, SID is sandwiched between the VIC II ship and color RAM. Specifically, 1K of memory from 54272 to 55272 to 54296, are of concern us now.

SID is capable of singing in three voices simultaneously. Each voice has an eight-octave range and one of four unique sound types. As a bonus, there are filters and an ability to synchronize two voices for complex harmony. This is a truly powerful set of capabilities; it is lacking only in that it is limited to one register for volume—all voices must be the same volume. The sound is produced through the speaker on your television or video monitor, set high enough to be comfortably audible.

Tuning Up Voice 1

We must first clear out any residual bits in the SID control registers. The computer does not do this automatically when it is turned on. A simple loop will take care of this:

```
FOR I = 54272 TO 54272+24:POKE I,0:NEXT
```

Now that we have a fresh slate, we'll set a moderate volume:

```
POKE 54296,10
```

You can control five factors that affect the sound produced. Two of these factors are controlled by 16-bit, or two-byte, registers, and the other three factors are controlled by single-byte registers. Our first POKE will determine the waveform—the type of sound—produced. SID produces its four waveforms by POKEing a 16, 32, 64, or 128 into the waveform register. We'll start with 16, the value for a triangle waveform.

```
POKE 54276,16
```

Now POKE a note value into the correct frequency registers for voice 1:

```
POKE 54272,8:POKE 54273,97
```

When you hit <RETURN> you should be greeted with a clear ringing C. The frequency registers are 16-bit registers—the combination of 8 in the lower register and 97 in the higher register produces a C in the middle, or fourth, octave. By moving the cursor up, you can change the POKE values and see what sounds are produced. Remember you can only POKE numbers between 0 and 255.

You may notice that most waveform values don't produce sound. Only part of a byte is used for waveform, and the only legal values are 16, 32, 64, and 128. But a waveform of 64 doesn't produce any sound, because the pulse waveform (64) requires two additional POKEs:

```
POKE 54276,64
POKE 54274,20:POKE 54275,17
```

Though you will discover an astonishing array of notes, remember that you're playing with only one voice now—there are two others to contend with. Fortunately, Commodore engineers arranged the array the SID registers in a meaningful way. Fig. 14.1 shows the registers and their uses.

Frequency Registers

The actual note that SID plays depends on the value in the frequency registers. The two registers work together to produce the entire range of sounds from very low to very high. While every number POKEd produces a sound, not all values relate to a normal musical scale. To sound a note on this scale, the POKE values are generated by the following formula:

```
N=freq/.06097
```

Figure 14.1
SID Registers

Locations for: Voice 1	Voice 2	Voice 3	Meaning
54272	54279	54286	Frequency 1
54273	54280	54287	Frequency 2
54274	54281	54288	Pulse Width 1
54275	54282	54289	Pulse Width 2(bits 0–3)
54276	54283	54290	Waveform
54277	54284	54291	Attack/Decay
54278	54285	54292	Sustain/Release
54293	54293	54293	Filter Cutoff 1
54294	54294	54294	Filter Cutoff 2
54295(0)	54295(1)	54295(2)	Filter/Resonance*
54296	54296	54296	Volume**

*Bits in parentheses control voice: 1 = filter, 0 = no filter
**Lower 3 bits only. Upper bits control type of filter.

where N represents the note value and freq represents the desired frequency. For example, to calculate the POKE values to generate a frequency of 1,046 cycles per second (Hertz or Hz), enter the following (with no line numbers):

```
A=1046
B=A/.06097
PRINT B
```

You should see 17155.97. This is the value needed by SID to generate a note with a frequency of 1,046 Hz. But we cannot POKE a value larger than 255, so this value must be split into high and low bytes. Recall that we performed a similar manipulation of addresses in Chapter 13. Try the following:

```
PRINT B/256
PRINT B-(256*INT(B/256))
```

You should see 67.0155078 and 3.97000122. These are the actual POKE values for the frequency registers. Of course, your Commodore 64 would round these off before it completed the POKE. The first number is the high frequency POKE value and the second number is the low frequency POKE value.

Here's a small program to show off some of these notes:

```
10 REM NOTE SHOW-OFF PROGRAM
100 SID=54272:F1=SID:F2=SID+1:W1=SID+4:A1=SID+5:S1=SID+6
120 GOSUB 900:REM CLEAR SID REGISTERS
130 POKE SID+24,15:REM VOLUME
140 W=32:REM WAVEFORM
150 POKE A1,10:POKE S1,10:REM ATTACK & SUSTAIN
160 FOR I = 1 TO 255 STEP 50
170 FOR J = 1 TO 255 STEP 10
180 PRINT "<SHIFT><CLR/HOME>F1=";I,"F2=";J
190 POKE F1,I:POKE F2,J:REM POKE NOTE VALUES
195 POKE W1,W+1:REM WAVEFORM & START ENVELOPE
200 FOR K = 1 TO 100:NEXT K
205 POKE W1,W+0:REM WAVEFORM & FINISH ENVELOPE
210 NEXT J:NEXT I
220 GOSUB 900
230 END
900 FOR I = SID TO SID+24:POKE I,0:NEXT
910 RETURN
```

You might wonder about the purpose of line 200, but try the program without it. When SID gets POKEd, it keeps on playing. This line gives all notes an arbitrary duration of 100 counts. Line 205 turns off the first voice by POKEing zeroes into the frequency registers.

Chromatic Scale

With 65,535 possible notes, our problem is to find which of these are actually notes in a normal chromatic scale. The algorithm shown above does work, but would be slower than molasses in January in an actual program. Added to this is the problem of Basic's accuracy when dealing with very small (or large) numbers. The best way to manage these POKE values is to keep them in a table—an array. Accessing them can be a matter of simply indexing the array. With 12 notes per octave and eight octaves, there are 96 notes. With two associated POKE values per note, the table will have 176 values. The values are given in Fig. 14.2.

Enter the following program fragment and save it. We can then load it and add the rest of our music programs to it. In this way, you'll never have to type the table again. It has high line numbers, so that it will not interfere with the lower line numbers used in the other programs.

Figure 14.2
POKE Values for Musical Notes

	Octave 0		1		2		3	
Note	High	Low	High	Low	High	Low	High	Low
C	1	12	2	24	4	48	8	97
C#	1	28	2	56	4	112	8	225
D	1	45	2	90	4	180	9	104
D#	1	62	2	125	4	251	9	247
E	1	81	2	163	5	71	10	143
F	1	102	2	204	5	152	11	48
F#	1	123	2	246	5	237	11	218
G	1	145	3	35	6	71	12	143
G#	1	169	3	83	6	167	13	78
A	1	195	3	134	7	12	14	24
A#	1	221	3	187	7	119	14	239
B	1	250	3	244	7	233	15	210

	Octave 4		5		6		7	
Note	High	Low	High	Low	High	Low	High	Low
C	16	195	33	135	67	15	134	30
C#	17	195	35	134	71	12	142	24
D	18	209	37	162	75	69	150	139
D#	19	239	39	223	79	191	159	126
E	21	31	42	62	84	125	168	250
F	22	96	44	193	89	131	179	6
F#	23	181	47	107	94	214	189	172
G	25	30	50	60	100	121	200	243
G#	26	156	53	57	106	115	212	230
A	28	49	56	99	112	199	225	143
A#	29	223	59	190	119	124	238	248
B	31	165	63	75	126	151	253	46

Source: *Commodore 64 Programmer's Reference Guide*, Commodore Business Machines, Wayne, PA, 1982. pp 384-386.

```
10000 DATA 1,12,1,28,1,45,1,62,1,81,1,102
10005 DATA 1,123,1,145,1,169,1,195,1,221,1,250

10010 DATA 2,24,2,56,2,90,2,125,2,163,2,204
10015 DATA 2,246,3,35,3,83,3,134,3,187,3,244

10020 DATA 4,48,4,112,4,180,4,251,5,71,5,152
10025 DATA 5,237,6,71,6,167,7,12,7,119,7,233

10030 DATA 8,97,8,225,9,104,9,247,10,143,11,48
10035 DATA 11,218,12,143,13,78,14,24,14,239,15,210
```

```
10040 DATA 16,195,17,195,18,209,19,239,21,31,22,96
10045 DATA 23,181,25,30,26,156,28,49,29,223,31,165

10050 DATA 33,135,35,134,37,162,39,223,42,62,44,193
10055 DATA 47,107,50,60,53,57,56,99,59,190,63,75

10060 DATA 67,15,71,12,75,69,79,191,84,125,89,131

10065 DATA 94,214,100,121,106,115,112,199,119,124,126,151

10070 DATA 134,30,142,24,150,139,159,126,168,250,179,6
10080 DATA 189,172,200,243,212,230,225,143,238,248,253,46
```

As formidable as this table may seem, it's really quite simple to use. All the numbers will be read into a pair of two-dimensional arrays: NH( ) and NL( ). The first index of the array will be by octave, while the second will determine the actual note. A simple loop will allow us to play chromatic scales:

```
10 DIM NH(7,12),NL(7,12)
20 FOR I = 0 TO 7
30 FOR J = 1 TO 12
40 READ NH(I,J),NL(I,J)
50 NEXT J:NEXT I
```

Run the program as shown above. It won't play music, but if you have omitted data items, the computer will tell you with an out-of-data error. You can facilitate debugging this type of error by adding the following line to the program:

```
45 PRINT NH(I,J),NL(I,J):IF J=12 THEN INPUT A$
```

This line will print the note values as they are read into the array. It will pause for an input after each octave. You can then compare the numbers on your screen with the values listed here. When you find a discrepancy, press <RUN/STOP> to halt the program and correct the offending data statement.

Even after the program reads the arrays correctly, it may still report errors in the values of the data. To check these for accuracy, you can debug by ear with the following directions. (Unless you are tone deaf, in which case you must get some help with this phase of the debugging.) Enter the following:

```
10 REM EAR-DEBUGGING ROUTINE
100 SID=54272:F1=SID:F2=SID+1:W1=SID+4:A1=SID+5:S1=SID+6
120 GOSUB 900:REM CLEAR SID REGISTERS
130 POKE SID+24,15:REM VOLUME
```

```
140 W=32:REM WAVEFORM
150 POKE A1,10:POKE S1,0:REM ATTACK & SUSTAIN
160 FOR I = 0 TO 7
170 FOR J = 1 TO 12
180 PRINT "<SHIFT><CLR/HOME>OCTAVE=";I,"F1=";NH(I,J),
    "F2=";NL(I,J)
190 POKE F1,NL(I,J):POKE F2,NH(I,J):REM POKE NOTE VALUES
195 POKE W1,W+1:REM WAVEFORM & START ENVELOPE
200 FOR K = 1 TO 100:NEXT K
205 POKE W1,W+0:REM WAVEFORM & FINISH ENVELOPE
210 NEXT J:NEXT I
220 GOSUB 900
230 END
900 FOR I = SID TO SID+24:POKE I,0:NEXT
910 RETURN
```

Most of this program is almost identical to our first SID program except for the central I/J loops, which now go through the octaves (I) and the notes (J), one by one. For ear debugging, you might want to increase the delay in line 200; a duration of only 100 counts may not be long enough, especially when your ear has to recover from the previous tone.

Save this program, as we will be using it as the basis for other programs.

The lower octaves are quite fuzzy on my speaker (in the Commodore 1701 Color Monitor) and may require an external speaker for full appreciation.

Encoding Music

To use this information to encode music we must consider what information we need for each note. First we must know what the NL( ) and NH( ) values are; they represent the octave and note number (with C being note 1 and B being note 12). We must also set the note duration. There are many possible ways to encode these three pieces of information, and it's important that we choose the best way. Too simple a structure will prevent us from adding a second voice or changing waveform or other SID value.

Although we must POKE a numeric value into the SID registers, we can convert strings into numerics with the VAL function. Strings are stored efficiently in memory and can be easily saved to disk in simple sequential files. Each character of a string takes up only one byte of memory, since it is stored as its ASCII code.

Basic maintains only a small amount of housekeeping information on strings. Numbers, on the other hand, require more storage; each integer

requires two bytes, and each decimal number as much as seven bytes. With string arrays, we have an easy way to connect voices or parts of a song, so we'll use strings to represent the notes. (For an alternative method, see the *Commodore 64 Programmer's Reference Guide*.)

Each string will have this structure:

"ONNDDD,"

where O represents the octave, NN the note number, and DDD the note duration. This structure should accommodate most simple musical efforts. We can access individual values by using the string functions:

```
O=VAL(MID$(N$(n),1,1))
N=VAL(MID$(N$(n),2,2))
D=VAL(MID$(N$(n),4,3))
```

where n, in N$(n), represents the individual notes of a song.

Let's put our theory into practice.

This program will be based on the previous program, which reads in all the music data. We will simply add the song data and add the necessary control routines to decode the note strings.

```
5 REM PROGRAM FLYING TRAPEZE

60 NM=16:REM NUMBER OF MEASURES
70 DIM N$(NM)
80 FOR I = 1 TO NM:READ N$(I):NEXT I:REM READ SONG

160 FOR I = 1 TO NM
170 C=1:TD=0
180 GOSUB 600:REM GET NOTES
190 POKE F1,NL(O,N):POKE F2,NH(O,N)
195 POKE W1,W+1:REM WAVEFORM & START ENVELOPE
200 FOR K = 1 TO D:NEXT K:REM NOTE LENGTH
205 POKE W1,W+0
210 TD=TD+D:IF TD<>300 THEN C=C+6:GOTO 180
220 NEXT I
230 GOSUB 900
240 END

600 O=VAL(MID$(N$(I),C,1))
610 N=VAL(MID$(N$(I),C+1,2))
620 D=VAL(MID$(N$(I),C+3,3))
630 RETURN

11000 DATA 000200403100
11010 DATA 403100408100410100
11020 DATA 412100412100412100
11030 DATA 501100405100405100
11040 DATA 410200403100
```

```
11050 DATA 403100407100412100
11060 DATA 401100412100410100
11070 DATA 401100408100405100
11080 DATA 403200403100
11090 DATA 403100408100410100
11100 DATA 403100403100403100
11110 DATA 501100405100405100
11120 DATA 410200403050403050
11130 DATA 410100407100408100
11140 DATA 410100412100410100
11150 DATA 408200408050408050
```

This program is straightforward, once you understand how the notes are chopped apart in the subroutine at line 600. The variable C represents the place where the string gets chopped up; the start of a note. Not all measures in the song are the same length, so we cannot depend on the length of the measure strings to be identical. Because this song is in three-quarter time, I chose an arbitrary measure duration of 300; a whole note would last for 300 counts in the FOR/NEXT loop at line 200. We can tell when a measure is completed by seeing if the total durations for each measure add up to 300 in line 310. If they have, the IF test fails and the NEXT I continues, assuring that both C and TD are returned to 0.

Envelopes: ADSR

Your Commodore 64 is capable of producing far more complex sounds than just the tone we've been playing with. While a technical discussion of waveforms and filters is beyond the scope of this book, we can look at some general principles and let trial-and-error techniques help us find our sound. The bibliography contains references for readers interested in a technical discussion of these areas.

The envelope controls the internal volume of each note. While our volume register determines how loud a note becomes, we have much control within each note. Percussion instruments, for example, reach peak volume almost as soon as they are struck. The envelope component that controls this facet of each note is called the attack. After reaching its peak, volume falls off. This component is decay.

Attack and decay are coded together (four bits or nibble for each). But with some musical instruments, the sound doesn't decay completely, but sustains itself at some diminished level; sustain is the third envelope component. Finally, the sustain level falls to zero—this phase is termed release. For SID, sustain and release are coded in one byte like attack and decay. Attack,

decay, sustain, and release, in combination with the waveform, determine what the note will sound like. By varying the envelope, you can make your Commodore sould like a piano or a violin.

Since both attack/decay and sustain/release are determined by nibbles rather than bytes, we can call on the algorithm we used in Chapter 12 to set a nibble, namely:

```
POKE register, ATTACK*16+decay
POKE register, SUSTAIN*16+release
```

Waveforms

The waveform is a different animal altogether; it is the shape of the wave. The notes you hear when you play an instrument on the Commodore 64 can be thought of as waves, like ripples radiating when a pebble is thrown into a pond. The pond waves have a sinusoidal shape—similar to the sine waves we drew in Chapter 11.

Real waves from musical instruments do not have so simple a shape. Their actual shape is determined by the harmonics of the instrument. Violins and other stringed instruments have resonating chambers to allow the waves to develop more subtle shapes. Harmonics are very predictable, and synthesizer circuits can easily produce waveforms with specific shapes. SID is one of them.

SID is able to produce four different basic waveforms: a sawtooth wave, a triangle wave, a pulsed wave, and a randomly shaped white noise wave. Adjusting the envelope to produce the correct sound is only half of what you need to make SID imitate a musical instrument. You must also select the right waveform. Waveforms are represented by bits in memory. You can ask SID to combine two or three waveforms by turning on the right bits (see Fig. 14.3 for more details). There is a different waveform byte for each voice.

Figure 14.3
Waveform Bits

Bit value	Function
128	Select White Noise
64	Select Pulse*
32	Select Sawtooth
16	Select Triangle
8	
4	
2	
1	Start ADSR Cycle

*Must also set waveform width bytes

Bit 0, the least significant bit, is used to start the envelope generator. If it is set, the attack is started; if it is a 0, the release is started. You should have noted already how this bit has been tweaked—cleverly manipulated—in the programs earlier in this chapter. The other bits in the waveform byte are used with synchronization and ring modulation.

Setting the waveform is a fairly straightforward process— just POKE the correct value (128, 64, 32, or 16) and add 1 to start the envelope generator.

The pulse waveform, however, requires two additional POKEs. This wave consists of pulses, and Commodore engineers gave you control of how wide to make them. Each voice has two registers (a whole byte and a nibble) that control the width of the pulse. Pulse waveforms produce piano-like sounds. Changing the width of the pulse can alter the basic sound to make it more like that of an organ. Of course, you must also endow the envelope with the fast attack and release of these instruments.

At least, this is the theory. To me, these synthesized sounds sound like synthesized sounds, regardless of the envelope or waveform. But this is not necessarily bad. Just as in graphics, computers bring vivid new tools to art and music. These tools should not be judged by how well they can be used to mimic other media. Instead, They should be judged on how well they accomplish their purposes. SID is an excellent computer music tool, the best standard equipment in the industry.

Sound Editor

Changing these parameters is extremely difficult; it would be convenient to be able to specify the numbers you wanted rather than to have to continually change and rerun the program. The program that follows is a sound editor that allows you to pick and choose the envelope and waveform parameters for your musical compositions.

The program is very similar to our last program. However, instead of looping through the whole range of SID, it allows you to change the SID parameters and then play only one octave with envelope and waveform. All we really need to do is replace the sound definition lines (lines 130-150) with an input request and make some minor changes to the play loop. Load the previous program and make the changes in the lines indicated with asterisks below:

```
* 5 PRINT "<SHIFT CLR/HOME>"
  10 DIM NH(7,12),NL(7,12)
  20 FOR I = 0 TO 7
```

```
  30  FOR J = 1 TO 12
  40  READ NH(I,J),NL(I,J)
  50  NEXT J:NEXT I
  100 SID=54272:F1=SID:F2=SID+1:W1=SID+4:A1=SID+5:S1=SID+6
* 110 VOL=SID+24:P1=SID+2:P2=SID+3:REM NEW CONSTANTS
  120 GOSUB 900:REM CLEAR SID
* 130 GOSUB 500:REM INPUT ROUTINE
* 140
* 150
* 160
  170 FOR J = 1 TO 12
* 180 PRINT "<SHIFT><CLR/HOME>OCTAVE=";O,"F1=";NH(O,J),
      "F2=";NL(O,J)
* 190 POKE F1,NL(O,J):POKE F2,NH(O,J):REM POKE NOTE VALUES
  195 POKE W1,W+1:REM WAVEFORM & START ENVELOPE
  200 FOR K = 1 TO 200:NEXT K
  205 POKE W1,W+0:REM WAVEFORM & FINISH ENVELOPE
* 210 NEXT J
  220 GOSUB 900
* 230 GOTO 130
  900 FOR I = SID TO SID+24:POKE I,0:NEXT
  910 RETURN
```

and add the following new lines:

```
500 PRINT "<CLR/HOME><8 CRSR DOWN>";
510 PRINT "<5 CRSR RIGHT>ATTACK -     ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=A
512 A=T:PRINT A
515 PRINT "<5 CRSR RIGHT>DECAY -     ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=D
517 D=T:PRINT D
520 PRINT "<5 CRSR RIGHT>SUSTAIN -     ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=S
522 S=T:PRINT S
525 PRINT "<5 CRSR RIGHT>RELEASE -    ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=R
527 R=T:PRINT R
530 PRINT "<5 CRSR RIGHT>WAVE FORM -     ";:GOSUB 920:IF F=0 OR
    T>255 THEN T=W
532 W=T:PRINT W
540 PRINT "<8 CRSR RIGHT>PULSE -     ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=X1
542 X1=T:PRINT X1
550 PRINT "<8 CRSR RIGHT>PULSE -     ";:GOSUB 920:IF F=0 OR
    T>255 THEN T=X2
552 X2=T:PRINT X2
560 PRINT"<5 CRSR RIGHT>VOLUME -     ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=V
562 V=T:PRINT V
```

```
570 PRINT"<5 CRSR RIGHT>OCTAVE -      ";:GOSUB 920:IF F=0 OR
    T>7 THEN T=0
572 O=T:PRINT O
580 POKE A1,A*16+D
582 POKE S1,S*16+R
584 POKE VOL,V
586 POKE P1,X1
588 POKE P2,X2
590 RETURN

920 PRINT "<3 CRSR LEFT>";:A$="":F=0
930 PRINT "<C=64 P><CRSR LEFT>";:GET T$:IF T$="" THEN 930
940 IF T$=CHR$(13) THEN 980
950 IF T$="Q" THEN END
955 IFT$=CHR$(20)ANDLEN(A$)>0THENPRINT"<CRSR LEFT><2 SPACE>
    <2 CRSR LEFT>";:A$=LEFT$(A$,LEN(A$)-1):GOTO930
960 IF T$<"0" OR T$>"9" THEN 930
970 PRINT T$;:A$=A$+T$:F=-1:GOTO 930
980 T=VAL(A$):PRINT " ";
990 FOR I = 1 TO LEN(A$)+1:PRINT "<2 CRSR LEFT><1 SPACE>";:NEXT
995 RETURN
```

How It Works

The theory behind the first subroutine should be relatively clear once we know what T and F are. T is a temporary variable that handles the music parameter entered by the user. Most of these parameters cannot accept a number larger than 15, so it's smart to protect against errors here. Thus, if T is greater than the largest valid number, it takes on the old value of the parameter. This is the T=A in the IF statement on line 510.

On the other hand, it's maddening to have to type in nine numbers when you are changing only one of them. So if <RETURN> is pressed in response to the prompt, F (for flag) has a value of zero and T once again takes on the old value of the parameter; this covers the IF F=0 OR T>15 THEN T=A part of the input subroutine. In the next line, 512, the value of T is actually assigned to A. If T is less than 15 and if more than <RETURN> was pressed, T now has a new value and the attack will change. Otherwise, T keeps the old attack value intact.

The actual input subroutine may be harder to understand. Because the program must prompt for nine different values and I wanted to cushion against errors, one general-purpose routine seemed ideal, hence T and F. I also wanted to avoid the question mark put on the screen by an INPUT statement.

In some microcomputer Basics, all you have to do is include a print string in the input line: INPUT "ATTACK – >A";A and the question mark

is supressed. But not on the Commodore 64. INPUT always displays a question mark. So that leaves us with GET, which unfortunately doesn't put a cursor on the screen. Thus, line 930 uses a character graphic <C=64 P> to put an underscore on the screen as a cursor.

Having done this, we now have the responsibility of maintaining the cursor—erasing it when the user types RETURN, moving it back when the user erases, advancing it when the user enters a number, and so on. All of the cursor movement keys imbedded in the print strings in this subroutine are there to maintain the cursor. Remove some of them and experiment with the simulated cursor; advanced programmers have to do this all the time.

This subroutine features restricted input. Only a limited set of key presses accomplish anything. Pressing Q stops the program. Pressing <RETURN> assigns T (the numeric variable) the VALue of A$, erases the "cursor," and sends the program back to the prompting subroutine (lines 500 . . .). Any number is passed by line 960 and it is concatenated onto A$, building a string that can be converted to a number by VAL.

The more difficult line is 955. Note that in the listing above this line contains no spaces. It is very close to the maximum line length (80 characters) on the Commodore 64, so I've omitted spaces that I normally include for readability. CHR$(20) is the code generated by the <INST/DEL> key. If T$ is CHR$(20) AND the input string (A$) is bigger than 0 then we move our cursor back, erase the old number from the screen, and remove one character from the right side of the string by using the LEFT$() function.

This subroutine is another for your programmer's tool kit. Although here it is configured for this specific program, it can easily be modified to accept other types of restricted inputs. Depending on who will be operating your programs, subroutines such as these can make the difference between frustration and success.

Sprucing Up the Display

You might have wondered why the first prompt appears so low on the screen. I was saving the top lines of the screen for some fancy graphics. The following short subroutine places part of a piano keyboard onto the screen:

```
300 K1$="<CTRL 2><CRTL 9><2 SPACE><C=64 N><CRSR DOWN>
         <3 CRSR LEFT><2 SPACE><C=64 N><CRSR DOWN>
         <3 CRSR LEFT><2 SPACE><C=64 N><CRSR DOWN>
         <3 CRSR LEFT><2 SPACE><C=64 N><CRSR DOWN>
```

```
        <3 CRSR LEFT><2 SPACE><C=64 N><CRSR DOWN>
        <3 CRSR LEFT><2 SPACE><C=64 N>"
305 K2$="<CTRL 1><CTRL 9><2 SPACE><CRSR DOWN>
        <2 CRSR LEFT><2 SPACE><CRSR DOWN>
        <2 CRSR LEFT><2 SPACE><CRSR DOWN>
        <2 CRSR LEFT><2 SPACE><CTRL 2><CTRL 0>"
310 K3$="<CTRL 2><CRTL 9><1 SPACE><C=64 N><CRSR DOWN>
        <2 CRSR LEFT><1 SPACE><C=64 N><CRSR DOWN>
        <2 CRSR LEFT><1 SPACE><C=64 N><CRSR DOWN>
        <2 CRSR LEFT><1 SPACE><C=64 N><CRSR DOWN>
        <2 CRSR LEFT><1 SPACE><C=64 N><CRSR DOWN>
        <2 CRSR LEFT><1 SPACE><C=64 N>"
320 PRINT "<CLR/HOME>";
325 PRINT K3$;"<6 CRSR UP>";
330 FOR I = 1 TO 12:PRINT K1$;"<6 CRSR UP>";:NEXT
335 PRINT "<6 CRSR UP>";K3$
340 PRINT "<CLR/HOME>";
345 FOR I = 1 TO 13
350 IF I = 4 OR I = 7 OR I = 11 THEN PRINT "<3 CRSR RIGHT>";:
    GOTO 360
355 PRINT "<4 CRSR UP><CRSR RIGHT>";K2$;
360 NEXT
370 RETURN

115 GOSUB 300
```

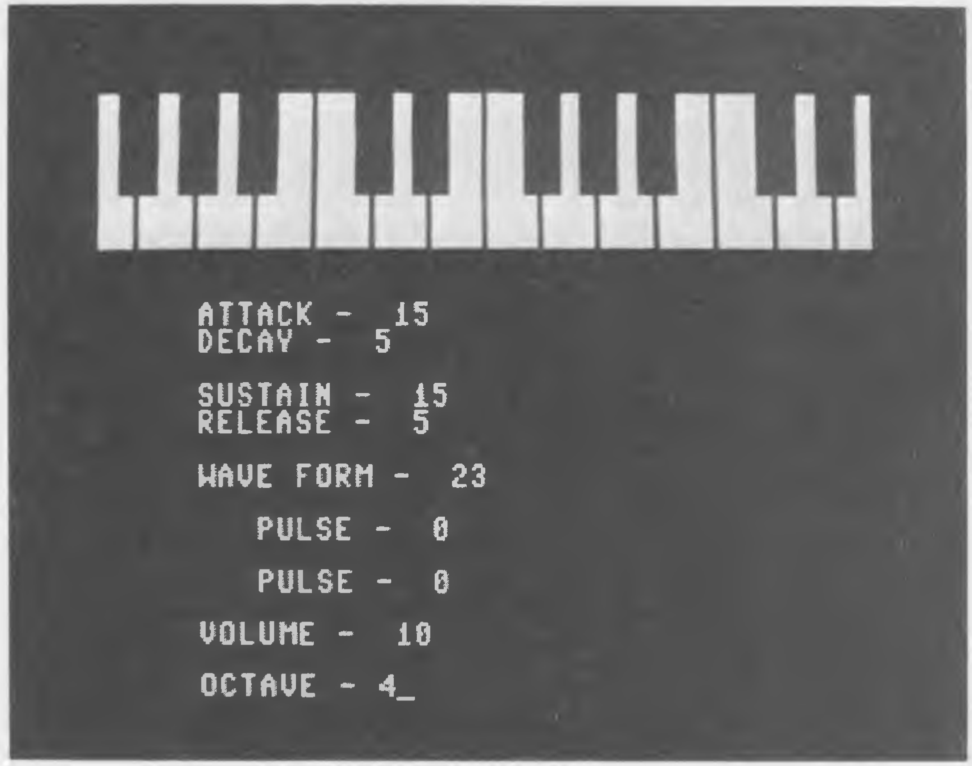
Lines 300 to 310 may be difficult to type in, but will fit easily on a Basic line when the conventions we're using for graphics keys are translated into Commodore symbols. K1$ holds the white piano keys and K3$ holds the narrower keys that appear on the far left and far right of the screen. The <C=64 N> keys insert spaces between the keys. K2$ holds the black keys. Pianos have black keys in alternating clusters of two and three; the IF statement in line 350 assures proper gaps. Fig. 14.4 shows how this looks on the screen when this program is running.

Now that we have a keyboard, let's use it. The following subroutine will read the POKE locations for a centered character on each of the keys of the keyboard:

```
10 DIM NH(7,12),NL(7,12),NP(24)
400 FOR I = 1 TO 24:READ NP(I):NEXT

11000 DATA 1224,1105,1227,1108,1230,1112
11010 DATA 1233,1236,1117,1239,1120,1242
11020 DATA 1245,1126,1248,1129,1251,1132
11030 DATA 1254,1257,1138,1260,1141,1262
```

Figure 14.4
Screen Display Produced by the Final Sound Editor



We must GOSUB to this routine and use it in the main play loop:

```
115 GOSUB 300:GOSUB 400
140 N=8:REM STARTING NOTE
195 POKE NP(N),81+128:POKE NP(N-1),32+128:N=N+1
225 POKE NP(N-1),32+128
```

The new variable N provides a convenient way to index into the note position array (NP()). Because this program plays a scale, the notes are sequential, and N can easily be incremented to the next note position. If you want to modify this program to play a song, N must be related to the frequency of the note being played.

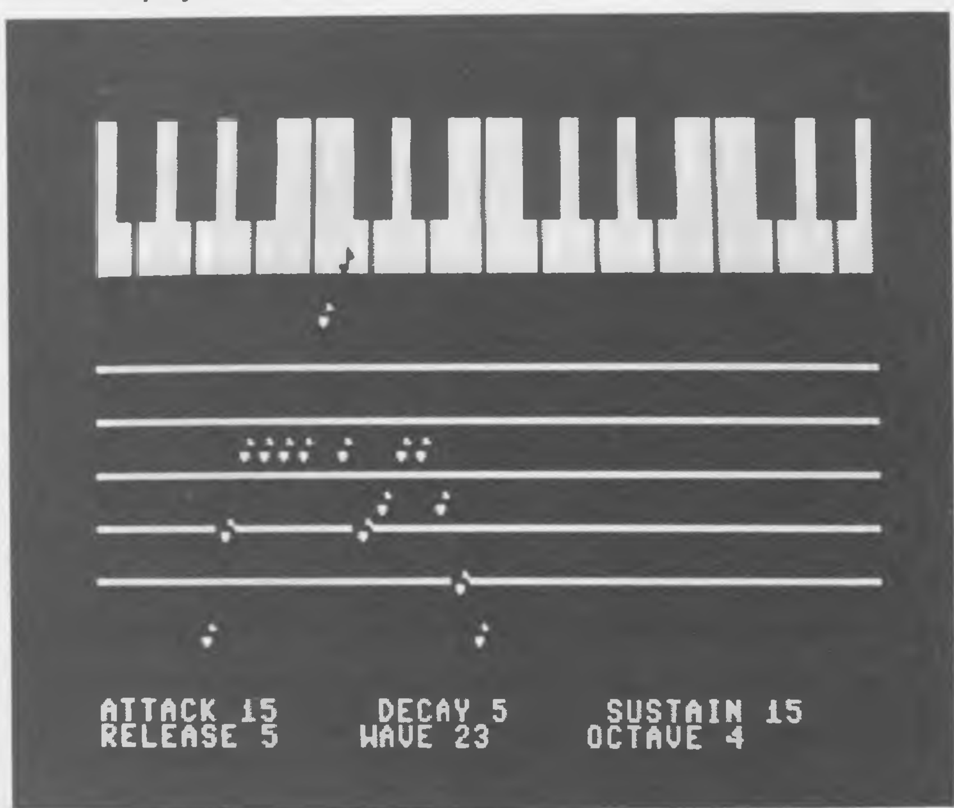
As a final graphics modification, the following additions will create a character in the shape of an eighth note:

```
410 POKE 56334,PEEK(56334)AND254:POKE1,PEEK(1)AND251
420 CHAR=14336
430 FOR I = 0 TO 2047:POKE CHAR+I,PEEK(53248+I):NEXT
440 POKE 1,PEEK(1)OR4:POKE56334,PEEK(56334)OR1
450 FOR I = 0 TO 7:READ T:POKE CHAR+(209*8)+I,T:NEXT
460 POKE 53272,31
470 RETURN

12000 DATA 251,249,248,250,251,195,195,231
```

With this program as a foundation, you will be able to create some very interesting music programs. Fig. 14.5 shows one of my modifications. But there's more to SID than one voice.

Figure 14.5
Screen Display with a Staff



Multiple Voice Music

SID has three voices and it is possible to program harmony or other multiple-voice effects. The main problem with having access to more than one voice is housekeeping. Since it is highly unlikely that both voices in a two-part piece will have the same number of notes per measure, multiple-voice programs need to keep track of each note in each voice and make the correct POKEs when needed. the voices are related, so using arrays to hold all the registers simplifies the logic a bit. This program is based on FLYING TRAPEZE. Load it and enter the following changes:

```
5 PRINT "<SHIFT CLR/HOME>"
10 DIM NH(7,12),NL(7,12),NP(24)
20 FOR I = 0 TO 7
30 FOR J = 1 TO 12
40 READ NH(I,J),NL(I,J)
50 NEXT J:NEXT I
60 <RETURN>
70 <RETURN>
80 <RETURN>
100 SID=54272:F1(1)=SID:F2(1)=SID+1:W(1)=SID+4:
    A(1)=SID+5:S(1)=SID+6
102 F1(2)=SID+7:F2(2)=SID+8:W(2)=SID+11:
    A(2)=SID+12:S(2)=SID+13
104 F1(3)=SID+14:F2(3)=SID+15:W(3)=SID+18:
    A(3)=SID+19:S(3)=SID+20
110 VOL=SID+24:P1(1)=SID+2:P2(1)=SID+3
112 P1(2)=SID+9:P2(2)=SID+10
114 P1(3)=SID+16:P2(3)=SID+17
```

These new variables, all in the form of arrays, represent the various SID registers for each voice (check back to Fig. 14.1).

Now let's turn our attention to our data. Recall the string format we used previously: "ONNDDD." This will work fine here with minor modifications. Now that we have more than one voice, there is a possibility that one measure may contain a whole note for one voice and four quarter notes for the other voice. To keep track of all this, we will index the data by measure and voice. Fig. 14.6 will help you visualize this method of data storage.

I have set an arbitrary count of 30 per measure—a whole note would have a duration of 30. This reduction from our previous count of 300 is

Figure 14.6
Data Structure for Multiple Voices

Index Down **Measure**	Index Across **Voice** 1	2	3
1	103500	510250610250	201075202075203350
2	104500	610500	202075203075204075205275

necessary because of the additional processing required for the extra voice. Voice 2, measure 2, does indeed have this duration. Note 5, octave 4, will be the note played here. While this note is playing, voice 1 will be playing three notes, each for a duration of 10 counts (or the equivalent of a quarter note). Initially, voice 1 will play note 3 from octave 4, then note 8 from octave 4.

When our main loop has counted to 30, the measure is done, and the measure index is incremented. Get the idea?

```
180 FOR J = 1 TO NM:REM NM = NUMBER OF MEASURES
200 FOR M = 1 TO 30:REM M = MASTER MEASURE COUNT
210 FOR K = 1 TO NV:REM NV = NUMBER OF VOICES
220 IF TD(K)=0 THEN GOSUB 600
230 TD(K)=TD(K)-1
240 NEXT K
250 NEXT M
260 NEXT J
270 GOSUB 900:REM TURN OFF SID
280 END
```

It's easy to chop up the strings; the problem is knowing when to GOSUB to the string chopper/new note routine. Easy! We'll keep track as we did in the one-voice program, by decrementing a counter. When it is zero, we are finished with the note. This time, because we can have up to three voices, the decrementing must happen in a loop, or at least happen once for each voice.

As we'll see, knowing the voice that reached one (K) will make the string chopper/new note subroutine much simpler. Therefore, with the index K representing the current voice, TD(K) will be the temporary duration of the current note.

Now we will modify the string chopper/new note subroutine to get the correct piece of the correct string. We already know what voice to get (it's held in K) and which measure we are in (it's held in J). So. . .

```
600 POKE W(K),WV(K)+0:REM END ENVELOPE
610 O=VAL(MID$(N$(J,K),C(K),1))
620 N=VAL(MID$(N$(J,K),C(K)+1,2))
630 TD(K)=VAL(MID$(N$(J,K),C(K)+3,3))
640 POKE F1(K),NL(O,N)
650 POKE F2(K),NH(O,N)
660 POKE W(K),WV(K)+1
670 C(K)=C(K)+6
680 TD(K)=TD(K)/10:REM REDUCE TO ACCOMODATE NEW COUNT
690 RETURN
```

Notice that there is a new variable in this listing. C() represents the chop point in the string. However, since we have more than one voice, and in any

one measure each voice could have different numbers of notes, the chop point is no longer just the old chop point plus six.

There must be a different chop point for each voice. If there is a whole note for a voice in a particular measure, the chop point never goes beyond six; but if there are several short notes, the chop point must advance each time a new note is retrieved. Keeping the chop point in an array solves the whole problem. Except for one small factor—it must be initialized to one for each measure:

```
185 FOR I = 1 TO NV:C(I)=1:NEXT
```

The new array WV() holds the waveform for each voice. With separate waveforms for each voice, you can make a white noise voice imitate percussion while the other two voices harmonize.

All that remains is an initializing section to establish the array of notes and the initial values for WV():

```
130 NV=2:NM=16:DIM N$(NM,NV)
140 WV(1)=16:WV(2)=16:WV(3)=32
150 FOR I = 1 TO NM:FOR J = 1 TO NV:READ N$(I,J):NEXT:NEXT
160 POKE VOL,10:POKE A(1),10:POKE S(1),10:POKE A(2),5:
    POKE S(2),48
```

and of course the data:

```
11005 DATA 000300
11015 DATA 403300
11025 DATA 312300
11035 DATA 401300
11045 DATA 310300
11055 DATA 307300
11065 DATA 401300
11075 DATA 405300
11085 DATA 312300
11095 DATA 403300
11105 DATA 312300
11115 DATA 401300
11125 DATA 310300
11135 DATA 410300
11145 DATA 403300
11155 DATA 312200312050312050
```

This is the data for the second voice. For the first voice we'll use the data from the FLYING TRAPEZE program.

With the programs in this chapter, you should have a good handle on harnessing the power of SID. However, you may also want to explore the capabilities of languages other than Basic to program music on your Commodore 64. Most of these alternatives support music to a much greater degree that Basic 2.0. The next chapter will introduce you to three of these.

```
4 REM SOUND EDITOR WITH PIANO DISPLAY
5 PRINT "<SHIFT CLR/HOME>"
10 DIM NH(7,12),NL(7,12),NP(24)
20 FOR I = 0 TO 7
30 FOR J = 1 TO 12
40 READ NH(I,J),NL(I,J)
50 NEXT J:NEXT I
100 SID=54272:F1=SID:F2=SID+1:W1=SID+4:A1=SID+5:S1=SID+6
110 VOL=SID+24:P1=SID+2:P2=SID+3:REM NEW CONSTANTS
115 GOSUB 300:GOSUB 400
120 GOSUB 900:REM CLEAR SID
130 GOSUB 500:REM INPUT ROUTINE
140 N=8:REM STARTING NOTE
170 FOR J = 1 TO 12
190 POKE F1,NL(O,J):POKE F2,NH(O,J):REM POKE NOTE VALUES
195 POKE NP(N),81+128:POKE NP(N-1),32+128:N=N+1:POKE W1,W+1
200 FOR K = 1 TO 200:NEXT K
205 POKE W1,W+0:REM WAVEFORM & FINISH ENVELOPE
210 NEXT J
220 GOSUB 900
225 POKE NP(N-1),32+128
230 GOTO 130
300 K1$="<CTRL 2><CRTL 9><2 SPACE><C=64 N><CRSR DOWN>
         <3 CRSR LEFT><2 SPACE><C=64 N><CRSR DOWN>
         <3 CRSR LEFT><2 SPACE><C=64 N><CRSR DOWN>
         <3 CRSR LEFT><2 SPACE><C=64 N><CRSR DOWN>
         <3 CRSR LEFT><2 SPACE><C=64 N><CRSR DOWN>
         <3 CRSR LEFT><2 SPACE><C=64 N>"
305 K2$="<CTRL 1><CTRL 9><2 SPACE><CRSR DOWN><CRSR DOWN>
         <2 CRSR LEFT><2 SPACE><CRSR DOWN>
         <2 CRSR LEFT><2 SPACE><CRSR DOWN>
         <2 CRSR LEFT><2 SPACE><CTRL 2><CTRL 0>"
310 K3$="<CTRL 2><CRTL 9><1 SPACE><C=64 N><CRSR DOWN>
         <2 CRSR LEFT><1 SPACE><C=64 N><CRSR DOWN>
         <2 CRSR LEFT><1 SPACE><C=64 N><CRSR DOWN>
         <2 CRSR LEFT><1 SPACE><C=64 N><CRSR DOWN>
         <2 CRSR LEFT><1 SPACE><C=64 N><CRSR DOWN>
         <2 CRSR LEFT><1 SPACE><C=64 N>"
320 PRINT "<CLR/HOME>;
325 PRINT K3$;"<6 CRSR UP>";
330 FOR I = 1 TO 12:PRINT K1$;"<6 CRSR UP>";:NEXT
335 PRINT <6 CRSR UP>";K3$
340 PRINT "<CLR/HOME>";
345 FOR I = 1 TO 13
350 IF I = 4 OR I = 7 OR I = 11 THE PRINT "<3 CRSR RIGHT>;:
    GOTO 360
355 PRINT "<4 CRSR UP><CRSR RIGHT>;K2$;
360 NEXT
370 RETURN
400 FOR I = 1 TO 24:READ NP(I):NEXT
410 POKE 56334,PEEK(56334)AND254:POKE1,PEEK(1)AND251
420 CHAR=14336
430 FOR I = 0 TO 2047:POKE CHAR+I,PEEK(53248+I):NEXT
440 POKE 1,PEEK(1)OR4:POKE56334,PEEK(56334)OR1
450 FOR I = 0 TO 7:READ T:POKE CHAR+(209*8)+I,T:NEXT
```

```
460 POKE 53272,31
470 RETURN
500 PRINT "<CLR/HOME><8 CRSR DOWN>";
510 PRINT "<5 CRSR RIGHT>ATTACK -     ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=A
512 A=T:PRINT A
515 PRINT "<5 CRSR RIGHT>DECAY -      ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=D
517 D=T:PRINT D
520 PRINT "<5 CRSR RIGHT>SUSTAIN -     ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=S
522 S=T:PRINT S
525 PRINT "<5 CRSR RIGHT>RELEASE -    ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=R
527 R=T:PRINT R
530 PRINT "<5 CRSR RIGHT>WAVE FORM -     ";:GOSUB 920:IF F=0 OR
    T>255 THEN T=W
532 W=T:PRINT W
540 PRINT "<8 CRSR RIGHT>PULSE -      ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=X1
542 X1=T:PRINT X1
550 PRINT "<8 CRSR RIGHT>PULSE -      ";:GOSUB 920:IF F=0 OR
    T>255 THEN T=X2
552 X2=T:PRINT X2
560 PRINT"<5 CRSR RIGHT>VOLUME -      ";:GOSUB 920:IF F=0 OR
    T>15 THEN T=V
562 V=T:PRINT V
570 PRINT"<5 CRSR RIGHT>OCTAVE -      ";:GOSUB 920:IF F=0 OR
    T>7 THEN T=0
572 O=T:PRINT O
580 POKE A1,A*16+D
582 POKE S1,S*16+R
584 POKE VOL,V
586 POKE P1,X1
588 POKE P2,X2
590 RETURN
900 FOR I = SID TO SID+24:POKE I,0:NEXT
910 RETURN
920 PRINT "<3 CRSR LEFT>";:A$="":F=0
930 PRINT "<C=64 P><CRSR LEFT>";:GET T$:IF T$="" THEN 930
940 IF T$=CHR$(13) THEN 980
950 IF T$="Q" THEN END
955 IFT$=CHR$(20)ANDLEN(A$)>0THENPRINT"<CRSR LEFT><2 SPACE>
    <2 CRSR LEFT>";:A$=LEFT$(A$,LEN(A$)-1):GOTO930
960 IF T$<"0" OR T$>"9" THEN 930
970 PRINT T$;:A$=A$+T$:F=-1:GOTO 930
980 T=VAL(A$):PRINT " ";
990 FOR I = 1 TO LEN(A$)+1:PRINT "<2 CRSR LEFT><1 SPACE>";:NEXT
995 RETURN
10000 DATA 1,12,1,28,1,45,1,62,1,81,1,102
10005 DATA 1,123,1,145,1,169,1,195,1,221,1,250
10010 DATA 2,24,2,56,2,90,2,125,2,163,2,204
10015 DATA 2,246,3,35,3,83,3,134,3,187,3,244
```

```
10020 DATA 4,48,4,112,4,180,4,251,5,71,5,152
10025 DATA 5,237,6,71,6,167,7,12,7,119,7,233
10030 DATA 8,97,8,225,9,104,9,247,10,143,11,48
10035 DATA 11,218,12,143,13,78,14,24,14,239,15,210
10040 DATA 16,195,17,195,18,209,19,239,21,31,22,96
10045 DATA 23,181,25,30,26,156,28,49,29,223,31,165
10050 DATA 33,135,35,134,37,162,39,223,42,62,44,193
10055 DATA 47,107,50,60,53,57,56,99,59,190,63,75
10060 DATA 67,15,71,12,75,69,79,191,84,125,89,131
10065 DATA 94,214,100,121,106,115,112,199,119,124,126,151
10070 DATA 134,30,142,24,150,139,159,126,168,250,179,6
10080 DATA 189,172,200,243,212,230,225,143,238,248,253,46
11000 DATA 1224,1105,1227,1108,1230,1112
11010 DATA 1233,1236,1117,1239,1120,1242
11020 DATA 1245,1126,1248,1129,1251,1132
11030 DATA 1254,1257,1138,1260,1141,1262
12000 DATA 251,249,248,250,251,195,195,231
```

15 The Multilingual Commodore

The Commodore 64 was designed to be extremely flexible. With 64K of RAM, it is possible to transform the computer into a very different machine by plugging in the right cartridge or booting the right disk. Fortunately, Commodore, and the independent programmers who developed many of these software packages, chose to support the unique nature of the Commodore 64.

All of the languages discussed in this chapter feature commands that deal explicitly with music and graphics. There is no attempt to make programs written in these languages compatible with other Commodore computer systems or other computers.

The computer industry changes so rapidly that by the time you read this many new products may be available. However, the products presented here are representative of what can be done with Commodore 64 software.

Simon's Basic, Commodore Logo, and Commodore Pilot are all programming languages. Tools for the nonprogrammer will be discussed next chapter, but if you've made it this far in *Commodore Magic*, you're definitely a programmer. All three systems offer enhanced programming environments, with better facilities for programmers than Commodore Basic 2.0. But to use these systems, you must purchase the new language.

Programs that incorporate the extended capabilities of Simon's Basic, for example, only work if a Simon's Basic cartridge is plugged into the Commodore 64 cartridge port. Commercial prospects for such programs are, therefore, limited. Commodore could make "run-time" packages available to commercial software developers, but whether it will is uncertain.

In a run-time package, enough of the language or operating system is included on the program disk to enable the computer to run the application program, but the language or system is not available to the user. However,

for the programmer who is developing programs for herself or himself, the advantages of these languages cannot be overestimated. Let's look more closely at each of them.

Simon's Basic

Simon's Basic is an 8K extension to regular Commodore Basic. In the United States, it is distributed in cartridge form, although the original release in England was on a disk. After plugging in the cartridge and turning the computer on (never plug in a cartridge when the computer's on), you are greeted with this message:

```
*** EXTENDED CBM V2 BASIC ***
30719 BASIC BYTES FREE
READY.
```

Notice that the free space has decreased by 8K bytes. Notice also that the description says "extended" Basic. This means that all of your programs written in unextended Basic will still run (assuming they can fit into 30K). But it also means that there is a host of new Basic commands—114 to be precise—at your disposal. The range and power of these commands is astonishing. The commands fall into the following categories:

Programming aids
Input validation and text manipulation
Extra numeric aids
Diskette commands
Graphics
Screen manipulation
Structured programming
Music
I/O functions

While we will concentrate on the graphics and music instructions, the other instructions are equally impressive. For example:

```
PRINT AT(12,8)"HELLO"
```

will print the word HELLO at the 12th column and the eighth row on the screen. Goodbye to all the clumsy cursor control codes.

```
ON ERROR: GOTO 1000
```

will cause the program to branch to line 1000 if an error is encountered. Goodbye to ILLEGAL QUANTITY ERROR and FILE NOT FOUND ERROR.

```
X=%10101010
```

will cause Simon's Basic to accept the ones and zeroes as a binary number. Goodbye to all the scratch paper and manual conversion.

You should realize from these few examples that Simon's Basic offers considerably more power for regular programming tasks than standard Commodore 64 Basic. But it is in the graphics and music areas that Simon's Basic really shines.

Hi-res with Simon's Basic

Like regular Basic, all of Simon's Basic high resolution activities must occur in programs—for split screen work (text and graphics) you'll have to turn to Logo or Pilot. Simon's Basic fully supports both regular and multicolor high resolution modes. To get into hi-res mode, the command is HIRES pc,sb where pc is the plotting color and bc is the background color. To indicate multicolor mode, follow the HIRES command with a MULTI c1,c2,c3 command; the arguments to the MULTI instruction are the colors.

After entering hi-res mode, the PLOT instruction plots a pixel, the LINE instruction plots a line, the REC instruction draws a rectangle, the BOX instruction draws a filled rectangle, the ARC instruction draws an arc, and the CIRCLE instruction draws a circle or ellipse. This is an impressive selection of drawing operations. Recall the difficulties we had drawing circles in conventional Basic. In Simon's Basic, the program becomes:

```
10 HIRES 2,7:REM ENTER HIRES WITH RED PEN AND YELLOW BKGND
20 CIRCLE 160,100,90,90*.8,1:REM DRAW CIRCLE AT 160,100
21                          :REM WITH AN X RADIUS OF 90
22                          :REM AND A Y RADIUS OF 90*.8
23                          :REM IN RED.
30 GOTO 30:REM STAY IN HIGH RES MODE
```

And that's it—no loops to contend with or difficult ANDing and ORing or PEEKs and POKEs. To make the circle solid, just add the following:

```
25 PAINT 160,100,1
```

and the circle will fill up with red, like magic. While the filling process is rather slow, it is done well.

The first pair of parameters in the PAINT command and the first two of the CIRCLE command are self-explanatory. The first set is the X and Y coordinates of the center of the circle or the center of the fill. The second set is the X and Y radius; recall that by giving different X and Y coordinates, you can easily draw an ellipse. The Y*.8 in the program corrects for the aspect ratio, just as we had to do with our more primitive circle program. The third argument of PAINT and the fifth of CIRCLE, however, are less transparent.

Since there are two colors in regular hi-res mode and four, including background, in multicolor mode, we need to know what color to use for plotting. This last argument is called "plot type," and is used to determine what color to draw in. The plot type can do even more—it can also indicate that the plot should clear a dot (make it a zero) or reverse a dot (make ones into zeroes and vice versa). These are very powerful graphics capabilities. Fig. 15.1 shows a bull's eye drawn using the CIRCLE and PAINT commands.

Figure 15.1
Bull's Eye Created with Simon's Basic

There is nothing that Simon's Basic does that cannot be done in regular Basic—but Simon's is much faster and easier to work with. Still, we've only scratched the surface of the graphics capabilities of Simon's Basic. There is a full complement of color change commands: COLOUR (the author is British), LOW COL, and HI COL. Others, such as NRM, return to the text page and erase the graphic.

DRAWing with Simon's Basic

One of the most powerful commands is DRAW, which allows you to draw a shape on the screen. The resulting shape is not a sprite, but a sequence of one-pixel moves: up, down, left, right. The figure to be drawn is created with a sequence of one-pixel plots. This sequence of moves is strung together (as a string, naturally).

For example, consider drawing an image of a floppy disk for use as a menu item. The first step is plotting the image in moves: down, down, down, down, down, down, down, left, left, left, and so on. Then, each move is translated into the equivalent DRAW number: 8 for down, 6 for left, and so on. This produces a very long number. Finally, add the quotation marks to make it a string. The program to display the shape is trivial:

```
10 HIRES 5,2
20 A$="888888666666655555555555555777777788888186657711186579"
30 ROT 0,2
40 DRAW A$,160,100,1
100 GOTO 100
```

The ROT instruction tells DRAW the orientation of the figure, which can be rotated in 45-degree angles, and the size in which to draw it. Because figures are defined with one-pixel moves, any shape can be enlarged—just multiply each number by a constant. The arguments to DRAW are quite simple. A$ (or any string name) must be a valid DRAW string; the following two numbers are the X and Y coordinates; and the last number is the draw mode. Fig. 15.2 shows the image drawn by this program.

Figure 15.2
Diskette Shape

```
10 HIRES 5,2
20 A$="8888886666666555555555557777777888881866577111186579"
30 FOR I = 0 TO 8
50 ROT I,4
60 DRAW A$,160,100,1
70 NEXT I
100 GOTO 100
```

This is a very powerful feature and greatly simplifies the creation of small figures (or large geometric ones). Since the Commodore 64 limits strings to a length of 255 characters, and program lines to 800 characters, you may have to do some string concatenation to get your entire figure into one string. Larger figures have to be drawn with separate strings. Even with these limitations, the ability to draw a figure is exceedingly useful to graphics programmers.

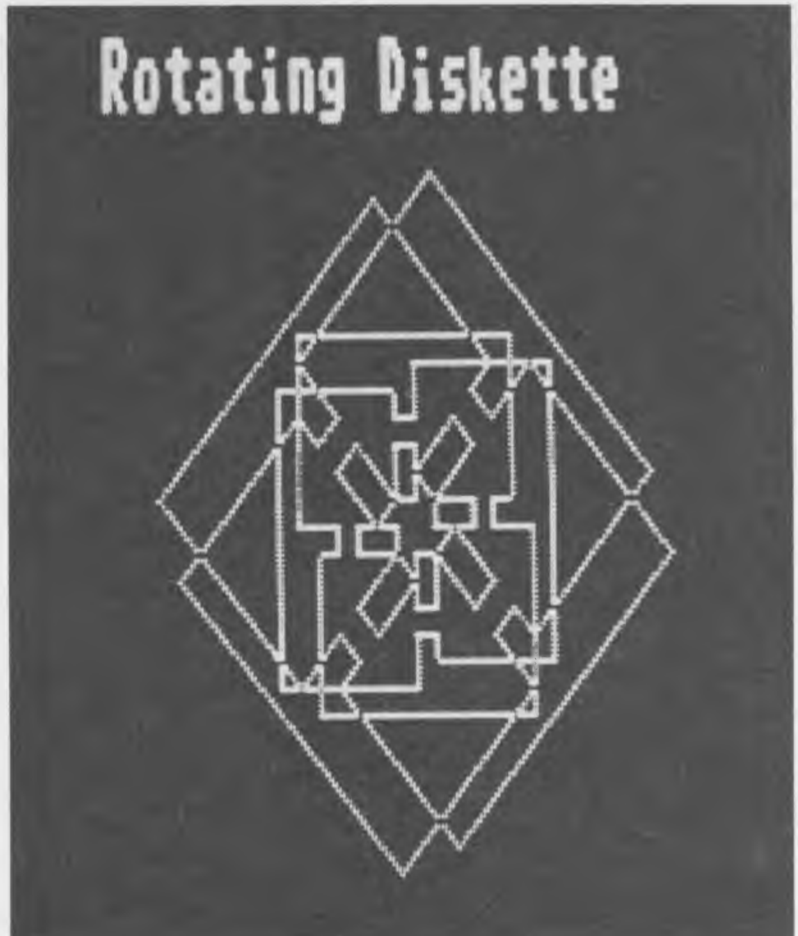
The high resolution graphics commands cannot be used interactively, and must be in programs. But Simon's Basic does allow you to place text on the hi-res screen. Addition of this line to the previous program will add a label to the screen:

```
80 TEXT 80,100,"<CTRL A>R<CTRL B>OTATING
   <CTRL A>D<CTRL B>ISKETTE",1,2,8
```

The text resides in quote marks following the X and Y coordinates that indicate where it is to appear. Three parameters determine how the text will appear: plot type, size, and the number of pixels to use for each character. Eight is the norm because Commodore 64 characters are designed in eight by eight grids.

Unfortunately, the size parameter enlarges the characters only vertically, not horizontally. However, the larger size characters are easily readable and would be useful in educational applications written for young children. The CTRL characters in the string determine whether the character will be printed in uppercase (<CTRL A>) or lowercase (<CTRL B>). Fig. 15.3 shows the result of the program above with an enlarged, rotated shape and text. Now, let's move on to hi-res sprites.

Figure 15.3
Rotated, Enlarged Shape with Text



Sprites in Simon's Basic

As in hi-res, both regular and multicolored sprites are supported by Simon's Basic commands. Designing a sprite (called MOBS—for Moveable OBjectS—in Simon's Basic) is a piece of cake:

```
10 PRINT "<SHIFT CLR/HOME>"
20 MOB SET 0,128,2,0,0
30 DESIGN 0,8192
```

```
100 @........BBBB...........
110 @......BBBBBBBB.........
120 @...BBBB..BB..BBBB......
130 @.BBBBBB..BB..BBBBBB....
140 @...BBBBBBBBBBBBBB......
150 @....BBBBBBBBBBB........
160 @......BBBBBBBB.........
170 @........BBBB...........
180 @.......................
190 @.......................
200 @.......................
210 @.......................
220 @.......................
230 @.......................
240 @.......................
250 @.......................
260 @.......................
270 @.......................
280 @.......................
290 @.......................
300 @.......................
```

That's all there is to it. The MOB SET instruction sets up a MOB. In this case, we've created MOB or sprite 0, in memory block 128, in color 2, with a priority over the screen data, and in high resolution (as opposed to multicolor). The DESIGN statement actually causes Simon's Basic to read the next 21 lines as a sprite design. It specifies sprite 0 to be located in block 128 (or 8192 in full decimal format).

The design statements that follow (@ . . .) have a B where a pixel is to appear and a dot where the pixel is to be shown in the background color. Multicolor sprites are equally easy to construct. There are half the dots across in the @ . . . statements because these have half the resolution of a normal high resolution sprite. Further, three letters are used: B, C, and D, to indicate the color of each pixel.

Moving the sprites is just as simple:

```
400 MMOB 0,0,0,100,200,0,10
410 MMOB 0,100,200,300,10,0,200
```

The first statement moves sprite 1 from 0,0 to 100,200 while displaying it in a size of 0 (normal) and at a speed of 10. Speeds range from 0 to 255, with 0 the fastest. The second statement once again moves sprite 0 from 100,200 to 300,10 in normal size at a much slower speed.

There are simple commands that bounce a sprite between two screen locations or check for collisions. The sprite capabilities here are truly superb. It should be noted, however, that the programmer must still partition

memory and beware of conflicts between the user program or data and the graphics data. However, it seems much less difficult when so many other problems have been solved.

Making Music with Simon's Basic

The Commodore 64's music capabilities are also enhanced in Simon's Basic, although not quite as spectacularly as graphics. There are direct instructions to set the volume, the waveform, and the envelope. No more POKEs to SID:

```
10 VOL 10
20 WAVE 1,0010000:REM VOICE, WAVEFORM BYTE IN BINARY
30 ENVELOPE 1,10,0,10,0:REM VOICE,A,D,S,R
```

Once these parameters are set, we must write the music to be played. This is accomplished much like a DRAW string; each component of the string has three parts: octave, note, and duration. Sound familiar?

```
40 MUSIC 10,"<SHIFT CLR/HOME>C4<F7><SHIFT C)4<F7>D4<F7>
             <SHIFT D>4<F7>E4<F7>F4<F7><SHIFT F>4<F7>"
50 PLAY 1
```

The idea of music as a string is terrific. The number refers to the length of a beat with 1 being the longest and 255 being the fastest. When the string is played, all note durations are based on this master beat. The <SHIFT CLR/HOME> indicates the start of the string. Notes are specified by letter, with shifted letters representing sharps. There are no flats; instead the lower note must be sharped. Octaves are specified by number, and duration by function keys. Really! Function keys.

You either have to remember the durations that each function key produces, or prop the excellent instruction book in front of you. The strings are unreadable on the screen because the shifted letters and function keys produce graphics characters. This is very disappointing after the ease of working with sprites. On the other hand, you can incorporate music in a program without much difficulty.

The PLAY command is what actually turns the music on. The single parameter can indicate whether to PLAY in the foreground, where the program pauses until it finishes playing, or in the background, where the

program continues while the music plays. This last capability is terrific, but will not work if your program uses high resolution graphics. PLAY 0 turns off the music.

Unfortunately, there is no multiple-voice music capability. Depending on your music needs, you may not find Simon's Basic terribly helpful. On the other hand, with all its other capabilities, Simon's Basic can be useful to you in other ways. After all, you can still write music the old-fashioned way if it suits your fancy and reap the other benefits of Simon's Basic.

Commodore Logo

Simon's Basic is similar to Commodore Basic, but has been pumped up into an outstanding language. Logo is another type of language altogether and is sold on a copy-protected disk. This means that you cannot make an extra copy of the disk to protect against disk drive failure.

Normally, I am opposed to copy-protection of system and utility software. But Commodore charges so little for Logo (in comparison to Logo for other personal computers) and will replace a disk that gets eaten by a drive for $5, so this practice may be justified.

Logo was developed at the Massachusetts Institute of Technology (MIT) in the 1960s as a language to help young people learn mathematics, and provide an environment for research on how people learn. Since then, it has become a powerful, full-fledged, computer language that is remarkably easy to pick up, though you still have to learn its programming rules and commands. Friendly error messages are still error messages, but learning Logo is far easier than struggling with the syntaxes of other languages.

Logo is far more than just music and graphics, but we will concentrate on its abilities in these areas.

Programming in Logo

Logo has one basic type of data: a word. A word is a character or group of characters separated by spaces. Type a word and Logo tries to find that word in a dictionary of things it knows how to do. If it doesn't find the word there, you'll get an error message. Programming in Logo consists of teaching Logo new words for its dictionary.

In Basic you can write a program to play a song. You can even save it with the name PLAY. But to get the Commodore 64 to actually play the music, you must type RUN. The program and Basic are separate entities. In Logo, you can write a music program and call it PLAY, but to actually play the music, you type PLAY. There's no need to say RUN; Logo knows to look for instructions under the name PLAY and knows you want the music played. This is only one of the important differences between Basic and Logo. You'll see others in the examples that follow.

Hi-Res

Logo supports both regular hi-res mode and multicolor mode. The Logo instruction to turn on one or the other mode is SINGLECOLOR or DOUBLECOLOR. When the switch is made, the screen is erased. The Commodore 64's 16-color palette is available, but the usual restrictions apply. The basic screen unit in either hi-res mode is the eight by eight character matrix. Within that matrix there can only be two colors in regular mode or four colors in multicolor mode. These are hardware restrictions imposed by the VIC II.

It is easy to change drawing colors—the PENCOLOR command does it for you. Similarly, the background color can be changed with the BACKGROUND command. These commands can be abbreviated to PC and BG respectively. Many other Logo commands also have abbreviations. In this section, I'll always spell out the whole command.

Curiously enough, you cannot control the border color. While this may seem inconsistent with the other excellent graphics facilities, there is a reason. Logo has three screen modes: all text, all graphics, and split screen. The split screen mode shows five lines of the text page on the lower portion of the screen and 20 lines (or 160 pixels) of graphics on the upper portion of the screen.

To do this, the Logo programmers had to create, in essence, a new VIC II mode in software. In doing so, the ability to have a border color different from the screen color was lost. But this is minor penalty for being able to see your graphics commands being executed; the restriction of being able to do hi-res only is gone. The screen mode can be switched by pressing one of the function keys; <F1> gives you all text, <F3> gives you a split screen, and <F5>, all graphics.

Turtle Graphics

Graphics in Logo are called **turtle graphics** because all graphics instructions are issued as if you were commanding a robot to "walk through" an image. The original Logo, at MIT, had a real robot that looked something like a turtle. The robot wheeled around the floor and drew with a retractable ball-point pen in its belly. When video terminals became common, in the late 1960s, the turtle migrated to the screen, but the graphics commands remained unchanged.

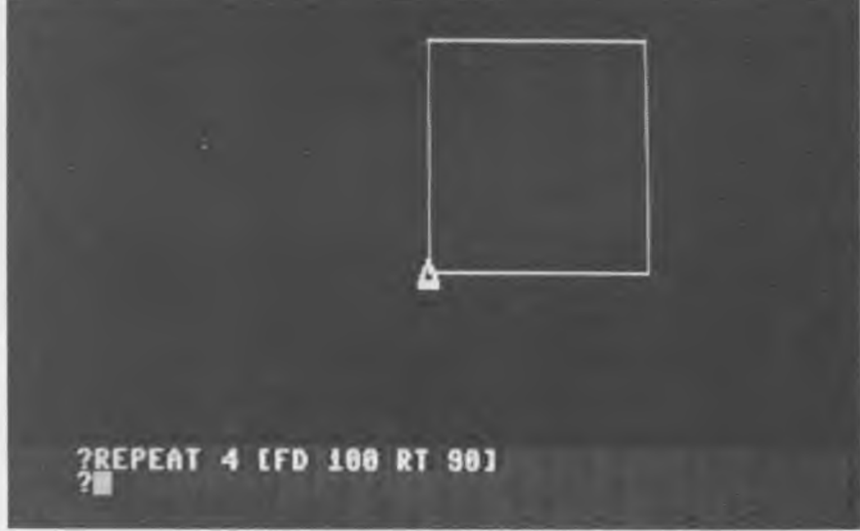
The basic graphics commands are FORWARD, BACK, RIGHT, and LEFT. FORWARD and BACK require a numeric argument that represents the number of steps (in pixels) that the robot is to take. RIGHT and LEFT require a numeric argument that represents the number of degrees the robot is to turn. The following sequence of commands would cause Logo to draw a square:

```
FORWARD 100 RIGHT 90 FORWARD 100 RIGHT 90
FORWARD 100 RIGHT 90 FORWARD 100 RIGHT 90
```

Abbreviations make this less formidable to type, but using the REPEAT command makes it even simpler (see Fig. 15.4):

```
REPEAT 4 [FORWARD 100 RIGHT 90]
```

Figure 15.4
Logo Square. Note the turtle in the lower left corner of the square.

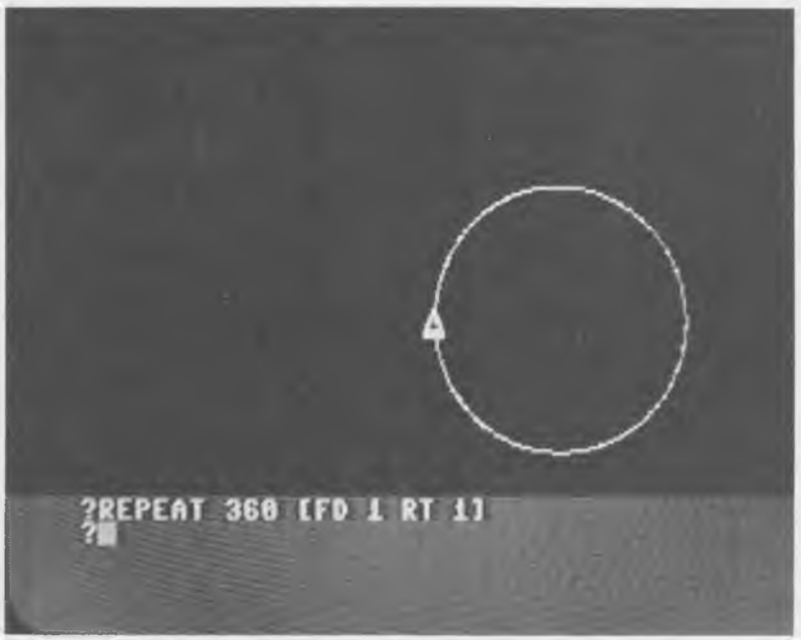


This short example shows the beauty of Logo. Given the premise that you are ordering a little robot around, programming in Logo becomes a very physical event. Logo instructions were carefully chosen to accommodate the short memory spans of children.

Compare the syntactical requirements of Basic's FOR/NEXT loop with the REPEAT loop in Logo. In Logo all you have to get right is the spelling of the word REPEAT, a number, and square brackets around what you want repeated. Probably the most important difference between Basic and Logo is the lack of an arbitrary index variable in the REPEAT. The counting variable in a Basic FOR/NEXT loop is such a source of confusion—and bugs—for beginners.

A circle is as easy to make as a square (see Fig 15.5).

Figure 15.5
Logo Circle



If you are more comfortable with the traditional graphics grid, Logo also allows you to SETHEADING, SETX, SETY, and SETXY. These instructions, in conjunction with the relative FORWARD, BACK, RIGHT, and LEFT, provide very complete control over the graphics screen.

Defining New Logo Words

The instruction to tell Logo to store the next sequence of instructions under a name is TO. To transform our square into a Logo procedure or program (as opposed to a some of Logo instructions that draw a square), we would enter:

```
TO SQUARE
  REPEAT 4 [FORWARD 80 RT 90]
END
```

To draw the square, type:

```
SQUARE
```

and we'll see the same square we would have seen had we typed the whole set of instructions. The new Logo procedure SQUARE is not available for use in additional procedures. It's possible to make a rather spectacular display by creating the procedure

```
ROTATE:

     TO ROTATE
       REPEAT 36 [SQUARE RIGHT 10]
     END
```

This innocuous procedure fills the screen with squares, all emanating from the turtle's initial position and differing from each other by their initial orientation. (Each starts ten degrees to the right of the next.) Incidentally, it's REPEAT 36 because everything in Logo centers around 360 degrees. I knew that with ten degree changes, I'd have to repeat the square 36 times. By reducing the amount of turn, it's possible to generate some very interesting moire effects caused by color artifacting in SINGLECOLOR mode.

The effect of the TO command can be startling to the new user. When you type "TO SQUARE" and press <RETURN> the screen changes dramatically—the screen clears and a message appears at the bottom of the screen in inverse text that reads:

EDIT: CTRL-C TO DEFINE, CTRL-G TO ABORT

Logo includes a full-featured screen editor. It is a line editor in that the cursor jumps in line units. But it features character deletions and insertions. It makes procedure development and modification a joy. There are no line

numbers in Logo, so an editor is necessary. But the variety of features is a real bonus. After entering your procedure, you type <CTRL C> (or <RUN/STOP>) and the procedure definition is entered into Logo's dictionary.

If you realize that you have made a mess of it and don't want to enter the procedure into Logo, <CTRL G> will exit the editor without a dictionary entry. If you've ever had to delete a large number of lines in Basic, you'll appreciate this editor. Fig. 15.6 shows the square procedure being entered in the Logo Editor.

Figure 15.6
Logo Editor

Variables in Logo

Logo can also handle variables. These variables are called "dot names" because the actual name always includes a colon. :SIZE is an example of such a name; the colon is part of the name and is mandatory. Without the colon,

Logo would look for the procedure SIZE and complain that THERE IS NO PROCEDURE NAMED SIZE. Here's what the square procedure would look like with a dot name for the size of the square:

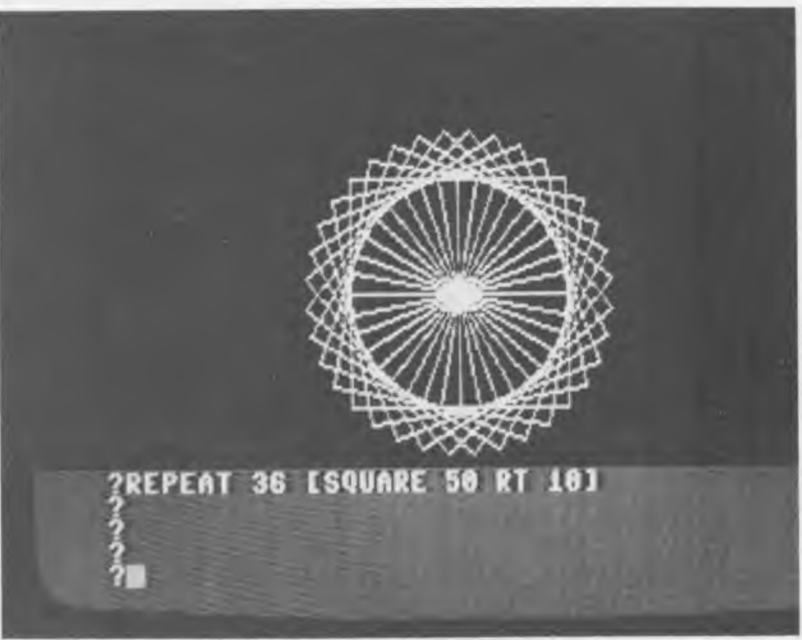
```
TO SQUARE :SIZE
  REPEAT 4 [FORWARD :SIZE RT 90]
END
```

And to call it, we'd now have to say:

```
SQUARE 10 or
SQUARE 50 or
SQUARE 100
```

Notice that the colon goes only in front of the name of the variable, not in front of the value you intend to be assigned to it (see Fig. 15.7). If you just type SQUARE, Logo would complain SQUARE NEEDS MORE INPUTS. Logo's variables are more complicated (and powerful) than Basic's. X=10 in Basic becomes MAKE "X 10 in Logo. It takes time to master the subtleties of Logo's variables.

Figure 15.7
Rotating Square



Sprites in Logo

Logo also has a set of sprite commands. In fact, the graphic depiction of the turtle that you see when doing Logo graphics is a sprite. The turtle commands affect only the active turtle. There can be up to eight turtles or sprites on the screen at any one time. Only the active one is actually drawing; the others remain stationary. To activate a second turtle, the Logo instuction is TELL. Thus

```
TELL 1
```

would direct all further turtle graphics commands to turtle 1 (or sprite 1) and the normal turtle (0) wouldn't move. This is quite different from Logo sprite graphics on other computers. It is very exciting to see a whole swarm of sprites dashing about the screen.

The graphic hardware of computers does differ from one computer to the next, and this is a reasonable compromise. Only the main turtle—turtle 0—shows the physical rotation of the shape on the screen as it is given LEFT and RIGHT instructions. The other turtles maintain the orientation they had when they were designed. (Remember the 24 by 21 bit map for sprites.) Nonetheless, they do respond to all the commands.

Like all sprites, these sprite turtles have priorities, with sprite 0 (the main turtle) having priority over all others. Sprite 1 is next in line, and can be hidden only by sprite 0, and so on. The SETSHAPE command allows you to change priority by changing the shape carried by a sprite. While such things are possible, even simple, in Basic, it's so much simpler in Logo with virtually no need for PEEK or POKE. If you really need them, however, they're here in the guise of .EXAMINE and .DEPOSIT. Note the period—one more example of Logo's attempt to protect the user from dangerous bugs.

This is only a very cursory look at Logo's graphics commands. The Commodore version is one of the best implementations available. In addition to all the goodies discussed above, you can SAVEPICT to a disk and READPICT from a disk, BSAVE and BLOAD character sets or other binary information, STAMPCHAR(acters) onto the graphics page, draw in PENERASE mode, and more. With.EXAMINE and.DEPOSIT, other capabilities of the Commodore 64 are available for advanced programmers. Like music.

Logo Music

There are no music commands in Commodore 64 Logo. Instead, Commodore supplies a disk called the "Logo Utilities," which contains a variety of useful Logo files and many demonstration programs. Among them is one called "MUSIC" that contains a gaggle of procedures that enable you to write single-voice music without much trouble. Like Simon's Basic, multivoice music is still a programmer's domain. Once this file is read into the computer, you need only enter:

```
ATTACK 10
```

to set the attack to 10. All the ADSR and waveform parameters are equally supported. This is handy, but the problem of how to present the music data (notes) remains. Logo provides three procedures for this: PLAY, SING, and NOTE.

SING requires a list of notes and plays them at an equal tempo. So:

```
SING [1 2 3 4 5 6 7 8 9 10 11]
```

plays a scale

PLAY requires two lists of numbers. The first list indicates the pitches of the notes and the second their durations. For example:

```
PLAY [1 2 3 4 5 6 7 8 9 10 11]
     [5 10 15 20 25 30 25 20 15 10 5]
```

will play a scale, with the duration of the notes first lengthening, then diminishing.

Both of these use the simpler NOTE procedure, which plays one note in a given pitch, waveform, and envelope.

It would have been better to have made these procedures a resident part of Logo, but they are easy to read off the disk, and, since not all programs use music, they provide valuable space for your own procedures. There is enough free space in the Commodore 64 Logo memory map to allow an assembly language programmer to write multiple-voice routines that can be called from Logo.

The Logo package—including the variety of procedures provided on the Logo Utilities Disk—is very impressive. Many of the Commodore 64's features that are difficult to use in Basic are accessible under Logo, so it's worth looking into.

Commodore Pilot

If Logo is the language for learning, Pilot is the languge for teaching. Pilot (like Basic) is an acronym. Pilot stands for "Programmed Instruction for Learning or Teaching." Both are programming languages, but their initial scopes differ radically. While Logo was first intended only for children, Pilot was first intended only for teachers. Today, however, both have evolved into general-purpose languages. Pilot is included here because, like Logo and unlike Commodore Basic, it allows easy creation of graphics and music.

Pilot, like Logo, is distributed on a disk. It too is protected from copying but is also reasonably priced, and the disk is backed up by a $5 exchange offer from Commodore if you damage it.

Using Pilot

Like Logo, Pilot takes over your computer. It no longer understands PRINT—that's Basic. Instead, Pilot possesses a different set of commands and syntax, which were created to allow nonprogrammers (such as classroom teachers) to develop instructional sequences. Commodore Pilot, can, of course, accomplish this.

Further, unlike both Simon's Basic and Logo, Pilot offers a run-time version that allows a user (perhaps a student) to execute a Pilot program but to have access to the program development tools. This is a very big plus in Pilot's favor. But we are concerned mainly with the graphics and music capabilities of the language, and they are considerable.

When you first bring Pilot up, it is obvious that something is different, even more than the appearance of the prompt and cursor. The character set looks different. It is cleaner and easier to read. As you begin to explore the world of Pilot, it is apparent that everything is done in the normal hi-res graphics mode. This means there is complete mixing of text and graphics. There must be a penalty in speed, but it is not discernible.

Text is put to the screen very rapidly, and all the draw operations are equally fast. The only problems lie in the number of colors available in normal hi-res mode. Recall that each eight pixel by eight pixel block of the screen must have two colors only: background and plot. Thus, doing graphics over text is fine, until the color is changed. Then the whole eight by eight block changes, possibly making your text unreadable.

When Pilot is started, the Commodore 64 displays PILOT: as the prompt. At this point, you can enter one of several modes—edit mode to write a program, run mode to execute a program, or immediate mode to test

program statements or do interactive graphics. You can also load or save a program from Command (PILOT:) mode, or restart Pilot, or start Basic.

Programs with modes may be difficult for some people. Because each mode features a different set of commands, you must always know the current mode. A command in one mode may do something entirely different in another. In Logo, you need only be concerned with two modes: edit and immediate. In Basic, there's only one: immediate. In Pilot, there are three.

We are most interested in immediate mode, although if you were developing programs to run in Pilot you would need to master edit mode. Edit mode is quite good. Pilot programs, like Logo programs, have no line numbers, so an editor is necessary. Like Logo, the Pilot editor is quite good—as full-featured as one could expect. Unlike Logo, however, the key functions are more comparable to their use in Commodore Basic. Many people find it difficult to remember different key functions. The Pilot editor may, therefore, be more comfortable to Basic programmers than the Logo editor.

Pilot Commands

Since Pilot was intended for novice computer users, it is very simple to use. There are only 18 commands, all abbreviated to their first initial, for example, T:. Pilot, like Logo uses colons—but Pilot uses them to separate the command from the argument of the command. For example:

t:Hello, this is my first line of Pilot.

prints (types—t:) the line "Hello, this is my first line of Pilot.". Pilot uses an upper- and lowercase character set. Logo is jut like Basic in regard to the keyboard character set—all the graphics characters are there. Pilot uses the standard ASCII character set, not the Commodore 64 character set. However, Commodore Pilot offers one of the simplest character redefinition commands possible:

```
N:65
...XXXX.
..XX..XX
..XX..XX
..XXXXXX
..XX..XX
..XX..XX
..XX..XX
........
```

This will redefine the letter A (ASCII number 65) as a much thicker character. The change stays in effect until Pilot is restarted. Sprites are similarly defined, but they use a 24 x 21 dot grid, with the B:sn instruction where sn represents the sprite number. Unlike Simon's Basic, however, you must be consciously concerned with bit patterns: xx, xo, ox, and oo make the different colors of a multicolored sprite. (In normal sprite mode, of course each "x" represents a pixel.) Such simplicity should be found in Basic.

Pilot Graphics

The graphics command is a g: or s: (s: for sprites). Following the instruction, there is a bewildering array of options (arguments). These include:

bn	background color n.
cn	foreground color n.
dx,y	draw from current location to x,y.
e	erase screen to current background color and change current x,y to 0,1
xn	exterior color (border) n
fx,y	fill rectangle from current x,y to x,y
mx,y	move to x,y from current x,y with no drawing
px,y	plot at x,y and set current x,y here
sn	show split screen (n determines amount of text to show)
gx,y	erase point at x,y and set current x,y here

In Pilot, a square looks like this:

```
g:p10,10
g:d100,10
g:d100,80 (remember the aspect ratio—Logo took care of it)
g:d10,80
g:d10,10
```

This is not particularly difficult. The first command g:p sets the initial plotting point. The next line is drawn (g:d) from 100,10 to 100,10, and so on, until a cube is drawn. What's nifty about this is that you can type in each command and watch the square get drawn. In Basic, you cannot—you must

write the program and run it. If there's a mistake, you've got to edit the program and rerun it. You can't even keep Basic in hi-res mode without a 30 GOTO 30 or a 30 GETA: GOTO 30. In Pilot, you are there—each command is executed as you type it in when you are in immediate mode. Get it right, and then you can enter it for posterity in the editor and save it to the disk.

This is almost enough power but not quite enough. If you have sophisticated graphics neds, Pilot will be strained to provide them. You can't draw circles easily because Pilot has limited looping facilities. Similarly, other much desired graphics features cannot be easily programmed except by experts in machine language. There is sufficient documentation for assembly language subroutines to be used effectively; it's too bad that they are not as easy to write as Pilot programs.

Pilot Sprites

Unlike hi-res graphics, sprites are very well supported. There is a Pilot argument (to follow the s: instruction) that handles nearly every aspect of sprite graphics—size, color, multicolor, location, and more. All have the appropriate arguments. Unfortunately, there is no all-purpose sprite move command. Instead, sprites must be moved in loops as they are in Basic and Logo. Simon's Basic has the edge here. Nonetheless, having the sprites available with one simple command is infinitely better than unenhanced Basic.

Music and Pilot

Pilot's support of music leaves much to be desired. It's all there, but you must remember the music register that you wish to change. Music commands are issued through the V instruction. This instruction is equivalent to SID = 54272 in Basic; all instructions are given as offsets to SID.

For example, to turn on voice 1, V:0,16;1,195 will play a C in octave 4. These are just the Basic POKE values: POKE SID + 0,16:POKE SID + 1,195. Other music commands are similar. In fact, Commodore Pilot provides a VX: instruction which can POKE anywhere in memory, without the SID constant added to it. Primitive, yet somehow V: seems better than the Basic equivalent of SID = 54272. At least the instruction's name implies voice.

Pilot, then, has less flexibility than Simon's Basic or Logo. Yet, it has its niche. With a run-time package, it is the only language of the three that is available for true commercial development. Further, there is a reasonable body of public domain Pilot programs that use only the basics of the language and can be easily ported to your Commodore computer. These plain vanilla programs can then be modified by adding sprites and customized characters to them. But most of all, there is high level support for graphics—if not for music—that is much needed if one is to explore microcomputers.

Simon's Basic, Logo, and Pilot are only three of the available alternatives to standard Commodore Basic. Other languages, such as Forth, could provide an equally rich environment for graphics or music programmers. The popularity of the Commodore 64—and the shortcomings of its Basic—assure continued development of additional languages and systems.

16 Software Tools

The title of this chapter is derived from a classic work by Brian Kernighan and P. J. Plauger that deals with the development of useful routines that can be saved and incorporated into programs whenever they are needed.

Similarly, *Commodore Magic* has presented subroutines that are general enough to be useful outside of the specific program for which they were written. But there's another kind of software tool—commercial products that enable us to focus on the art, animation, and music, rather than on the programming.

The editors developed in this book are steps in this direction. In fact, they may provide you with enough support that you don't need to purchase additional software. But there are advantages to commercial programs. One is that most of the time-critical routines are written in assembly language so they are fast. Second, the best of these programs offer very specialized functions. Given the time that it takes to develop and program these functions, the price of the software may not be an important consideration. Time is money, after all.

There are dozens of programs that could have been included in this chapter but most are fairly primitive and not different enough from our editors to be useful. The programs in Commodore's Disk Bonus Package are a good example. By the time you read this there will, no doubt, be more programs available.

Your local computer users group is one of the best places to get advice about good software. This group may even offer a graphics or music subgroup, called a **special interest group** or SIG, whose members have expertise in these areas. In any case, the programs discussed below are representative of the best available at this time.

Evaluating Software

How should you evaluate software? This is a thorny issue. Reviews in computer magazines range from being totally uncritical to being so cynical that only the best programs get mentioned at all, leaving many good, serviceable programs undiscovered. It is important that you be aware of the critera on which the software presented here was evaluated, so here are mine:

Delivery of Goods. A program should do what it says it is going to do. If it promises easy animation, it should deliver easy animation.

Ease of Use. A program should be easy to use. If possible, a help screen should be available. Command keystrokes should be logical and fit within the norms for the computer even if those norms have been established de facto.

Recovery from Errors. There are two kinds of errors, user errors and program bugs. While a commercial program should be bug-free, some bugs do filter through the testing phase. Because everyone knows this is going to happen, the program should offer some method of recovering from bugs without losing your data. Programs in Basic, for example, might provide a GOTO 1000 that will enable the program to resume without initializing the data. Similarly, machine language programs should provide SYS call that will do the same thing. If it is the user who makes a mistake, such as trying to save when there is no disk in the drive, the program should inform the user of the error and resume operation with no loss of data.

Documentation and Support. Along with the program, the authors should provide adequate instructions where all program options are explained. If the program is complex enough, a tutorial should also be provided. The documentation should provide an index or very complete table of contents that could serve as an index when necessary. Quick reference cards or keyboard overlays also add value to complex programs.

These criteria are by no means exhaustive. Everyone has his or her own idea of what a program should do and how it should behave. With these points in mind, let's look at some utility software.

Sorcerer's Apprentice

Sorcerer's Apprentice is a high resolution screen creator. It allows you to paint high resolution screens with a joystick or with the keyboard. Its initial operation is not unlike our hi-res editor, but a very complete set of additional capabilities take it well beyond what we've done.

The initial screen shows a cross-hair cursor smack in the center of a grey background. The border color is red. Like our editor, this indicates the current draw color. The joystick or the cursor keys move the cursor around. Pressing the joystick button or the space bar puts a point on the screen at the current cursor location. so far nothing dramatic. Fig. 16.1 shows a screen created with *Sorcerer's Apprentice*.

Figure 16.1
Graphic Created with *Sorcerer's Apprentice*. Notice the crosshair cursor in the upper right-hand corner of the image.

The dot represents more than just a painting dot although this is one of its functions. It is also the initial point for several special functions accessed from the function keys. These functions include lines, boxes (frames or solid), circles (frame or solid), and triangles (frame or solid). The initial point set by pressing the joystick button and the current point indicated by the cursor position serve as parameters for these functions.

For example, pressing the button when the cursor is in the center of the screen, and then moving to an edge and pressing the <F1> key draws a line

from the center (the initial point) to the edge of the screen (the current point). Moving the cursor and pressing the <F1> key again will draw a second line originating at the center of the screen and going to the new current point. Such radiating lines are very simple to draw.

Similarly, concentric circles or frames are also easy to draw with *Sorcerer's Apprentice*. Most of the functions require two points: for circles, center and circumference; and for boxes, opposite diagonal points. Triangles require three points, and the program uses the last two initial points in combination with the current position to create a solid or empty triangle. The only problem with these functions is remembering which key does what.

Fortunately, pressing <H> causes the drawing page to disappear and one of three help screens to appear. Thus, given a little practice with the different functions, you're never completely lost—help is only a key away.

Sorcerer's Apprentice offers a paint mode. Accessed by the <P> key, the paint mode allows you to use a predefined set of nine brushes for drawing. The brush shapes include an airbrush that looks like lots of dots, some solid boxes from small to big, an empty circle, and the cross-hairs. In dot-plot mode, the cursor shape is irrelevant to what gets drawn, but in paint mode, all the functions are executed with the current brush shape. A circle, then, can be made quite thick by choosing a big brush.

You can choose colors by using the <A>, <S>, and <D> keys, each representing one of the multicolor hi-res color patterns. Changing the color is done by pressing the <U> key—the border steps through the available colors. The <@> key changes the background color. Shifting the <U> or <@> keys changes the direction of the stepping. If you overshoot your desired color by one, you can get back to it by shifting the key rather than stepping through all the other 15 colors again.

Enclosed areas of background color can be filled by pressing the <SHIFT F> key when the cursor is positioned inside the area to be filled. Previously filled areas cannot be refilled.

Magnify Mode

The program functions described so far are good but hardly spectacular. We accomplished much of this in a small *Commodore Magic* Basic program. But *Sorcerer's Apprentice* offers three very sophisticated functions that are the hallmark of a well-done graphics program. These are a magnify mode, a screen image mover, and a text mode.

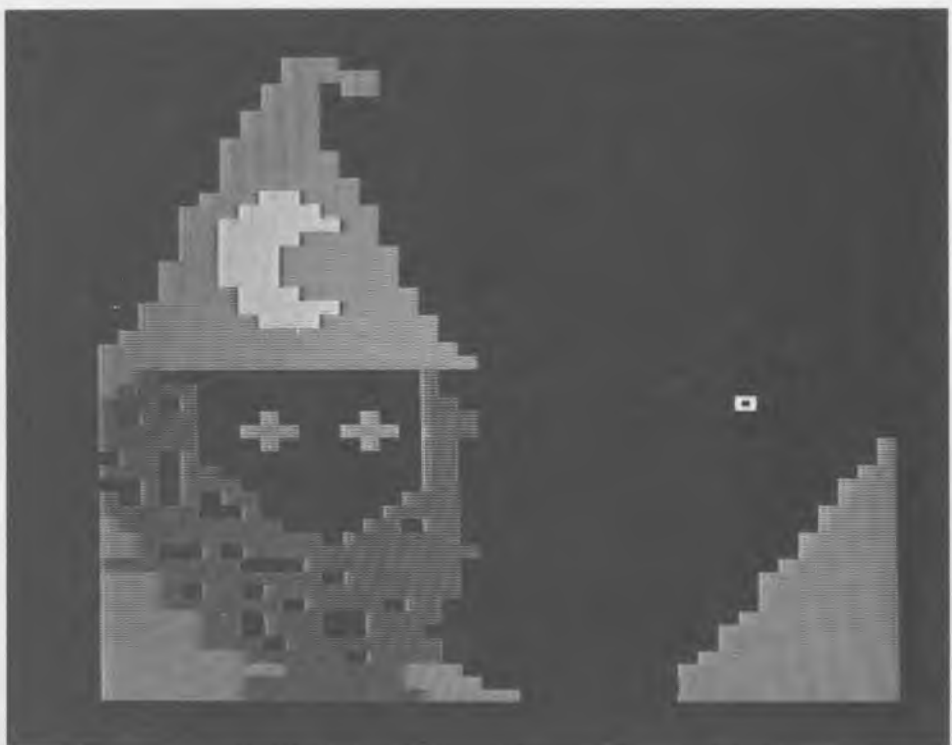
The magnify mode is quite important when you are trying to create detailed, professional-looking graphics. In normal drawing mode it's hard to

get every pixel where it should be. By using a magnification mode, the artist can touch up his or her images so they are as perfect as the application requires they be. <CTRL W> displays the magnification window. The upper left-hand corner is the current cursor location.

Pressing <RETURN> selects the magnified view, and the screen suddenly changes to a greatly enlarged image (see Fig. 16.2). The cursor is now an open box, the size of the magnified pixels. All the functions work within this window, but in dot-plot mode, not paint mode. Drawing a circle in the magnified window will result in a circle. If the circle extends beyond the window, it is clipped and will appear as an arc in the unmagnified view.

The magnified window can be scrolled up, down, left, or right with the <I>, <M>, <J>, or <K> keys in combination with the <CTRL>, <Commodore 64>, or <SHIFT> key. Apple users will be comfortable with this because these are the Apple cursor movement conventions. The amount of scrolling depends on the combination of keys. Pressing <CTRL W> while in magnify mode returns the screen to the nonmagnified view.

Figure 16.2
Magnify Mode in *Sorcerer's Apprentice*



It's difficult to write reassuringly about these functions, as they tend to sound so complicated. The implementation of magnify in *Sorcerer's Apprentice* is one of the best I've worked with. It's fast and easy to use. And best of all there is an "undo" feature. Pressing <C=64 W> rather than <CTRL W> when in magnify mode restores the original appearance of the page if you have not scrolled the magnify window.

Move and Text Mode

The screen image mover function is equally impressive. This function is accessed by the up-arrow key. A rectangle formed from the initial point and the current cursor position can then be maneuvered around the screen. Pressing <RETURN> anchors the rectangle. Pressing <SHIFT RETURN> aborts the move. This feature allows repetitive images to be created easily; one application that springs immediately to mind is fabric design.

Sorcerer's Apprentice goes even further, allowing "transparent" moves in which either the piece being moved or the background has color priority. These are very poweful and useful capabilities especially in combination with the magnify mode. Fig. 16.3 shows move mode in operation.

Figure 16.3
Move Mode in *Sorcerer's Apprentice.* The wizard's head has been captured and moved.

The text mode is less successful, but it is useful enough for many purposes. No font is automatically loaded with the main program, so to use this you must first load a font. Only one font is provided on the distribution disk. Called ASCII.FONT, it is relatively pedestrian.

The author of *Sorcerer's Apprentice* chose to stay within the Commodore 64's eight by eight dot character set, which in combination with the resolution limitations of multicolor high resolution, makes it difficult to produce a readable font. A better solution would have been to provide a larger character grid: 16 by 16, for example. This would have reduced the screen to 20 characters by 10 lines, but the characters would have been more fully formed and decorative fonts would have been possible. Still, a primitive labeling ability is available.

Once loaded, pressing <DEL> or <INST> enters typing mode. Typing can either be destructive or overlay. In the former, the image underneath the text will be destroyed. In the latter, the image is eXclusive ORed (XORed) with text. Pressing <INST> or <DEL> returns you to normal drawing mode and, once again, activates the keyboard for commands rather than for text. Pressing <CTRL E> will display the entire font and allow you to edit it using all the normal draw functions, which really must be done in magnify mode. <CTRL F> saves your changes into the character set.

All of this capability would be lost without the ability to save and load images and fonts. These functions are accessed by <CTRL S> and <CTRL L> respectively. This part of the program, like the text section, is less than well thought out. There is no cursor on the screen, so it is hard to correct an error in typing. Furthermore, corrections are made with the <CRSR LEFT> key rather than with the <INST/DEL> key as is the custom with the Commodore 64.

There is no way to get a directory of the disk, which means you might save an image with a name that is already in use. *Sorcerer's Apprentice,* in this case, reports a DISK ERROR with no other indication of what the problem might be. Further, the error channel is not cleared—the disk drive light flashes. This directly disregards warnings included in the 1541 disk drive manual. If there is no error, pictures and fonts are correctly saved. I hope that the author will clean up the disk operation portion of the program by the time you read this.

Once saved, a small machine language program on the *Sorcerer's Apprentice* disk can be used to load and display the picture with no damage to your program. Using this small routine requires three POKEs and a SYS call as

well as the LOAD of the picture, using the normal LOAD "name",8,1 instruction. Three more POKEs will restore the text page. This program can be copied to your disks, although there is no mention of how to do it in the *Sorcerer's Apprentice* documentation.

The program documentation consists of a tiny seven-page manual. All the features of the program are explained, but without the help pages in the program, it would be a real chore to find the information you need. With the help pages, the total documentation is barely adequate.

Despite these reservations, I remain impressed by the *Sorcerer's Apprentice* and have found it very useful in my work with the Commodore 64.

Spritemaster 64

This program, as its name implies, is a sprite editor. With the bare adequacy of the *Sorcerer's Apprentice* documentation still fresh in my mind, let me say at the start that *Spritemaster*'s documentation is superb. However, using sprites is more complicated than loading a hi-res image, and I would demand better documentation. The *Spritemaster* manual is a 25-page typeset booklet and includes a tutorial, a command description section, and, best of all, a sprite reference guide for programmers.

The program offers many options including Build, Modify, View, Copy, Delete, Purge, Load, Save, Transfer, and Animate. Unlike *Sorcerer's Apprentice, Spritemaster* is menu driven; you select an operation from a list of choices on the screen. There is no need for help screens if the menus are complete enough. *Spritemaster*'s are, for the most part.

Spritemaster Options

If you have no existing sprites, you must start with the Build option. The basic sprite editor is similar to our editor. It functions in both multicolor and regular modes and is simple to use. The cursor (not "curser" as appears throughout the program, but not in the documentation) is moved with the joystick. Personally, I prefer the keyboard and, sure enough, *Spritemaster* offers the <S>, <SPACE>, <Z>, and <X> keys for up, down, left, and right. Colors are selected with the function keys.

After building a sprite, you can modify it or see what it looks like when it

is animated. Colors can easily be changed, although you cannot change a multicolored sprite into a regular sprite or vice versa. If you discover your sprite requires editing, the Modify choice can be used to alter its appearance.

View allows you to see all the sprites in groups of three and modify some of the parameters, such as X and Y magnification and color.

Copy copies an existing sprite to another. This may not seem too useful, but is is critical to what *Spritemaster* does best.

Delete and Purge allow you to remove one sprite or all of the sprites respectively.

Load and Save allow you to do just that. A version is available for cassette users so there is a little more flexibility than *Sorcerer's Apprentice* offered. Sixteen sprites may be in memory at any time. The sprite files are saved as binary files from the sprite memory; up to 10 "sections" can be saved. The main use of these functions is to save sprite data in a form that *Spritemaster* can use.

There is no problem with file names here—the program manages them. There does seem to be a problem with error protection, though. With no disk in the drive, *Spritemaster* cheerfully informs you that it is loading sprites while the disk drive light blazes away. Weird. The same thing happens on a save. Users should be more careful about correctly labeling and managing their collections of data disks or cassettes.

Transfer is the option that makes *Spritemaster* most useful to you. Transfer enables you to create a set of DATA statements that will recreate your current sprite designs. When this option is selected from the main menu, a screenful of instructions appears. You then specify the sprites to be transferred, by area numbers, and the program asks what line number to use to start the data statements with. Like magic, the screen fills with data statements.

You are then asked if you really want to erase the current *Spritemaster* program; a "yes" will give you one final chance to abort the operation. Then, more instructions appear and entering LIST produces nothing. Not to fear, your data statements are there, hidden in memory by clever manipulation of Basic's pointers.

Now you can load your driver program, or merely type 10 REM, followed by SYS 32768. Bingo! When you type LIST again, there are your sprites all neatly arrayed as DATA statements, with five bytes per line. This program can now be saved or run. The sprites that you've created are safe and sound in the data. Of course, you must write the FOR/READ/POKE/NEXT loop to install the sprites, but most of the hard work is already done.

The Animation System

Now for *Spritemaster's* really special feature: true animation. Think of the eight (or more) sprites as frames. By changing the sprite pointers, it is possible to overlay one sprite with another. *Spritemaster's* animate option allows you to view these overlaid sprites without leaving the editor system. In their demonstration, a funny-looking multicolor man, displayed with both X- and Y-axes enlarged, walks in place. It's very impressive, but incredibly simple to do with *Spritemaster.*

In the animation section, you first create a sequence of sprites, using their numbers; for example, 1,2,3,1,4,5,1. *Spritemaster* will then know to show—at the same X and Y coordinates—sprite 1, then 2, then 3, then 1 again, and so on. If the sprite is properly designed, you can achieve excellent animation.

Since the sprites are erasing themselves, there is no need for an erase cycle, and since the VIC II takes care of vertical and horizontal movement, there is no need for a border of empty pixels. What's really classy about this system is that if you don't like what you've done you can change the sequence or go back and modify a sprite design very easily. It's hard to understand the beauty and simplicity of such a system unless you have slaved over a design on graph paper and in temporary programs just to see what it looks like moving.

Aside from the problems with the disk or tape, my major criticism of *Spritemaster* is that the instruction and menu screens occasionally misspell a word. While this does not damage the operation of the program, it does tarnish its image slightly.

Spritemaster is an adequate sprite editor coupled with a superb animation tool. However, you must be able to write a driver program to recreate the activities of the *Spritemaster* animation system. To help, Access Software provides a very thorough sprite tutorial as the last chapter of the manual, including sample programs to get you started. If you have very extensive animation in mind, *Spritemaster* can only help you get the images right. But that is all it is touted to be.

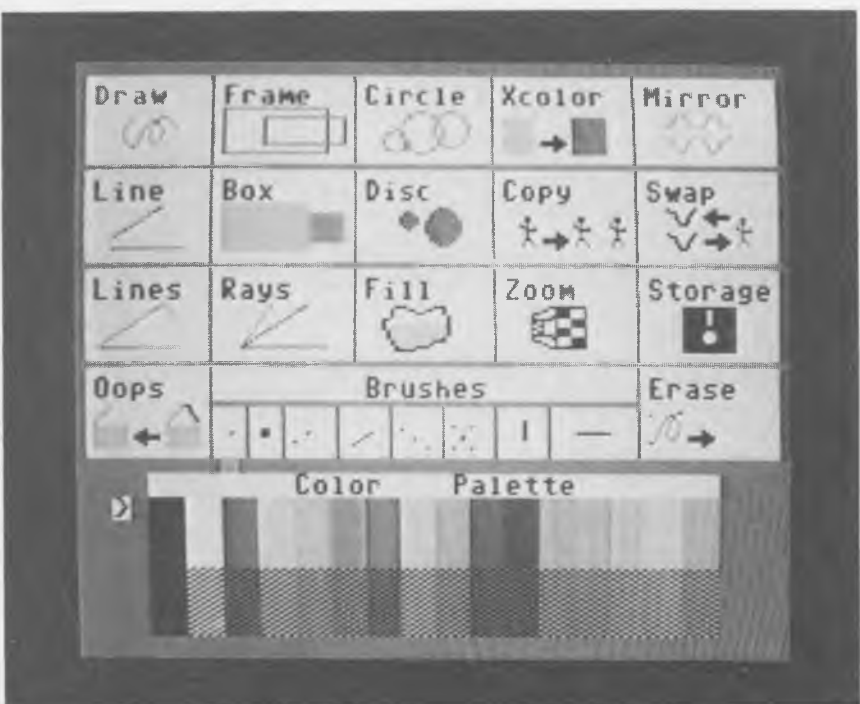
KoalaPainter

Unlike both *Spritemaster* and *Sorcerer's Apprentice, KoalaPainter* is a hardware/software system. The program is included in the purchase price of

the KoalaPad touch tablet. This tablet is approximately the size of a large paperback book with a recessed 4-1/4-inch panel and two large buttons. The tablet is plugged into game port one of your Commodore 64. The recessed surface of the tablet is sensitive to pressure, and pressing this surface causes the KoalaPad to send X and Y coordinates to the computer. Programmers can read the CIA paddle locations (54297 and 54298) to recover and use these coordinates. In the case of *KoalaPainter* software, the authors use the coordinate readings to allow drawing.

Like *Spritemaster*, *KoalaPainter* is menu driven. However, *KoalaPainter*'s menu is visual rather than verbal. All program options are indicated with tiny icons or images representing the function. You select a function by moving your finger around the touch tablet until the cursor—a small arrow—is on top of the icon for the desired function, and pressing one of the buttons on the tablet. A superb array of functions is available (see Fig. 16.4).

Figure 16.4
***KoalaPainter* Menu**



Painting Options

Like *Sorcerer's Apprentice, KoalaPainter* offers freehand drawing, construction with lines, automatic creation of circles or boxes, color-fill of enclosed areas, and a variety of brushes with which to paint. The touch tablet makes it simple to use these drawing functions. Furthermore, the action of drawing on the tablet with your finger or the Koala stylus is closer to drawing on paper than moving the handle of a joystick.

Color selection is also done through the menu, and all of the Commodore 64's 16 colors are available. The usual color restrictions exist. After all, the VIC II is generating the *KoalaPainter* images. The program's authors have found a way to give you a larger palette by mixing pixels of different colors.

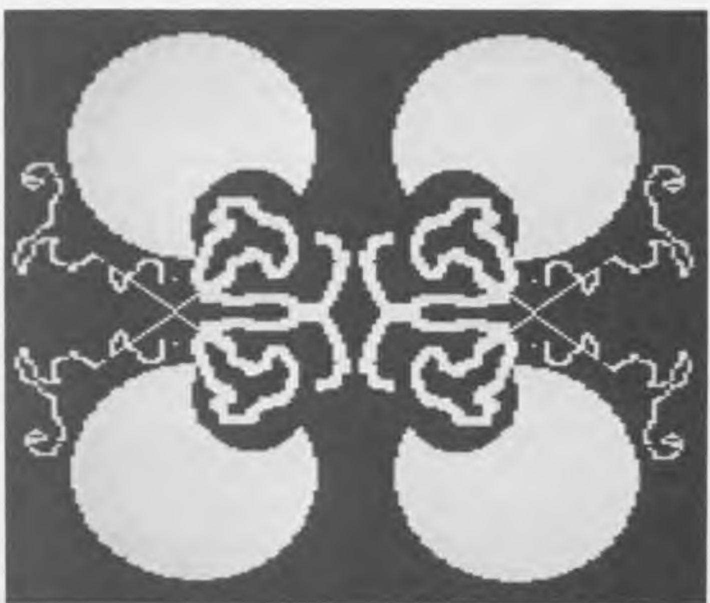
The color menu is divided in half. Selecting a color from the top half of the menu yields a pure color, but selecting one from the bottom yields a mixed color. When painting with these mixed colors, the color resembles a checkerboard—alternating pixels of two colors. However, they do allow for much more subtle shading than the pure colors and add significantly to the program's use. Because all 16 colors can be mixed, the total palette contains 128 "colors."

KoalaPainter also offers a zoom or magnify mode that is superior to the same function in *Sorcerer's Apprentice*. When in zoom mode, the screen is split. The top quarter of the screen shows the drawing in its normal size, the middle of the screen shows the magnified view, and the bottom of the screen provides a color menu (see Fig. 16.5). All parts of the screen are active; to change the area being magnified you move the cursor onto the top part of the screen and scroll the window. Changes made in the magnified view are immediately reflected on the normal size view.

The program also offers an interesting selection of special functions including a copy image option, change color (Xcolor) option, undo last command (oops) option, and mirror option. Mirroring causes all of you drawing to be flipped. Fig. 16.6 shows an image created with the mirror option. These functions are powerful and fun.

Figure 16.5
Zoom Mode in *KoalaPainter*

Figure 16.6
Mirroring in *KoalaPainter*



Multiple Screens

Swap is perhaps most interesting of the special functions. *KoalaPainter* allows you to work on two screens in memory at the same time. With swap, you can move portions of one screen onto the other. To help you learn this function, the distribution disk contains a variety of saved pictures—animals, alphabets, and other images. By loading the jungle animal images onto one screen, you can create your own jungle without having to draw anything. This function is especially handy for young children.

KoalaPainter allows you to save and load your own screens onto your own work disks. You are provided with a catalog of the pictures on the disk; to load a picture you simply point to it on the catalog and press the button on the tablet. The authors have even provided a disk formatting option on the storage menu should you forget to format a data disk. The disk functions are very well thought out. But there is one problem: there is no simple way to load the pictures from your own programs.

The documentation consists of a small, 14-page bookct that describes all program functions. The program and the tablet are so simple to use that most people will not have to consult the booklet much.

KoalaPainter and *Sorcerer's Apprentice* are both useful tools, but the former is tied to optional hardware. While the pad is very useful, it does use the paddle inputs rather than the joystick ports. This makes it more difficult to incorporate the tablet in your programs, and limits its use with other software. However, Koala plans to release additional programs which will take advantage of the tablet's unique user interface.

We are by no means endorsing these products as the only useful programs available. Software tools are very personal items. Aspects of these programs that I like may drive you crazy. However, these are among the most capable tools on the market today. You should survey the market and talk to people who have used a product before you purchase it. A reputable computer store should let you test a product. Remember, if you find yourself in need of a tool, check and see what is already available before building it yourself.

Spritemaster is a registered trademark of Access Software. KoalaPad and *Koala Painter* are registered trademarks of Koala Technologies Corporation.

APPENDIX Programming Primer

What is programming? It's telling a machine how to do a certain job for you—giving it commands that both you and your computer understand. Do you want it to draw a picture? Play a tune? Keep track of your friends' birthdays? Write poems? You can have your computer do all this if you can learn how to give it instructions. It's really simple, once you get the hang of it.

Programming can be extremely rewarding. It's a terrific feeling to watch your computer do what you wanted it to. Most people can learn the basic tricks in an afternoon. Some people, however, have trouble. The difficulty stems from the fact that computers are perfectionists. Even so, a good teacher can show you how to avoid the few rough spots.

Microcomputers cannot yet tolerate the uncertainty we humans are accustomed to. "About a cup of flour" may work in a recipe, but it wouldn't in a computer program. Scientists in the field or artificial intelligence are working on this problem, but for now, we need to be precise when talking to our computers.

For example, the presence or absence of a space may make the difference between a program running right or it running at all. Put one semicolon where a comma should go and your program may go haywire, or not run at all. Many people are frustrated by this; others like the challenge.

How Does It Work?

The heart of any computer is its central processing unit, or CPU. The heart of your Commodore 64 is a 6510 microprocessor. To explain in detail how it works would fill many chapters in this book. Briefly, a microprocessor

takes two pieces of information and manipulates them by following rules stored on the chip. It may, for instance, compare two numbers and, based on the results of this comparison, take one of two actions. The kinds of manipulations and actions the CPU can do vary from one brand of processor to another. (This is one reason why programs for the Radio Shack Color Computer cannot be loaded and run on a Commodore 64—they have very different microprocessors.)

The information in the microprocessor itself is held in currents running through wires. We're able to change the current patterns and therefore change the **data**—the information—that these current patterns represent. The modern computer revolution is the result of the realization by computer scientists that **instructions** could also be represented by these current patterns.

Early computers were hard-wired, that is, they were based on mechanical switches and gears. Programming in those days was a matter of arranging these switches and gears. The design of modern microprocessors enables them to distinguish between the instructions and the data. (A cook, after all, knows the difference between the list of ingredients—the data—and the cooking instructions, even though they are both written on the same recipe card.) Programming is simply arranging these instructions, and the accompanying data, correctly.

Machine Language and Assembler

Since the easiest way to deal with electricity is to see whether the current in a wire is either on or off, the instruction set of modern microprocessors is generally represented in **binary notation.** Binary means that there are only two states: on or off. New research in computer science could lead to the obsolescence of binary computers, but it will be decades before such research translates into inexpensive, home computers. We human don't do well with binary. It's too easy to make a mistake and type `00010000` when you actually mean `00100000`.

To deal with this difficulty, computer scientists have invented a veritable Tower of Babel of computer languages. These languages are actually master programs that allow users to enter instructions into computers without the potential typing mistakes associated with binary coding. They translate the programmer's entries into **machine language**—the set of binary-notation instructions that computers understand. Like most other human endeavors, people's ideas about how to interact with computers vary. Therefore, compu-

ter languages vary. Some languages were developed for very specific programming problems, while others are capable of solving a more general range of problems.

Assembly language (commonly referred to as **assembler**) is closest to the binary machine language used internally by the computer. In assembly language, there is a one-to-one correspondence between the binary, or machine language, instruction and the assembler instruction. The difference is that in assembler, the instruction is entered as a mnemonic—a short code-word—rather than a binary number. After all, it's much easier to remember ADC (which is short for ADd with Carry) than 01101001.

Machine language is not a bed of roses; there are problems in using it. First, since each microprocessor instruction does a very tiny part of the program, assembly language code tends to be long, with more opportunities for error. Second, assembly language is not interactive. That is, you cannot type a command and see the computer do it immediately, as you can with Basic. The instruction must first be translated to machine language, by the assembler. Finally, since there are now five or six popular microprocessors, each with a different machine language, assembler code cannot be easily shared among different brands of computers.

Basic Made Easy

These problems are largely solved by the **high-level languages.** They're called high-level because they no longer have a one-to-one correspondence to machine code, and are miles closer to English. High-level languages that can translate instructions into machine code on-the-fly are called **interpreted languages.**

Basic and Logo are this type of language. **Compiled languages** follow the assembler model and require a separate translation phase after a whole program has been written. Finally, while high-level languages are somewhat standardized, there are important differences, and you have to consider these dialect problems when trying to move a program from one computer to another. Unfortunately, Basic—the most common language—is also the most different from one brand of computer to another.

Commodore Basic is built into the Commodore 64 computer. It's at your service the minute you turn the computer on. Basic is not well-loved by the professional programmer, but it has the advantage of Mr. Everest—it's there. Since Commodore has been making computers from almost the

beginning of the microcomputer revolution, it has released several different versions of Basic.

These are all based upon the original personal computer version, Microsoft Basic. Microsoft wrote the version of Basic that ran on the very first personal computer, the Altair, and a version of Microsoft Basic runs on nearly every microcomputer.

Commodore chose to include the earlier Basic version 2.0 in the Commodore 64 rather than the newer and more advanced Basic version 4.0, which would have eaten more valuable memory. Unfortunately, most of the interesting features of the Commodore 64 are not easily accessible from Basic. But Basic is a full-fledged computer language and given that, magic can be wrought. What's the trick?

For any programming problem, there are multiple solutions. Some are likely to lead to dead-ends—programs that cannot be further modified. Other solutions may be so complicated that recalling how something was done six months later may be impossible. A program is often built on decisions made very early in the programming process that can have very serious consequences later. This is true in both the technical aspects and the design aspects of programming.

The best way to avoid mistakes is to plan your programs carefully. When I first took a computer science course, I had to formally flowchart all my programs. A flowchart is a diagram, using strictly defined symbols, that describes how a program works. Back then, I thought this was required because the keypunch machines that we used were in short supply and the more time students could be kept writing and rewriting flowcharts, the less time they would need to spend on the keypunches or on error-filled runs.

With personal computers, getting access to a machine is no problem. The machine is dedicated to you. Today, with thousands of programs under my belt, I know that time considerations were only part of the rationale for the earlier emphasis on flow charts. The main reason for using flowcharts was that they forced students to plan their programs. Planning is still important: it can save you valuable time. I still don't think much of flowcharts, as they are too close to program code; I prefer a less formal method such as a program plan.

Program Plans

The first step in constructing a program plan is to decide what your program is supposed to do. It's very difficult to program something in

constant flux. You should write down all the major subfunctions that are necessary to achieve your goal. This is the true design phase in which you must begin to consider how the user will interact with the program as well as how the computer will achieve its computational goals. A good start for a plan for a dinner party might be:

```
Goal: Dinner Party
    Subgoal: guests
                        Subgoal: select
                        Subgoal: invite
    Subgoal: plan party
                        Subgoal: menu
                        Subgoal: shop
    Subgoal: prepare
                        Subgoal: clean house
                        Subgoal: cook dinner
                        Subgoal: set table
    Subgoal: give party
```

Next you should consider how to chop the program idea or ideas into programmable pieces. What common subroutines might be shared by separate pieces? In the example above, both the "set table" and "give party" pieces involve the washing of dishes and utensils at my home. Several components also share a telephone call "routine."

Lengthy programs often only have one input subroutine that is used by the other program pieces. While you are considering these subroutines, you must also consider what each will have to do. A common input subroutine, for example, may have to check for errors. Has anything other than a number been entered? Has a number larger than 15 been entered? And so on.

On Data

While you're doing this, you must also consider the data. How will the information be structured so that you have a balance between ease of use and satisfactory computations? For example, programs often require users to

enter numbers rather than letters. In Basic, if someone erroneously types a letter in response to the statement INPUT N, Basic will complain with a ?REDO FROM START message and refuse to budge. So it is common programming practice to accept only strings or literals—which may be in the form of letters or numbers—and convert them to numerics by using the VAL function. For instance, the value of the string "4" is 4. This also allows you to issue a more lucid error message, perhaps: YOU TYPED A LETTER; TRY AGAIN BUT TYPE A NUMBER.

Another aspect of this planning process is deciding whether to use arrays or simple variables to store your data. Arrays are more complex than simple variables, and often intimidate beginning programmers. But arrays are usually the simplest way to hold a set of related numbers or words. I'll be using them rather freely for colors in multicolor mode, for sprite designs, and for tone values.

How do you know when to use an array? When a set of variables is related. In writing a tic-tac-toe program, for example, I'd probably use an array to hold the playing grid—the tic-tac-toe grid even looks like an array. In a checkbook-balancing program, I'd use two arrays to store the monthly debits and credits. The most important thing to realize is that arrays are no harder to use than simple variables; they just require more practice.

Variable Warnings

You must also pay attention to how you name your variables. Most programming languages support so-called "local" variables in subroutines whose values do not affect the rest of the program. Basic, however, has only "global" variables—those that are known throughout the program.

Since we will be using subroutines very heavily, it is important to make some conventions: A$ represents user input, T and T$ are used for temporaries whose values constantly change, FOR/NEXT loops use I, J, K—and so on. I have adopted these conventions over years of programming. Other variables should have more descriptive names, but Basic 2.0 only looks at the first two characters of a variable name: GRID and GRIPE are exactly the same to the Commodore 64! It is important to do careful planning here.

You may have wondered why so many people use I, J, and K in loops. Habit. In FORTRAN, the language many programmers learned first, any variable name that starts with a letter from I to N is predefined as an integer—a whole number. And in FORTRAN, these integers are used to control "loops," repetitive structures that are fundamental program building blocks. Old habits die hard.

Once the basic program flow is clear and the variables have been defined, you are ready to start coding. Obviously, the planning process as described above could take a lot of time. It will be worth it.

Debugging

Debugging is, to my mind, the most interesting part of programming. Bugs or errors in a program can range from simple syntax errors to serious errors in logic. The term **debugging** refers to fixing them all.

No one produces bug-free programs the first time around. Sometimes bugs are subversive and don't appear until very late in the testing process—just when you thought the program was done and out the door. Syntactical bugs, typing errors, or other problems with the structure of a line itself are usually trapped quickly. Happily, Basic reports SYNTAX ERROR IN [line number].

You can fix such errors quickly, because Basic points you to the offending line. When typing in programs from books and magazines, these kinds of bugs are easily created. Poor printing can smudge a colon into a semicolon; typesetters can accidentally repeat or omit a section of code. Careful analysis of the offending line or program section is the only cure. But don't forget to do your research. Study the Commodore manuals or other reference guides.

Some errors may not be so obvious. You can accidentally change a variable's value in a subroutine or use an AND instead of an OR. A FOR/NEXT loop can go one too many (or one too few) times. Or you can confuse the first index with the second in a two-dimensional array. These bugs can cause weeping and gnashing of teeth. There are, however, some smart ways to deal with them.

Debugging Tools

The simplest technique is to use liberal numbers of PRINT statements. Print out, as often as you can, the value of the variable that is causing the problem. Since I often use screens that are heavily formatted, I usually include a bit of screen formatting in these temporary prints. For example: PRINT "<HOME/CLR>"; A will print out the value of A in the upper lefthand corner of the screen. I also number these temporary bug-hunting lines uniquely, ending them with 1s (such as 111 or 121). This lets me

quickly scan the program code and remove all the temporary lines once the program is working properly.

More useful, in complex situations, is the STOP statement. Stop can be inserted anywhere in your program. Basic will stop the execution of the program and report BREAK IN [line number]. At this point, you can print out the value of any variable you want to examine. You can also change the value of any or all of the variables by merely keying them in at this point; for example A = 50 will automatically set the value of A to 50 no matter what else happened to it earlier in the program. When you've accomplished your goals, you can restart the program where it left off by keying CONT. Beginners don't use STOP/CONT enough. Try it, you'll like it.

Unfortunately, Commodore 64 Basic lacks a TRACE command, although some of the Basic enhancements for the Commodore 64 do offer this useful feature. TRACE allows you to see the flow of a program, by printing the number of the line that is currently executing on the screen. You might imagine what an unending loop looks like.

But there is a trick you can use if your version of Basic doesn't contain TRACE. Pepper your program with temporary PRINTs such as 205 PRINT "200": REM INPUT SUBROUTINE. This technique will accomplish exactly the same thing as TRACE and has the benefit of allowing you to control what parts of the program you want to follow. If 200 never appears on the screen, you know that somehow you forgot to branch to 200 or that an IF statement is being improperly evaluated. Very useful.

These are the Commodore 64's debugging tools. While they could be richer, in combination with your knowledge of the program plan they should be adequate. The programs in this book have all been tested and debugged—they work flawlessly. However, you'll probably make typing errors when entering them. Use the techniques described above to help remove the kinks.

Editing on the Commodore 64

The Commodore 64 offers a full screen editor. That means the cursor can be freed from the line it's on to roam around the screen. Further, you can change anything on the screen and have that change be reflected in the program. But there's a rub. You must press <RETURN> on each line to have your changes actually recorded in the program. If you don't press <RETURN>, the changes will be to the screen only. I often do a lot of editing around the screen, and, when all the changes are done, move the

cursor to the head of the top line and press <RETURN> repeatedly. This saves a lot of cursor movement in many cases.

A Basic program can be up to 80 characters in length: two screen lines. This is very constricting if you are used to computers with 255-character lines. There's no easy way around this. Commodore programmers, therefore, often jam as much on a line as possible by omitting unnecessary spaces and by using the special command abbreviations. For the sake of readability, I've avoided using both of these tricks although they can be quite handy. The spaces and nonabbreviated Basic instructions do take up valuable RAM, but with 64K we have memory to spare.

I've experienced one difficulty with the Commodore 64 editor that bears attention. Occasionally, two lines glue themselves together (a flaw of the IBM PC as well). The first time this happened it took me a few perplexing minutes to figure it out. It happens only when a line is exactly 40 characters in length. But let's see it in action. Fire up your computer and type:

NEW press <RETURN>
10PRINT"THIS IS A FORTY CHARACTER LINE." press <RETURN>

Now press the <CURSOR UP> key (that is, press the <SHIFT> and the left <CRSR> key) until the cursor is directly underneath the 1 in 10. Now type:

20 PRINT press <RETURN>
RUN press <RETURN>

Odd behavior . . . . Let's go on. Type:

15 PRINT "HELLO" press <RETURN>
LIST press <RETURN>

Very strange. Line 20 is out of place!

This seemingly bizarre behavior is due to the way Basic tells how a line has ended, in concert with the cursor movement. Basic, unbeknownst to you, puts an end-of-line marker at the end of each program line when you press <RETURN>. When you moved the cursor to underneath the 1 in 10, you destroyed that marker and Basic thought you were still editing line 10. As written, Basic cannot deal with this.

This won't happen often, but it's worth knowing the fix. Retyping lines is not necessary, since we can fix it with the very same screen editor that

caused the problem. Use the cursor keys to move the cursor to the 2 in 20. Now use <INST/DEL> to erase the whole first part of the line. (It will take 40 key presses.)

At this point, all that should be left of the line is 20 PRINT. Press <RETURN>. Now list the program. Line 20 lists as it should. Now move the cursor back to line 10—this time at the very end of the line, one space beyond the T in the second PRINT. Press <INST/DEL> until the cursor is underneath the 1 in 10 again. Press <RETURN>. List the program. Phew!

Merging Programs

Normally, the screen editor does save a lot of time. There's another way that we can save time. The programs in this book (and many others that use standard subroutines or data) share a lot of code. Many Basics have a MERGE command to allow you to combine separate subroutines from disk. With this command you can maintain a library of sprites or music data and save a lot of typing.

First, you must make sure the line numbers are compatible. When merging programs with identical line numbers, lines get lost. You can use the screen editor to manually renumber lines so that they are correct. Just move the cursor to the line number, change it, and press <RETURN>. Then delete the old, duplicate line by typing the line number and hitting <RETURN>. Save this modified program. Repeat this activity until all program pieces are correctly numbered.

Once renumbered, load the program to be merged and list it on the screen. Then load the second program. Move the cursor to the top of the screen, above the first line number, and hit <RETURN> until all lines have been entered. Save the new program. All you've done is use the screen editor as a temporary storage place.

Technically, these techniques will also work for tape, but you may find the time it takes to wait/rewind/search unbearable.

Unfortunately, this technique sounds a lot better on paper than it does in practice. Frequently, programs to be merged are too long to fit on one screen. There are techniques to accomplish a merge, but they are beyond the scope of this book. Instead, you can repeat the process above for "screen-sized" pieces. Don't forget to save the intermediate copies to which you are adding. It takes a while, but beats retyping a long list of data items.

Dealing with Quote Mode

I've also had trouble with the screen editor when editing lines with

graphics or cursor control characters in them. The editor goes into "quote mode" only when an opening quote has been entered or when characters are inserted with the <SHIFT INST/DEL> key. Some people type a new opening quote, edit, and hit <RETURN>, then move the cursor back to the offending line and delete the extra quote. I just make liberal use of the <SHIFT INST/DEL> key to open up new spaces in the line. I then delete the extra characters before pressing <RETURN>. This is probably a matter of style, but my method does save some cursor action.

Getting Started

Computer experts like to debate the merits of learning to program. While I don't think anyone *needs* to program, I do feel that everyone *should* at least try it. If you're like most people, you will often find that commercial software falls short of your expectations. It does almost what you want. But if you know how to program, you can usually make the computer do exactly what you want. Of course, programming is not always the answer. For example, the effort necessary to create your own word processing program would probably far outweigh the cost of commercially available programs.

There's more to programming than this, though. When you learn to program you learn a way of thinking. There are some who say that such thinking will create a society of robots. This is not likely. There are many rules and restrictions in writing a computer program, but within these rules there is much room for creativity. The ability to think clearly and plan logically are skills that are useful far beyond the realm of the computer.

The mystique of the egghead programmer is just that—myth, not substance. Programming doesn't require magic (though the results may be magical) nor does it require mathematical genius. Which is not to say we can't use the techniques and tools that brilliant computer scientists have developed over the last 30 years or so.

There's only one way to know if you'll like programming, and that's to try it. Follow the rules outlined above, and turn to Chapter 1. Read everything carefully—there's a lot to learn, but you'll get the hang of it pretty quickly. If you type the examples in exactly as they appear in the text, you'll begin to see some of the Commodore's magic right away.

Best of all, by the time you're done, you'll be a pretty sophisticated programmer yourself. And you'll have your own copies of the powerful software listed inside this book, which will let you create just about any kind of graphics and sound you want. So what are you waiting for? Open to the first page and let's get started.

APPENDIX

Commodore Character Set

Screen Codes

POKE Code	Set 1	Set 2	POKE Code	Set 1	Set 2	POKE Code	Set 1	Set 2
0	@		21	U	u	42	*	
1	A	a	22	V	v	43	+	
2	B	b	23	W	w	44	,	
3	C	c	24	X	x	45	-	
4	D	d	25	Y	y	46	.	
5	E	e	26	Z	z	47	/	
6	F	f	27	[		48	0	
7	G	g	28	£		49	1	
8	H	h	29	]		50	2	
9	I	i	30	↑		51	3	
10	J	j	31	←		52	4	
11	K	k	32			53	5	
12	L	l	33	!		54	6	
13	M	m	34	"		55	7	
14	N	n	35	#		56	8	
15	O	o	36	$		57	9	
16	P	p	37	%		58	:	
17	Q	q	38	&		59	;	
18	R	r	39	'		60	<	
19	S	s	40	(		61	=	
20	T	t	41	)		62	>	

Screen Codes

POKE Code	Set 1	Set 2
63		?
64		
65		A
66		B
67		C
68		D
69		E
7Ø		F
71		G
72		H
73		I
74		J
75		K
76		L
77		M
78		N
79		O
8Ø		P
81		Q
82		R
83		S
84		T
85		U
86		V
87		W
88		X
89		Y
9Ø		Z
91		
92		
93		
94		
95		
96		
97		
98		
99		
1ØØ		
1Ø1		
1Ø2		
1Ø3		
1Ø4		
1Ø5		
1Ø6		
1Ø7		
1Ø8		
1Ø9		
11Ø		
111		
112		
113		
114		
115		
116		
117		
118		
119		
12Ø		
121		
122		
123		
124		
125		

Screen Codes

POKE Code	Set 1	Set 2	POKE Code	Set 1	Set 2	POKE Code	Set 1	Set 2
126			148	T	t	170	*	
127			149	U	u	171	+	
128	@		150	V	v	172	,	
129	A	a	151	W	w	173	-	
130	B	b	152	X	x	174	.	
131	C	c	153	Y	y	175	/	
132	D	d	154	Z	z	176	0	
133	E	e	155	[		177	1	
134	F	f	156	£		178	2	
135	G	g	157	]		179	3	
136	H	h	158	↑		180	4	
137	I	i	159	←		181	5	
138	J	j	160			182	6	
139	K	k	161	!		183	7	
140	L	l	162	"		184	8	
141	M	m	163	#		185	9	
142	N	n	164	$		186	:	
143	O	o	165	%		187	;	
144	P	p	166	&		188	<	
145	Q	q	167	'		189	=	
146	R	r	168	(		190	>	
147	S	s	169	)		191	?	

Screen Codes

POKE Code	Set 1	Set 2	POKE Code	Set 1	Set 2	POKE Code	Set 1	Set 2
192			214		V	236		
193		A	215		W	237		
194		B	216		X	238		
195		C	217		Y	239		
196		D	218		Z	24Ø		
197		E	219			241		
198		F	22Ø			242		
199		G	221			243		
2ØØ		H	222			244		
2Ø1		I	223			245		
2Ø2		J	224			246		
2Ø3		K	225			247		
2Ø4		L	226			248		
2Ø5		M	227			249		
2Ø6		N	228			25Ø		
2Ø7		O	229			251		
2Ø8		P	23Ø			252		
2Ø9		Q	231			253		
21Ø		R	232			254		
211		S	233			255		
212		T	234					
213		U	235					

ASCII Codes

CHR$ Code	Action or Character	CHR$ Code	Action or Character	CHR$ Code	Action or Character
Ø	n/a	3Ø	Green Chars.	62	>
1	n/a	31	Blue Chars.	63	?
2	n/a	32	Space	64	@
3	n/a	33	!	65	A
4	n/a	34	"	66	B
5	White Chars.	35	#	67	C
6	n/a	36	$	68	D
7	n/a	37	%	69	E
8	Disable Shift Commodore	38	&	7Ø	F
		39	'	71	G
9	Enable Shift Commodore	4Ø	(	72	H
		41	)	73	I
1Ø	n/a	42	*	74	J
11	n/a	43	+	75	K
12	n/a	44	,	76	L
13	Return	45	–	77	M
14	Lower Case	46	.	78	N
15	n/a	47	/	79	O
16	n/a	48	Ø	8Ø	P
17	Cursor Down	49	1	81	Q
18	Inverse Video	5Ø	2	82	R
19	Home	51	3	83	S
2Ø	Delete	52	4	84	T
21	n/a	53	5	85	U
22	n/a	54	6	86	V
23	n/a	55	7	87	W
24	n/a	56	8	88	X
25	n/a	57	9	89	Y
26	n/a	58	:	9Ø	Z
27	n/a	59	;	91	[
28	Red Chars.	6Ø	<	92	£
29	Cursor Right	61	=	93	]

ASCII Codes

CHR$ Code	Action or Character	CHR$ Code	Action or Character	CHR$ Code	Action or Character
94		126		158	Yellow Chars.
95		127		159	Cyan Chars.
96		128	n/a	160	Space
97		129	Orange Chars.	161	
98		130	n/a	162	
99		131	n/a	163	
100		132	n/a	164	
101		133	**f1**	165	
102		134	**f3**	166	
103		135	**f5**	167	
104		136	**f7**	168	
105		137	**f2**	169	
106		138	**f4**	170	
107		139	**f6**	171	
108		140	**f8**	172	
109		141	Shifted Return	173	
110		142	Upper Case	174	
111		143	n/a	175	
112		144	Black Chars.	176	
113		145	Cursor Up	177	
114		146	Inverse Video Off	178	
115		147	Clear Screen	179	
116		148	Insert	180	
117		149	Brown Chars.	181	
118		150	Light Red Chars.	182	
119		151	Gray 1 Chars.	183	
120		152	Gray 2 Chars.	184	
121		153	Light Green Chars.	185	
122		154	Light Blue Chars.	186	
123		155	Gray 3 Chars.	187	
124		156	Purple Chars.	188	
125		157	Cursor Left	189	

ASCII Codes

CHR$ Code	Action or Character	CHR$ Code	Action or Character	CHR$ Code	Action or Character
190		212		234	
191		213		235	
192		214		236	
193		215		237	
194		216		238	
195		217		239	
196		218		240	
197		219		241	
198		220		242	
199		221		243	
200		222		244	
201		223		245	
202		224		246	
203		225		247	
204		226		248	
205		227		249	
206		228		250	
207		229		251	
208		230		252	
209		231		253	
210		232		254	
211		233		255	

APPENDIX
Selected References

GRAPHICS

Foley, J. D. and A. Van Dam. 1982. *Fundamentals of Interactive Computer Graphics.* Reading, Mass.: Addison-Wesley.

Harrington, S. 1983. *Computer Graphics: A Programming Approach.* New York: McGraw-Hill.

Hearn, D. and M. P. Baker. 1983. *Microcomputer Graphics.* Englewood Cliffs, N.J.: Prentice-Hall.

Newmann, W. M. and R. F. Sproull. 1979. *Principles of Interactive Computer Graphics.* New York: McGraw-Hill.

6502 (6510) ASSEMBLY LANGUAGE PROGRAMMING

Leventhal, L. A. 1979. *6502 Assembly Language Programming.* Berkeley, Calif.: Osborne/McGraw-Hill.

Leventhal, L. A. and W. Savill. 1982. *6502 Assembly Language Subroutines.* Berkeley, Calif.: Osborne/McGraw-Hill.

MAGAZINES

Commander. Published monthly by Micro Systems Specialties, 3418 S. 90 St., Tacoma, WA 98409.

Commodore: The Microcomputer Magazine. Published bimonthly by Commodore Business Machines, 1200 Wilson Dr., West Chester, PA 19380.

Compute! and *Compute!'s Gazette.* Published monthly by Computer! Publications, PO Box 5406, Greensboro, NC 27403.

Micro. Published monthly by Micro, Amherst, NH 03031.

Run. Published monthly by Wayne Green, Inc., 80 Pine St., Petersborough, NH 03458.

Index

algorithm, 28; to copy characters, 53; to decode byte to bits, 61-64; to display sprites, 75; to determine-hi-res X, Y locations, 108-10; to read joystick, 127-28; memory, 70; to decode music, 158; to determine odd or even, 43-44, 57; to generate random numbers, 30; character bank selector, 48; to center text, 27-28; to position text, 27-28; to generate tone frequency, 152
aliasing, 114
AND, 12, 16-17, 48-49, 92, 98, 110, 132
animation, 40-46, 71-83; cycle, 41; multiframe, 43, 81-82; raster, 40; vector, 40
argument, 22
artifacting, 55, 112
aspect ratio, 72, 113-14
assembly language, 138-50, 213; branching, 143-44; labels, 142-43
attack. *See* SID envelope

binary, 12-13, 16-17
bit changing, 16-17; relation to pixel, 96
bit map, 71, 105, 107, 109
Boolean functions, 16, 20-21, 44, 45. *See also* AND, OR, and NOT
byte, 8, 12, 16-17, 26-27, 63, 72-73, 91

cassette buffer, 73
character generator, 10, 11, 47, 51-53
CHR$ function, 25-27
CIA (Complex Interface Adaptor), 10, 11, 12, 126-28

color, 32, 33; background, 32, 52; foreground, 32; hi-res, 107; masks, 132; memory, 35-36; multicolor hi-res, 117
Commodore key, 6, 23, 34
computer languages, 213; compiled, 213; high-level, 213; interpreted, 213. *See also* assembly language, Simon's Basic, Pilot, and Logo
cursor control characters, 22-25

debugging, 217-18
decay. *See* SID envelope

editor, 218-19

flags, 19-20
function library, 29; user-defined, 28-29

hi-res (high resolution) graphics, 105-37

joystick, 126-28

kernal, 9-11, 53, 140-42, 144
Koalapainter, 206-10

logo, 183-91; hi-res, 185-89; music, 191; procedures, 187-89; sprites, 190

machine language, 212
masks, 97-100, 131-32
memory map, 9-11
merging, of programs, 219-20
multicolor graphics, 96
music, single-voice, 157-59; multi-voice, 168-70

nibble, 35, 48, 107
NOT, 21, 122

ON/GOTO, 30
op code, 111, 141
OR, 12, 16-17, 48-49, 92, 110

PEEK, 7-8, 12, 35
Pilot, 192-95; hi-res, 194-95; music, 195; sprites, 195
pixel, 72, 96, 106
pointers, 18-19
POKE, 7-8, 12, 32-33, 35, 46, 48, 49, 144-45, 148, 154
programming, 211; data for, 215-16; plans, 214-15

quote mode, 26, 220

READ/DATA, 18, 20
registers: 6510 microprocessor, 190-91; extended background color, 103-04; sprite multicolor, 101; text multicolor, 97; SID/sound, 153
release. *See* SID envelope
resolution, 96, 106
ROM (read only memory), 9-11, 35, 51-52

screen memory, 10-11, 35-36, 48, 74
SID (Sound Interface Device), 10-11, 151-73; chromatic scale, 155; envelope, 159-60; registers, 153; waveform, 160-61
Simon's Basic, 174-83; hi-res with, 176-79; music with, 182-83; sprites with, 180-82
software, evaluation of, 198
Sorcerer's Apprentice, 198-204
sound. *See* SID
SPC function, 22, 41, 43
special interest group (SIG), 197
sprites, 71-95; color, 75-76; control registers, 75; design of, 72; display, 75; expansion of, 78-80; pointers, 74; positioning of, 75-77; priorities, 80; video plane sizes, 80
Spritemaster 64, 204-06
sustain. *See* SID envelope
SYS, 138, 147

TAB function, 22
turtle graphics, 185-86

VIC II, 10-11; banks, 51-52, 73-74, 106, 111-12

waveform, 160-61
wrapping, 31

zero page, 10, 11

INDEX TO PROGRAMS

Program	Page	Program	Page
Basic Loader	148-49	Quickclear	111
Binary/Decimal Converter	14-15	Random Ball One	29
BLOAD	146-47	Random Ball Three	38
Box	110-11	Random Ball Two	36
BSAVE	145-46	Screen Maker	135-37
Character Editor	67-69	Sine Wave	111
Circle	113	Sine/Cosine	119
Color Bars	33-34	Sound Editor	171-73
Ear Debugging	156-57	Sprite Editor	94-95
Flashing Text	27	Text Animate One	41-42
Flying Trapeze/One Voice	158	Text Animate Three	45
Flying Trapeze/Two Voice	168-70	Text Animate Two	43
Marque	23	Three Boxes	117-18
Notes Showoff	154		

MORE MAGIC!

The eye-popping programs and super software tools in this book are guaranteed to delight and astonish you. There's only one thing they can't do—type themselves in. If you'd like your computer to do *all* the typing for you, just send $19.95 plus $2 shipping and handling (and applicable sales tax) to the address below. A Commodore-64 disk containing ready-to-run versions of this book's dazzling demonstrations and software tools will magically appear in your mailbox.

Send your name and address (please print!), and a check payable to Hard/Soft Press to:

More Magic
Hard/Soft Press
Post Office Box 1277
Riverdale, NY 10471

The hand is quicker than the eye, but the Post Office is not. Please allow 2 to 8 weeks for delivery.

$12.95 01258-370 A DUTTON PAPERBACK ORIGINAL ISBN: 0-525-48120-6

What's so magical about COMMODORE MAGIC?

Valuable software! Now, for the first time a book on the Commodore 64 gives you a package of powerful software editors you can use to create your own dazzling animations and startling sound effects. These reusable, professional-quality editors let you completely control your Commodore 64 so your imagination can run wild. It's never been this easy to generate astonishing rainbow-bright pictures and sounds you just won't believe!

Here's what you get:

- A character editor that lets you invent or customize your own professional-looking alphabets
- A sprite editor that turns your Commodore into a magical animation machine
- A high-resolution graphics editor that can paint astonishing computer art masterpieces
- A music and sound editor that rivals expensive sound synthesizers
- A programmer's tool-kit crammed with ingenious tricks and routines to double or triple your computing ability
- And dozens of other magical examples, demonstrations, and eye-popping tricks

Cover design by Nancy Etheredge