COOKBOOK OF CREATIVE PROGRAMS

FOR THE COMMODORE® 64™

PROJECTS FOR MUSIC, ANIMATION AND TELECOMMUNICATIONS

SHOWS YOU HOW TO BUILD UP PROGRAMS IN SECTIONS AND PROVIDES TIPS PROFESSIONAL PROGRAMMERS USE

EXPLORES THE COLOR AND SOUND CAPABILITIES OF THE COMMODORE 64 BY SHOWING YOU HOW TO CREATE LIGHT SHOWS AND MUSIC

BEGINS WITH SIMPLE PROGRAMS TO BUILD YOUR EXPERIENCE AND THEN MOVES ON TO ADVANCED TOPICS TO SHARPEN YOUR SKILLS

FOR NOVICES AS WELL AS EXPERIENCED PROGRAMMERS

NO KNOWLEDGE OF BASIC REQUIRED

INCLUDES 9 READY-TO-RUN PROGRAMS

ROBERT RINDER

COOKBOOK OF CREATIVE PROGRAMS FOR THE COMMODORE®64™

ROBERT RINDER

0-452-25571-6
COMPUTERS

WHAT'S COOKING WITH YOUR COMMODORE 64?

If you have a Commodore 64, you can be sure that it's just waiting for you to put it to good use—whatever you want to do.

You might want to start a computer art program to fit your own tastes. Or teach your computer to play rhythm. Or have drawings rotate on the screen. Or produce a dazzling light show. Or combine colors in wild and wonderful harmonies. Or hook up to bulletin boards and data bases thousands of miles away.

Now you can put your Commodore 64 to the work it was built to do—with the book that shows you how to do it—

COOKBOOK OF CREATIVE PROGRAMS FOR THE COMMODORE® 64™: PROJECTS FOR MUSIC, ANIMATION AND TELECOMMUNICATIONS

ROBERT RINDER is a computer consultant and the author of many books and articles on computers and communications. He has worked on the design and application of very large computer systems in industry and government, and since the advent of microprocessors, he has devoted attention to the use of them at work and in the home. When not consulting or writing, Mr. Rinder relaxes by working on one of his three computers.

COOKBOOK OF CREATIVE PROGRAMS FOR THE COMMODORE® 64

PROJECTS FOR MUSIC, ANIMATION AND TELECOMMUNICATIONS

ROBERT RINDER

A PLUME BOOK
NEW AMERICAN LIBRARY
NEW YORK AND SCARBOROUGH, ONTARIO

NAL BOOKS ARE AVAILABLE AT QUANTITY DISCOUNTS WHEN USED TO PROMOTE PRODUCTS OR SERVICES. FOR INFORMATION PLEASE WRITE TO PREMIUM MARKETING DIVISION, NEW AMERICAN LIBRARY, 1633 BROADWAY, NEW YORK, NEW YORK 10019.

Copyright © 1984 by Robert Rinder

All rights reserved

Photographs of the Commodore® 64™ appear in this book courtesy of Commodore Electronics Limited.

Several trademarks, trade names, and/or service marks appear in this book. The companies listed below are the owners of the trademarks, trade names, and/or service marks following their name.

CompuServe, Inc.: CompuServe Information Service; Dow Jones & Company, Inc.: Dow Jones News/Retrieval Service; Source Tele Computing, Inc.: THE SOURCE.

SIGNET, SIGNET CLASSIC, MENTOR, PLUME, MERIDIAN and NAL BOOKS are published *in the United States* by New American Library, 1633 Broadway, New York, New York 10019, *in Canada* by The New American Library of Canada Limited, 81 Mack Avenue, Scarborough, Ontario M1L 1M8.

Library of Congress Cataloging in Publication Data

Rinder, Robert M.
Cookbook of creative programs for the Commodore 64.
1. Commodore 64 (Computer)—Programming. 2. Computer programs. 3. Basic (Computer program language)
I. Title.
QA76.8C64R55 1984 001.64'2 84-16493
ISBN 0-452-25571-6

First Printing, October, 1984

1 2 3 4 5 6 7 8 9

PRINTED IN THE UNITED STATES OF AMERICA

CONTENTS

Introduction xi

1 **HOW TO USE THIS BOOK** 1

TIPS FOR BEGINNERS 1

Entering Programs 2

Changing Programs 4

Adding to Programs 5

Changing Line Numbers 7

Debugging Tools 7

Deleting Lines 8

Non-English Expressions 9

PEEKing and POKEing in Memory 10

Disks and Tapes 12

How to Use Disks 12

How to Use Tapes 16

How to Protect Your Files 18

How to Find Errors and Correct Problems 19

TIPS FOR EXPERIENCED USERS 19

PROGRAM TEMPLATES 20

CUT-AND-PASTE PROGRAMMING 21

Pasting Program Lines 21

2 **A NEW WORLD OF COLOR** 25

POSITIONING BY ROWS AND COLUMNS 29

HOW TO COMBINE PROGRAMS 30

PATTERNS OF LIGHT 32

COLOR CONTROL 34

KEYBOARD ENTRY 35

SPIRALS AND SYMMETRY 38

3 **LIGHT SHOW** 41
GRAPHIC CHARACTERS 41

TEMPLATES 43

TRIANGLES 44

DIAMONDS 45

TITLE SCREEN 46

AUTOMATING LIGHT SHOW 47

4 **A NEW WORLD OF SOUND AND MUSIC** 51
WHAT IS SOUND? 53

Frequency 53

Overtones 55

Amplitude Envelope 55

THE TEMPERED SCALE 56

HOW YOUR COMPUTER MAKES SOUNDS 57

Sound Memory 57

The Tempered Scale Program 58

5 **MUSICAL KEYBOARD** 62
MUSICAL KEYS 63

MUSICAL KEYBOARD PROGRAM 65

More Notes 67

SAVEing Your Work 68

More Octaves 68

Shaping the Sound 68

Turning Notes Off 70

Setting the Timbres of Sound 71

Tone Control 74

AUTOMATIC PLAYBACK BY THE COMPUTER 78

Controlling Playback 79

How Music and Sounds are Saved 80
How Music and Sounds are Recreated 81
SAVEING MUSIC ON TAPE OR DISK 82
Writing Sequential Files 82
Reading Sequential Files 83
THE LAST NOTE 84

6 ELECTRONIC DRAWING BOARD 87
BIT MAPPED GRAPHICS 87
KEYBOARD CONTROL OF THE DRAWING BOARD 88
THE DRAWING BOARD PROGRAM 90
Initialize Graphics 90
Locate a Bit 92
Aspect Ratio 93
Keeping the Dots on the Screen 93
Moving, Writing, and Erasing Dots 93
POKEing and PEEKing in Graphics Memory 95
Displaying the Cursor 96
The Main Line 97
DRAWING WITH THE KEYBOARD 98
Drawing Straight Lines 100
Moving and Erasing 101
Changing Key Assignments 101
Cursor Steering 102
COLOR FOR THE DRAWING BOARD 103
SETTING THE LENGTH OF CURSOR MOVES 105
ANGLES AND ARCS 105
DRAWING CIRCLES 108

7 THE WORLD OF SPRITES 111
SPRITES THE EASY WAY 112
SPRITE ART 113
Display Eight Sprites 113

SPRITE GRAPHICS 115

Setting Sprite Colors 119

Selecting Sprites 119

How to Move Sprites 120

Patterns for Sprites 123

Expanding Sprites 124

Turning Sprites On and Off 125

8 SPRITES FROM YOUR KEYBOARD 127

SPRITE STUDIO 127

Drawing Sprites 133

How the Program Draws Sprites 134

Expansion and Contraction of the Sprites 136

SEMIAUTOMATIC DRAWING 137

Reversing Colors 137

Drawing Symmetric Sprites 137

DISK AND TAPE OPERATIONS 139

Menu for Disk Operations 139

Saving Sprites on Disk or Tape 141

Saving Eight Sprites at a Time 142

Retrieving Sprite Files from Disk or Tape 143

Recovery from Error 144

Tips on SAVEing and LOADing Files 144

9 SPRITE THEATER 149

BUILDING THE THEATER 149

Cutting and Pasting the LOAD Routine 150

THEATER SOUND 152

THEATER SCENERY 155

SWITCHING MEMORY BANKS 157

10 A NEW WORLD OF COMMUNICATION 163

HOW COMPUTERS COMMUNICATE 164

Signal Rate 167

Call Mode 167
Transmission Mode 168
Parity 168
Word Length 169
Stop Bits 169
End of Line 169
Character Code 169
Color Change 170
End of Line Break 170

INFORMATION SERVICES 170
Communication 171
Electronic Mail 171
Electronic Bulletin Boards 171
Electronic Conversations 171
Finance and Business 171
Travel Services 171
News Reports 172
Electronic Shopping 172
Games 172

APPENDIX A **TURNING BITS ON AND OFF** 174

APPENDIX B **PROGRAMS** 177
SPIRAL 177
KEY ENTRY 179
LIGHT SHOW 180
MUSICAL KEYS 183
MUSICAL KEYBOARD 184
DRAWING BOARD 189
SPRITE ART 192
SPRITE STUDIO 196
SPRITE THEATER 201

LIST OF TABLES, QUICK CHARTS, AND FIGURES

TABLE 1 KEY ASSIGNMENTS FOR LIGHT SHOW 49
TABLE 2 ATTACK, DECAY, AND RELEASE TIMES 61
TABLE 3 TIME OF SOUND OR NOTE, TN 61
TABLE 4 KEY ASSIGNMENTS FOR THE MUSICAL KEYBOARD 85
TABLE 5 KEY ASSIGNMENTS FOR THE ELECTRONIC DRAWING BOARD 110
TABLE 6 KEY ASSIGNMENTS FOR SPRITE ART 125
TABLE 7 KEY ASSIGNMENTS FOR SPRITE STUDIO 148
TABLE 8 KEY ASSIGNMENTS FOR SPRITE THEATER 162
TABLE 9 INFORMATION SERVICES 173

QUICK CHART 1 GETTING PROGRAMS TO WORK 22
QUICK CHART 2 SOLVING DISK OR TAPE PROBLEMS 23

FIGURE 1 DISPLAY OF CHARACTERS AND COLOR ON THE SCREEN 27
FIGURE 2 COMPONENTS OF SOUND 54
FIGURE 3 TIMBRE AND TONE CONTROL 76
FIGURE 4 BIT MAPPED GRAPHICS 89
FIGURE 5 LINES DRAWN WITH DOTS ON A TV SCREEN 94
FIGURE 6 SPRITE GRAPHICS 116
FIGURE 7 SPRITE POSITIONING 118

Introduction

This book describes how to use the Commodore 64. You are shown the easy way to get the most from this powerful home computer. A number of fascinating and useful programs are included that will provide many enjoyable hours of entertainment and recreation.

The Commodore 64 has numerous features for creating drawings, sounds, music, sprites, color, and motion. However, these features are not easy to use by themselves, and are often difficult to understand. They require the entry of long lists of numbers, a tedious and inflexible method of using your computer. In this book we will cut through these difficulties by presenting programs that use the keyboard to draw, make music, and control sprites. You concentrate on the creative side of computing instead of getting bogged down in minute details.

The programs are built up in small sections that you can use and test as the sections are added. This makes it much easier for the beginner to find the inevitable typos and other errors that go with the keying-in of programs. Many of these sections can be used as templates for developing your own programs. With the detailed explanations and diagrams that are included, you can learn how the Commodore 64 works and how to use it.

A chapter on telecommunications introduces you to one of the most popular uses of home computers—the computer as communication terminal.

If you are interested only in copying and using the programs presented in this book, you can safely skip the technical details. However, if you want to know how the Commodore 64 works, you are given hands-on demonstrations and line-by-line explanations so you can use the same methods in the programs that you write.

Chapter 1 gives preliminary information on programming and equipment. A section for beginners explains how to use disks and tapes and how to enter programs. Two Quick Charts are included to help you quickly find errors and solve problems. The use of program templates is explained and cut-and-paste programming methods are described.

Chapter 2 presents some simple programs that illustrate how to display colors and characters on your screen. This chapter will allow beginners to practice the techniques described in Chapter 1, gain confidence and, at the same time, create some interesting screen displays.

Chapter 3 presents LIGHT SHOW, a program of dazzling light, color, and graphic patterns. You can control LIGHT SHOW from your keyboard, or you can put it in automatic mode and just let it run.

Chapters 4 and 5 explain the use of the Commodore's very extensive sound capabilities. These are put to use in MUSICAL KEYBOARD, a program that turns your keyboard into a musical intrument and sound synthesizer. MUSICAL KEYBOARD can automatically play back what you enter. When you are satisfied with your composition, save it on tape or disk for later recall.

Chapter 6 presents DRAWING BOARD, a program that demonstrates the use of the Commodore 64's bit mapped graphics. DRAWING BOARD turns your computer into an electronic drawing board with "cursor steering," one of the most enjoyable and challenging ways of drawing with a computer. As with LIGHT SHOW, DRAWING BOARD can run automatically, drawing an endless number of fascinating patterns.

Chapers 7, 8, and 9 explain sprites, those wonderful computer objects that inhabit the world of arcade games and other graphics-type applications. You are shown how to use sprites without copying long lists of numbers into your programs.

Chapter 7 uses the program, SPRITE ART, to introduce you to the ways of sprite graphics. In SPRITE ART, up to eight sprites can be displayed and controlled from your keyboard.

Chapter 8 presents SPRITE STUDIO, a program that provides for drawing and displaying up to eight sprites at a time. And up to eight sprites at a time can be saved on tape or disk for use in other programs. There are four semiautomatic drawing modes for making it easy to draw symmetrical sprites, or for switching foreground and background colors of a sprite. SPRITE STUDIO has extensive color control, enabling you to turn the studio into a gallery for exhibiting your sprite creations.

Chapter 9 contains a program, SPRITE THEATER, that can be used for sprite shows, or as a workshop for creating sprite scenarios that can be incorporated into games and other programs. In SPRITE THEATER you will learn how to detect sprite collisions and how to produce sound effects so you can hear sprites as they crash or gently brush past one another. All colors, sound, and sprite activity are controlled from the keyboard.

In a more sophisticated version of SPRITE THEATER, THEATER HI, you will learn how to switch memory banks in your Commodore 64 to get more working space in memory. This will enable you to use the bit mapped graphics of DRAWING BOARD to create background scenery for your sprite theater. Now your sprites can fly through cities, enter caves, land on Jupiter, or swim in the ocean.

Chapter 10 switches orientation from the inner world of the Commodore 64 to the outer world of communications. You learn the basics of telecommunications and how to connect your Commodore 64 to the telephone line. A description is given of the many services and activities available over the telephone network.

Templates and cut-and-paste programming techniques are explained and used extensively throughout the book to make the keying-in of programs easier, faster, and less prone to error.

COOKBOOK OF CREATIVE PROGRAMS FOR THE COMMODORE® 64™

1

How To Use This Book

The Commodore 64 is one of the most useful home computers on the market. This is because the Commodore has built-in capability for producing sounds, graphics, and sprites. Commodore has also made it very easy to tap into the telephone network and use various information services. In this book we will explore all of these areas.

Before you begin this book, we suggest that you read Chapters 1, 2, and 3 of the *Commodore 64 User's Guide*; Chapters 4–8 of the *Guide*, are also recommended. If you have a disk unit, read pages 1–15 of the disk unit *User's Manual*; if you have a tape unit, read the operating instructions that come with it.

In this chapter we begin with tips for beginners and then give tips for experienced users. Finally, we discuss program templates and cut-and-paste programming techniques. Both of these programming tools will make it easy and pleasurable for you to create programs.

TIPS FOR BEGINNERS

First, we will review certain aspects of the BASIC programming language; then, we will describe how to use the disk and tape units to store programs and files. If the material is familiar to you, please be patient and don't skip it because we will be stressing points that are important in using this book.

Photo 1 The Commodore 64 computer system. Shown are the keyboard-computer combination, the tape unit, and the printer. *Courtesy of Commodore Electronics Limited*

Entering Programs

You do not have to be a programmer to enter and use the programs in this book. All you have to do is learn certain elementary operations and procedures. BASIC, the programming language we will be using, consists mainly of English-like statements that can be read and understood by nonprogrammers. For example, the following is a typical section of a BASIC program:

```
10 REM ** DEMO OF ENGLISH-LIKE BASIC **
20 A=25:B=3
30 PRINT "THE SUM OF A AND B IS" A+B
```

You can probably understand this program without much explanation. There are, however, a few things to notice. First, each line is numbered. The numbers are incremented by 10 but they could have been incremented by 1, 5, or any other convenient number. The reason for spacing lines by 10 is to leave room for inserting new lines later, something that is frequently done when working with computer programs.

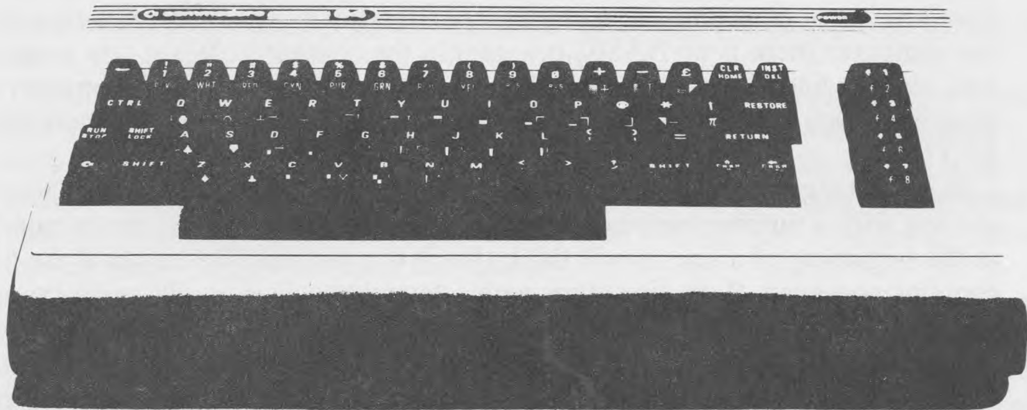


Photo 2 The Commodore 64 keyboard and computer are conveniently housed in this single unit. *Courtesy of Commodore Electronics Limited*

Line 1Ø contains a title. It begins with REM, which is an abbreviation for the word "remark." Whenever BASIC sees REM, it ignores everything on the line following the REM. You can write anything you want after REM—for example, titles, comments, and notes. If you want to put in blank lines to make your program more readable, REM is followed by all blanks.

Line 2Ø is also easy to interpret. The letter "A" is set equal to 25, and "B" is set equal to 3. The *colon* is used as a separator in BASIC, similarly to the way in which a period is used in English. However, instead of sentences, BASIC has *statements*. Each statement in BASIC says one thing that is to be done. As you can see in line 2Ø more than one statement can be placed on a line as long as the statements are separated by a colon. All the other lines in this program have just one statement. REM is a statement in BASIC.

In line 3Ø PRINT does not mean print on a printer; it means print on your computer screen, that is, display the results on the screen. The *phrase in quotes* will be displayed exactly as it is written. Whenever you want to display something on the screen, all you do is enter PRINT followed by what you want to say in quotes. After the quoted phrase in line 3Ø you see A + B. Since this is not in quotes, the statement is telling the computer to do something. The value of A + B will therefore be PRINTed.

To see how this works, enter this works, enter the above program lines into your computer by keying them in just as you see them. After each line is complete you must press the key labeled RETURN. The RETURN key brings the cursor to the left side of the screen, one line down, just like the carriage return on a typewriter. But it does more. When you press RETURN the line you have keyed in is stored as part of your program. In other words, when you turn on the computer there is no BASIC program in the computer. When you enter a line, such as line 1Ø and press RETURN, that line is stored in the computer as a program. Each additional line entered is added to the program when you press RETURN.

Is everything that is keyed in entered as part of a program? No. Only lines starting with a number become part of a program. When BASIC sees a number at the beginning of a line, it will think that it is a program line, even if the line contains nonsense. If the line starts with a letter, however, it will never be made part of a program.

You can run the program by keying in RUN then pressing RETURN. From now on, whenever we say "key in" or "enter," it is understood that after keying in entries, you must press RETURN to make things happen.

By RUNning the program you can see the difference between placing and not placing quotes around an item in the PRINT statement. Items in quotes are called *strings*. They are PRINTed as written. Items not in quotes are *numeric values*, such as A+B.

What happens to the program after it is RUN? You can find out by entering LIST. The program is LISTed on the screen as you wrote it. You can RUN a program as many times as you like without destroying it. In fact, the only way to get rid of a program is to enter the command, NEW, or to turn off the computer. Don't enter NEW unless you want to write a new program.

Changing Programs

Since your program is still in memory, you can change it or add to it. First let's change it. Move the cursor to line 3Ø by pressing the SHIFT key and the up/down arrow key labeled CRSR. Now move the cursor to the right by pressing the left/right arrow key, also labeled CRSR. Move the cursor until it is on the first quote; then press the insert/delete key labeled INST/DEL. You will delete the character to the left of the cursor, which, in this case, is a space. The result is that everything moves left one space toward PRINT. Repeat this process with the cursor on the A after the second quote. You have now removed two spaces.

Press RETURN to enter the change you have made. Let's see how the program works without the spaces. Enter RUN.

If you enter RUN on a line that has other information, you will get an *error message*. BASIC gets confused. The same applies to LIST or any other BASIC command. You can either move the cursor down the screen to a line that is clear or erase everything on the line after the RUN or LIST command.

After you RUN the program, you will see that the results are identical to those obtained previously. Spaces make no difference to BASIC. Insert them or delete them, the program will work the same.

Here is another change. Move the cursor up to line 3Ø and to the right so that it is again on the A after the second quote. Press the SHIFT and INST/DEL keys at the same time. Each time you press the INST/DEL key a space is inserted and A+B is pushed to the right. Press the INST/DEL key several times until A+B is pushed past the right border. You will see how it wraps around to the left side of the screen, one line down.

Continue pressing INST/DEL until A+B moves to the far right. If you keep pressing, A+B will not go past the right boundary. This is because BASIC will not allow you to go past 8Ø characters, which is the maximum number of characters allowed on a program line. A program line is not limited to the same length as the 4Ø-character line on your screen. A program line starts when you enter a line number and ends when you press RETURN.

Press RETURN. RUN the program.

The results are identical to those you saw before. Each program line can be up to 8Ø characters long, that is, two lines on your screen, each of which can hold 4Ø characters.

What you have just done is shown that programs can be *crunched* by removing spaces or *expanded* by inserting spaces, that a program line can go past the end of the screen, and that a program line can have up to 8Ø characters. In order to make programs easy to read and copy, all program lines in this book are on one screen line and words are spaced to appear like normal printing. But you do not have to follow this rule. If you find it easier to crunch or spread out programs you can do so. Don't overdo the spreading out, though, or you may find your programs using too much memory and running slowly.

Adding to Programs

Adding program lines to the end of an existing program is no different from entering lines when originally writing a program. You enter a line number, the program statement(s), and press RETURN. Suppose, however, you want to insert program lines between existing lines of a program. BASIC has a very useful feature for doing this. BASIC will accept program lines in any order you care to enter them. It will then automatically sort the lines in ascending order and execute the program in that order.

To see this work enter NEW and the following:

```
1 REM *** INSERTING LINES ***
10 FOR J=0 TO 10
20 PRINT J
30 NEXT J
```

The English-like statements make it easy to read this program. First, RUN the program exactly as it is to see if it does what you expect. Then, move the cursor to the position after the J on line 20, and enter a *semicolon*. Now, RUN this version.

The first version prints values of J from 0 to 10 in one column. The second version prints the same numbers in one row. Placing a semicolon after a PRINT statement has the effect of suppressing the typewriter-like operation of carriage return/line feed that PRINT normally will perform.

Remove the semicolon to restore the program to its original version for the next entry. Now let's insert a line into the program. Enter the following:

```
28 PRINT
```

LIST the program and you'll see that BASIC inserted the new line in numerical order. RUN the program and you'll see a column of numbers from 0 to 10, this time separated by one line. The PRINT statement entered by itself, as in line 28, is used to insert blank lines on the screen.

Here's another change you can try. LIST the program and remove the J from line 30. RUN the program. There is no difference. The J in the NEXT statement is optional. Programs run faster and take less space if variables such as J are left out of NEXT statements. We include them to make it easier to follow the program logic, but feel free to omit them if you wish.

Let's add some more lines. Enter:

```
20 PRINT "J="J
25 FOR K=1 TO 6:PRINT J*K:PRINT J/K:NEXT
```

The first line has the same line number as an earlier line—20. BASIC will replace the previous line 20 with the new line 20. Even if the previous line were longer than the new line, all of the old line will be gone. This gives us another way of changing lines in addition to moving the cursor to the line and keying each change. Both methods are useful.

Line 25 contains four statements on one line, each separated by a colon. The

four statements on one line, make up a FOR . . . NEXT loop. In our program it does two things: It prints the product of J times K (J*K) and the quotient of K into J (J/K). When you entered this line on the screen, it took all 4Ø screen line characters, thereby forcing the cursor to the left of the next line. As you know, this is okay because two full screen lines are allowed on one program line. However, in this case one line is blank, as you will see when you list the program. Even though the cursor seems to be waiting for a new line, don't forget to press RETURN after the blank line.

When this program is RUN, a stream of characters will run down the left side of the screen faster than you can read it. The program stops when J equals 1Ø. You can then examine the results to see if they are what you expected.

Line 25 is called an *inner* FOR. . .NEXT loop because it is inside the FOR. . .NEXT loop defined by lines 1Ø and 3Ø. It will be executed one full cycle—that is, for K taking values from 1 to 6—each time the *outer* loop steps one J value. Thus, on the screen you see all values from K=1 to K=6 for J=1Ø.

Notice that the program puts a space between the equal sign and the 1Ø even though no space was entered on line 2Ø. This is because whenever BASIC prints a number an extra space is inserted in front of the number. If the number is positive the space is blank. If the number is negative, a minus sign is printed in the space.

Changing Line Numbers

Move the cursor so it is on line 2Ø. Change the line number to 21 and change both J's to K's. Press RETURN and LIST the program. The program now contains *both* line 2Ø and line 21. Changing a line number does not eliminate the changed line; rather, it adds a new line.

Debugging Tools

When a program is written, it is seldom correct the first time. It often has errors, called *bugs*. The process of finding the bugs and correcting them is called *debugging*. In this book you will be copying programs and may make errors while doing so. Quick Charts 1 and 2 at the end of this chapter will help you find typo errors. There are also two tools provided by BASIC that are useful for debugging. The first debugging tool is the STOP statement.

Add the following line to your INSERTING LINES program:

```
26 STOP
```

RUN the program. You will see a string of zeros followed by a message from BASIC—BREAK IN 26. When BASIC comes to the STOP statement, it does just that—it stops. It then prints the BREAK message followed by the line number of the STOP statement. If there is more than one STOP in a program, you will know exactly where the program stopped.

A nice feature is that after a program is STOPped it can be continued. Key in CONT for continue and press RETURN. The program continues until it finds STOP again. You can enter CONT until the program ends. As you can see, STOP enables you to isolate parts of a program to see how they work. This can be useful in isolating errors. Knowing where a typo is located is half the battle.

There will be times when STOPping the program interferes with BASIC so that the program cannot be continued from that point. In response to your CONT entry, BASIC will respond with a message that it can't continue.

The PRINT statement is the second debugging tool. It can be used either by itself or just before a STOP. By temporarily inserting a PRINT statement, you can ask for a display of the variables mentioned in the PRINT statement. Knowing the value of variables at various times in a program can often give a clue to where errors are located.

Deleting Lines

When a program is being changed to perform a new function it is often necessary to delete several lines. Deleting lines is a simple operation, but it requires care to avoid making errors that are difficult to find.

LIST the INSERTING LINES program. Key in the number 26 and press RETURN. LIST the program again. Line 26 has been deleted. When you enter any line number followed by all blanks that line will be deleted. Line deletion is very easy. Too easy. Lines can be deleted inadvertently, and it is difficult to find where deleted lines were. The deleted line can be far away from the part of the program on which you are currently working.

Lines can be deleted by mistake. One error is entering an existing line number and accidentally pressing RETURN before any further entries are made. A more common error is a simple typo. You intend to delete one line, but in fact key in the wrong number thus deleting another line. You then have two errors in the program because the intended deletion is still there. Line deletion can be made less error prone with the following technique.

LIST the INSERTING LINES program. Move the cursor up to line 1. Move the cursor right just past the line number. Press the space bar and hold it down until the line is all blanks. Press RETURN. Line 1 is deleted and the cursor is on the next line. Again move the cursor just past the line number and erase the rest of the line with the space bar. Press RETURN. By repeating these operations, you can delete a whole group of lines quickly and accurately.

This method of line deletion makes errors unlikely. If they do occur because

of wrong cursor positioning, they are immediately evident. This can save a good deal of debugging time.

Non-English Expressions

Not all of BASIC reads like English. Here are some examples:

```
A$="147"

PRINT ASC(A$)

PRINT CHR$(49)

PRINT CHR$(147)
```

Note that these statements don't have line numbers. BASIC executes such statements as soon as you press RETURN. These statements are in what is called the *immediate mode*. Since the statements don't have line numbers, no program is created from them.

The first statement assigns the value 147 to A$. The characters A$ are called a *string variable*. A string can be any group (string) of alphabetic, numeric, or other characters. The characters of a string must be enclosed in quotation marks, as is 147, above.

Strings can be stored in string variables. A string variable must end in a dollar sign—for example, A$. The first statement, above, stores the string of number "147" into the string variable A$.

A number not enclosed in quotation marks is not a string. It is called a numeric constant, or just a number. Numbers are stored in numeric variables that must not end with a dollar sign.

The rules in BASIC for strings is that string variables end in a dollar sign and the string itself must be in quotes. If you violate these rules, BASIC will tell you by displaying the error message TYPE MISMATCH. When you see this message, check your entries for dollar signs and quotes.

The second statement says ASC(A$). The function of ASC is to give the American Standard Code for Information Interchange (ASCII) code for the first character of a string. ASCII codes are widely used in computers and communication. You don't have to concern yourself with these right now, except to realize that when you see ASC, a translation from one code to another is being performed. The first character of A$ = "147" is a 1, so ASC will return the ASCII code for a 1, which is 49.

The third statement takes an ASCII code and produces a string in another code. If you enter this statement you will get a 1. In other words, the ASCII code 49 represents a 1.

Let X stand for some number. When the statement PRINT CHR$(X) is executed, two kinds of things can happen. The computer may print a character on the screen, as in the next to last example. Or it may perform some operation, as in the last example. The number 147 is a code for clearing the screen. Enter this statement and that is exactly what will happen.

When you see non-English expressions in programs, these are probably nothing more than translations of one code into another.

One more word on keying in program lines. When you enter a quote character, ", the cursor keys no longer control the cursor. Instead, they print strange symbols. A similar thing happens when you press the combination SHIFT–INST/DEL used for insertions. You may be able to use the INST/DEL key to delete the unwanted characters. But this method of operation can tie you in knots because you lose control of the cursor. Rather than fight it, press RETURN. This frees the cursor. Now you can go back to the problem line and clean it up.

PEEKing and POKEing in MEMORY

Computer memory is where data and programs are stored. Items are stored in memory by specifying a *location*, or *address*, and then either reading the contents of that location or writing in new data. When you turn on your Commodore 64, you get a message that says 64K RAM SYSTEM. The value 64K means that your computer has about 64,000 memory locations. Each location can store 1 *character*, or *byte*; therefore, you have a 64K-byte system.

Data are always transferred one character at a time. In order to transfer a string of characters, each character must be read or written individually and in succession.

A character is not the smallest unit in memory. The smallest unit is the *bit*. There are 8 bits in a byte; therefore, each character transfer involves the transfer of 8 bits. It is not possible to read or write 1 bit from memory. Only full bytes can be read or written. This has important consequences for some of the control functions we will be performing. We'll talk about this later.

Another unit of interest is the *nibble*. A nibble is half a byte, or 4 bits. The significance of a nibble is that it can store any of the digits Ø–9. It is too small to store alphabetic characters (26 characters plus special symbols); this requires a full byte. Since much data are made up of only numbers, they are often pushed into nibbles, two characters to a byte, saving time and memory. But nibbles, like bits, cannot be accessed from memory by themselves. They must first be accessed as part of a byte and then the byte must be split into nibbles.

Not all memory locations are used for storing data or programs. Some locations are used for *control*, which is a very useful way of handling the computer. Instead of having lots of extra codes for control purposes, we just read or write

into a memory location reserved for a specific function, and the control operation is performed. Examples are given below.

The BASIC statements that let us read and write into memory are named PEEK and POKE. We will be using PEEK and POKE often to transfer data and to control what the computer does. From the above discussion, it is clear that PEEK and POKE can only deal with one character at a time. When PEEKing we need to specify only one thing, the memory location we want to read. When POKEing we must specify the character that is to be stored as well as the memory location.

Here are some examples you can enter on your computer. First, clear the screen with the key combination RUN/STOP–RESTORE. Then insert the following:

```
POKE 55335,1
POKE 1063,83

PRINT PEEK(1063)

POKE 1103,PEEK(1063)
POKE 55375,PEEK(55335)
```

Note that these statements are entered without line numbers and are therefore executed in the immediate mode.

The first POKE writes a 1 in location 55335. This is one of the control locations that we mentioned above. It controls the color of the character displayed on the screen. Many of the control locations are high up in memory; they have numbers in the 50,000's. Data and programs are usually stored in low memory with addresses below 16,000.

The number 1 is the color code for white. By POKEing this into location 55335, we set the color of the character displayed by the second POKE.

The second POKE writes 83 in location 1063, which is where characters that are displayed on your screen are stored. POKEing into this area writes characters on your screen. Location 1063 corresponds to the upper right-hand corner of your screen. The 83 POKEd into this location is not what is displayed. It is translated by the computer into the figure of a heart. The same thing happens for letters and numbers. Codes are used that are very different from the actual characters displayed. The codes are chosen based on technical considerations, and tables are used to translate back and forth. See tables in Appendix E and F of the *Commodore 64 User's Guide*.

The third entry listed above is a PRINT of a PEEK. A PEEK cannot live by

itself. It must always be PRINTed, stored in a variable, or acted upon in some way. When you enter the third statement, the number 83 is printed on your screen. This is no surprise, since you're PEEKing right into the same location that you POKEd 83 into in the previous statement. I don't know why the inventors of BASIC decided to make us put parentheses around the address in a PEEK statement, but there it is.

The last two statements are interesting variations of the first two and are very pertinent to what we will be doing later. They may look complicated, but they're really not. The first of these says to write in location 1103 the result of a read from location 1063. When the computer sees this, it PEEKs into location 1063 and finds the number 83. It then POKEs this number into location 1103, which happens to be just one line below location 1063. The second statement does the same thing for the color code.

PEEKs and POKEs are used with OR and AND operators to turn individual bits on and off. This is a rather technical subject, which is not necessary to understand unless you plan to do some of this in your own programs. If you are interested, there is an explanation in Appendix A.

Disks and Tapes

Disks and *tapes* perform two functions: They provide large amounts of storage at low cost and they provide permanent storage when the computer is turned off. Computer memory is *volatile*, which means that when you turn off the switch or otherwise interfere with normal operation, all your data and anything else that was in memory is gone. The only way to make sure that your data are not lost is to store them on tape or disk.

The computer cannot work directly with the data stored on tape or disk. This data must be transferred into computer memory. When you have obtained something of value that is worth saving, the data can then be transferred from computer memory to tape or disk. When you transfer from computer memory to tape or disk, the data is *saved* or *written*. When you transfer from tape or disk to computer memory the data is *loaded* or *read*.

Disk and tape transfers are important often-used operations when working with computers. To make sure you can do the operations that will be needed in this book, the following sections are hands-on explanations of using disks and tapes. Disks and tapes are described separately; if you will be using tapes, skip to that section.

How to Use Disks

We will go through operations of initializing a new disk and saving and loading programs step by step. As you read this book, use these models until you become familiar enough with them to perform the operations on your own.

Initializing Disks. In order for the computer to store a file on a disk, it must have a place in which to store the file and a place in which it can keep a directory of where each file is stored. A new disk has neither of these things. It is a blank magnetic surface. The purpose of *initializing* a disk is to create locations, called *blocks*, and *directories* so that files can be written and later retrieved.

Turn on your TV, your disk unit, and your computer. Insert a new disk into your disk unit. Pull down the latch door; just pulling down is all that is necessary, the door will latch itself. Enter the following command:

```
OPEN 15,8,15
```

When you press RETURN, the computer responds with

```
READY
```

Now enter

```
PRINT#15,"NØ:WORK DISK,1"
```

When entering the above print command do not enter spaces. Everything up to and including the colon must be entered exactly as you see it. After the colon is the *disk name*. We called the disk WORK DISK, but you may give it any name you wish. After the name, there is a comma followed by an identifier, which can be any one or two characters you select. The N in NØ: stands for New Disk.

Press RETURN and the computer again responds with

```
READY
```

The disk unit makes some strange noises. Its red light comes on and you hear steady scraping sounds as the computer writes timing signals on the disk. Then the red light goes off and the disk stops. It is now ready for reading and writing.

To disconnect the command channel, 15, enter

```
CLOSE 15
```

We will test the next disk by writing a short program and writing to and reading from the disk.

Writing New Program Files on Disk. Enter the following test program:

```
10 REM ** TEST DISK **
20 PRINT "NEW DISK OK"
```

To write on disk, use the SAVE command as follows:

SAVE "TEST DISK",8

> **NOTE: This command can be used only the first time a file is written on disk. To write over existing files, see below.**

After you press RETURN, the red light comes on, there are some scratching sounds, the red light goes off, and the disk stops—all in just a few seconds.

The item within quotes is the *file name*. You can use any file name up to 16 characters. If you use more than 16 characters, the computer takes only the first 16.

The 8 is the *device name* for disk and must be included in all disk read and write operations. If you forget to enter the 8, the computer will think you have a tape unit. It will ask you to PRESS RECORD AND PLAY ON TAPE. If this happens, simply press the RUN/STOP key and correct your entry.

Now let's see if we can retrieve TEST DISK. Enter LIST. The computer responds by listing TEST DISK, showing you that SAVEing a program does not erase it from memory. You will often take advantage of this by SAVEing partially completed work and working on the part of the program still stored in memory.

Enter NEW to erase the program. Enter LIST again to see if the program was erased.

Reading Programs from Disk. The command for reading is LOAD, so enter

LOAD "TEST DISK",8

The computer may respond with

?LOAD ERROR

in which case move the cursor back to the LOAD command and reenter. This error is nothing to be alarmed about as long as it doesn't happen repeatedly. Sometimes it helps to open the disk latch door and reinsert the disk. When the computer reads the file without an error it responds with READY and no error message.

Enter LIST, RUN, or both to see if the LOAD worked.

Reading the Disk Directory. Let's check the disk directory to see if TEST DISK is recorded there. A dollar sign is used for the name of the disk directory so enter

```
LOAD "$",8
```

To see the directory enter

```
LIST
```

The computer will list the directory. The top line contains the disk name and identifier, "WORK DISK " 1, displayed in reverse colors. Then, sure enough, "TEST DISK" is listed followed by the file type PRG, an abbreviation for program. Other types of files are sequential and relative. Only program and sequential files will be used in this book. The directory tells you the type of each file stored.

Writing Over Existing Files. In order to prevent unintentionally writing on top of, or overwriting, an existing file, a safeguard is provided. When you want to change or replace an existing file with one having the same name, you must precede the file name with the characters @Ø:. First, let's see what happens if you don't include these characters.

Enter

```
SAVE "TEST DISK",8
```

The screen displays READY as if everything is all right. But the disk's red light is blinking, indicating an error. It is unfortunate that the screen gives no warning since it is easy to miss the signal from the disk until it is too late. It therefore makes sense to always check the red light after a SAVE.

You can correct the situation by entering

```
SAVE "@Ø:TEST DISK",8
```

The red light changes from blinking to steady, the disk goes through its scratching act, and the red light goes out. The file has been *overwritten,* as you requested.

I don't recommend this approach. Sometimes it works, but sometimes you can lose a file. I can't pin down what goes wrong or when, but the best way to protect files against overwriting is to SAVE them to nonworking disks and then back them up. More on this below.

Since you can use @Ø: whether a file is new or old, the safest approach is to always use @Ø:. It's a good idea to get in to the habit of using @Ø: when SAVEing files.

Disk Errors. We have mentioned two disk errors that can occur: overwriting and LOAD errors. Other common errors are forgetting to insert a disk or inserting the wrong disk, both of which result in error messages on the screen and the blinking red light. Let it blink. There is no easy way to turn it off. It will stop as soon as you enter a valid command. On pages 43–46 of the disk *User's Manual* there is a complete listing of errors.

Deleting Files from Disks. Avoid deleting files from disks. The SCRATCH command used for doing this has problems, and unless you know for a fact that it works properly in your computer, forget it. You can save up to 144 files on a disk. If the disk is full, transfer what you want to save to a new disk and then reinitialize the disk.

We have only described the disk commands used in this book. Other useful commands are described in your disk *User's Manual.*

How to Use Tapes

You will need to use the tape SAVE and LOAD commands for creating the programs in this book. We will demonstrate how each of these is used in sample commands. As you read this book use these models until you become familiar enough with them to perform the operations on your own.

To begin, insert a new or expendable tape in your tape unit. Rewind the tape to the beginning. Set the tape counter to Ø.

Enter the following test program:

```
10 REM ** TEST TAPE **
20 PRINT "TAPE OK"
```

Saving on Tape. Save the test program with the following command:

```
SAVE "TEST TAPE"
```

When you press RETURN, the computer responds with

PRESS RECORD AND PLAY ON TAPE

Press the two levers marked RECORD and PLAY on your tape unit. The red light will come on, the tape will move, and the screen will go blank, leaving you in limbo. After about 1Ø seconds the tape will stop and your screen will display

SAVING TEST TAPE
READY

Press the STOP lever on your tape unit. The red light will go off and the operation is complete.

Note that the tape counter reads about 7. If you are going to keep several files on the tape, you can save considerable time by keeping a log of the counter reading corresponding to the name of each file on the tape.

The procedure just described can be used with any program. Just substitute the program's name for the name "TEST TAPE," making sure to place the new name in quotes.

LIST the program you have SAVEd. The program is still in memory. This is an important and useful point. You can SAVE your work on tape without losing it in memory. Often you will want to SAVE a program and then continue to work on it in memory.

Loading from Tape. To test the LOAD command, first enter NEW to erase TEST TAPE from memory. Then enter LIST. This time your program is gone. To LOAD what you have written back into memory, rewind the tape and enter

LOAD "TEST TAPE"

When you press RETURN, the computer displays

PRESS PLAY ON TAPE

Press the PLAY lever on the tape unit. The tape unit starts and the screen goes blank for about 8 seconds. Then the screen displays

SEARCHING FOR TEST TAPE
FOUND TEST TAPE

The tape stops, so you might logically think the operation is complete. It is not. After several seconds, act two starts: The screen goes blank and the tape

starts up again. Your computer was only resting from its labor of finding the file. Now it's LOADing. This pause may be to let you scan the tape to find out what there is, but it's disconcerting until you get used to it. You can speed things up by pressing the space bar; then the tape unit will restart immediately.

After the file is LOADed, the screen comes back on and you get this message

```
LOADING
READY
```

Enter LIST and/or RUN to see if the program is really in memory.

Place several files on tape by writing new programs or changing the name of TEST TAPE to TEST TAPE 2 and to other names. As you SAVE these programs, notice that it is not necessary to press RECORD and PLAY levers for each SAVE. You can just leave them down until you are finished.

Rewind the tape to the beginning, and enter a LOAD command to read back the last file SAVEd. Each time the computer finds a file, it stops to let you know about it and then continues its search. Finally it finds the file you requested and LOADs it into memory.

If the PLAY lever is pressed before the LOAD command is entered, you don't get the PRESS PLAY ON TAPE message—an advantage when doing the cut-and-paste programming described later.

We have only described commands that are necessary for this book. There are other useful tape commands described in your tape manual.

Key Points. There are two key points about working with tapes that should be emphasized. First, the computer does nothing to prevent you from writing over valuable files. To prevent this, you must keep track of where your files are on tape. Keep a log of the file names and their locations indicated by the tape counter. Number each cassette and label each side of the cassette, entering these in your log.

Second, the computer does not maintain a directory of what is on tape. Don't trust your memory or rely on lengthy searches. Again, the solution is a log of tape files as described above.

How to Protect Your Files

Tapes and disks are wonderful storage media—if they are used properly. Neither tapes nor disks are the most rugged or permanent types of storage. They wear out. Consider also what might happen if there is a power failure while you are writing a file. Finally, think how easy it is to mistakenly overwrite a file, especially with tape. You must face the fact that after hours of work, you could lose everything unless you provide for the worst.

Providing for the worst means keeping *backup copies* of your files. This is so easy that it makes no sense not to do it.

You can backup a file by saving it twice. After you SAVE your file once, remove the tape or disk, insert the backup, and SAVE the file again. You don't even have to reenter the command. Just run the cursor back up to the command line and press RETURN.

Sometimes it is a good idea to develop a program on a *working disk* or *tape*. Then periodically you can transfer significant results to the permanent disk or tape and its backup. The working disk or tape gets the wear and tear and the coffee spills. The other two get treated like the family jewels.

As you gain experience with tapes or disks, you will get a feel for the number of read and write errors to expect under normal operation. When the number of these errors starts climbing, look out. Assuming the equipment is working properly, it is time to retire that tape or disk. Don't wait until the tape or disk becomes unusable.

How to Find Errors and Correct Problems

Quick Charts are available at the end of this chapter to help you find errors and correct problems. When you copy program lines it is easy to make errors. By using Quick Chart 1 as a step-by-step guide, you will be able to find and correct errors quickly. The vast majority of errors are the simple typos shown in items 1–3.

Tests have shown that people see what they expect to see. If you type FOR J=O TO 1ØØ, for example, you can look straight at it and not see that the letter O has been typed for the digit Ø if the line is part of a group of lines. As soon as you isolate the line or, even better, each character, the error becomes apparent. Therefore, if a scan of your entries does not find an error, try concentrating on smaller and smaller sections.

Learning to use a tape or disk unit takes some patience. As you gain experience, you will find that they are really quite simple to work with. Until then, Quick Chart 2 will help you solve most of your problems.

TIPS FOR EXPERIENCED USERS

Programs in this book are formatted for readability and ease of copying. This means that there are things in the statements than can be left out. You can safely crunch programs if you pay attention to the following points.

Since all branches are to executable code, most REM statements can be eliminated. In other words, the entry points to individual routines refer to the first line of executable code of the routine. The REM statements in which the rou-

tines are identified with a title line marked with asterisks can thus be left out. The exception is when the REM states a branch entry point, for example,

```
1098 REM * DUMMY BRANCH POINT *
```

Branch entry points should not be moved because later additions may branch to these points.

Routines can be reduced in size. They should not, however, be expanded or moved to higher line numbers because they may conflict with later additions. Many of the programs in this book are built up from short routines whose line numbers can be higher or lower than previously entered code.

It is best not to skip chapters, but if you do, the following groupings should be observed because they use related programs: Chapters 2 and 3; Chapters 4 and 5; Chapters 6, 7, 8, and 9.

PROGRAM TEMPLATES

Why reinvent the wheel? A good deal of any one program is similar to other programs. Instead of reentering the same program lines over and over, the lines can be saved on disk or tape and reused at any time.

Although programs or sections of programs may be similar to previously written code, they are rarely identical. There is always a need for some changes. Hence, we cannot just string previously written program lines together without some adjustment. This is the function of *templates*. Using the techniques described below, you will be able to display a template on the screen, make the necessary changes, and integrate the modified lines into a new program. The result is less time spent in the boring task of keying in programs, fewer keying errors, less frustration hunting down program bugs, and more time enjoying your computer.

Templates can be a few lines of code or complete programs of any length. The criteria of whether to save program lines as a template are the likelihood of whether you will use those lines again and how much work you will save by so doing. As an example, when using the music facilities of the Commodore there are several initial program statements that are usually required. When you are learning to use the sound features a template that performs all initial tasks will be useful to you. As you gain experience, you may find that you can hammer out the initializing lines on the keyboard faster than you can pick them off a tape or even a disk.

Individual program lines can often be used as templates when writing programs. A great many program lines in this book differ in only one or two characters. Rather than keying each line, you can just change the line number, move the cursor to where changes are necessary, and press RETURN. Repeated use

of this technique can transform the entry of long programs into a rather simple task that is more like play than work.

CUT-AND-PASTE PROGRAMMING

There are a number of tricks you can use to save time and effort when entering a new program or when modifying an old program. These tricks are based on the way BASIC handles line numbers. Adding, changing, and deleting program lines can often be reduced simply to entering line numbers.

Pasting Program Lines

If you have a large program, A, to which you want to add lines from another program, B, you can do so without keying in the new lines. In addition to saving time and effort, you have the assurance that if the lines were previously correct, they are likely to be correct after pasting. Here are the steps for pasting lines from program B into program A. You don't have to memorize these steps. As we go along we will explain how to do this in each particular case.

1. LOAD program B into memory.
2. LIST the lines you want to cut from B to paste into A. With disks, you can list up to 19 lines at a time. With tapes, you can list up to 18 lines at a time.
3. LOAD program A into memory. This deletes all of program B from memory. But the lines from B remain displayed on your screen.
4. Move the cursor to the first line of B displayed on your screen.
5. Press RETURN. This enters the displayed line of B into A, and the cursor moves to the next line displayed.
6. Repeat point 5 for each line of B that is displayed.
7. If there are more lines to be pasted, SAVE program A, and repeat points 1–7.

This process will either add new lines to A or replace old lines if the line numbers from B match those from A. There will be plenty of examples of how to do this as we go along, so we will not give examples at this point.

The total number of lines that can be displayed on the screen at one time is 25. Since some of these lines are taken up by the messages BASIC sends to the screen in the process of loading program A, the number of lines you can enter is reduced. If you try to enter too many lines, they will be pushed off the top of the screen by the messages from BASIC.

From the above description, pasting in program lines may seem like a ticklish operation, but it is not. If lines get pushed off the screen, the worst that hap-

pens is that you repeat the process for the lost line. You can make multiple entries with no harm being done. You can always list the lines in program A so you know exactly what has been pasted and what remains to be done.

QUICK CHART 1 GETTING PROGRAMS TO WORK

1 Check the most common typo errors

- □ Ø or 0 entered for 0 or Ø
- □ 1 or I entered for I or 1
- □ 8 or B entered for B or 8
- □ ; or : entered for : or ;
- □ S entered for $

2 Check punctuation

- □ Missing quotes—"
- □ Missing colons—:
- □ Missing semicolons—;
- □ Missing or added parenthesis—( )
- □ Incorrect spacing when quotes are separated by blanks—" "
- □ INPUT # entered for INPUT# (space not allowed)
- □ PRINT # entered for PRINT# (space not allowed)

3 Check line numbers

- □ After adding or deleting lines, line numbers are a frequent source of error.

4 Check line count

- □ Count the number of lines entered and count the number of lines shown in the book. Besides catching missed lines, you will often catch incorrectly deleted lines.

5 Error messages

- □ ?SYNTAX ERROR
  A BASIC statement was copied incorrectly. In addition to items 1–4, check each statement against the original.
- □ OTHER ERROR MESSAGES
  In addition to items 1–4, check your entries line by line against the original.

QUICK CHART 1 (Cont.)

6 LIST the program in sections that fit on the screen, for example, LIST 800–820.

- ☐ If there are lines present that should not be there, you forgot to enter NEW at the start of a new program or you forgot to delete unwanted lines.
- ☐ If there are whole sections missing, you entered NEW when program lines you needed were in memory. Or you neglected to paste new lines onto a program after LOADing into memory.

QUICK CHART 2 SOLVING DISK OR TAPE PROBLEMS

1 A file that was written cannot be found.

- ☐ Tape was not rewound or was not rewound enough before reading. Was the counter reset after the last rewind?
- ☐ The name used to retrieve the files was not *exactly* the same as that used to record the file.
- ☐ Check a disk's directory for the *exact* name of a file. Be sure singular and plural words in the file name are singular and plural in your entry.
- ☐ Tapes don't have directories. You can find what files are on the tape by using a nonexistent file name in the LOAD statement. The tape unit will search the whole tape, displaying the name of each *program* file on the tape. For example, enter LOAD "NO FILE".
- ☐ If the file cannot be found by the above methods, the file was incorrectly SAVEd or was overwritten. Always move the tape well past the last entry before writing a new entry. Don't place files too close to each other, and don't insert a file between other files.

2 On reading from tape, the error message, STRING TOO LONG, is received.

- ☐ Back up the tape, being sure it is well ahead of the file you are reading. Then try again.
- ☐ An end of record marker may be missing. It may be necessary to rewrite the file. First, carefully check program lines that read and write files, for example, lines 1900–3000.

3 When using disk, the message PRESS PLAY ON TAPE or PRESS RECORD AND PLAY ON TAPE is displayed when you are trying to read or write a file.

- □ The device number, 8, was not entered so BASIC thinks you have a tape unit:

  Wrong: SAVE "@Ø:LIGHT SHOW"
  LOAD "LIGHT SHOW"

  Right: SAVE "@Ø:LIGHT SHOW",8
  LOAD "LIGHT SHOW",8

4 When writing or reading with tape, a DEVICE NOT PRESENT mesage is received.

- □ Check format of LOAD and SAVE statements. An incorrect device number may have been entered.
- □ Check the OPEN statements. You may be using the disk format instead of the tape format for these statements. The correct format for tape is:

  WRITE: OPEN 2,1,1,F$
  READ: OPEN 2,1,Ø,F$

5 The red light on the disk unit blinks on and off.

- □ There may not be a disk in the disk unit.
- □ The latch door of the disk unit may be open.
- □ You may be trying to save a file that already exists. This requires the parameter @Ø:. For example,

  SAVE "@Ø:LIGHT SHOW",8

CAREFUL: Don't correct this situation by reentering the command using the same disk. Make a temporary switch to a working disk, SAVE your file, and then go back to the previous disk.

- □ The disk you are trying to write on may be protected. A piece of tape over the notch on the edge of the disk protects the disk against writing.
- □ If the screen displays the error message FILE NOT FOUND, check the *exact* name of the file in the disk's directory against your entry. Verify that the correct disk is in the disk unit.

2

A New World of Color

Your Commodore 64 opens up a whole new world of color for you. In this chapter we will explore some of the fascinating things that can be done with computer-generated color patterns.

Let's start with a simple program that will display the Commodore 64 character set. This program will show you how screen and color memory is used. Copy the following program into your computer:

```
1 REM *** CHARACTER SET ***
2 REM DISPLAY THE COMMODORE 64 CHAR-
3 REM ACTER SET IN BOTH NORMAL AND
4 REM INVERSE MODES.
20 PRINT CHR$(147)  :REM CLEAR SCREEN
30 S=1024
40 FOR K=0 TO 255
50 POKE S,K:POKE S+54272,5
60 S=S+2
70 NEXT K
80 GOTO 80
```

Line 1 is the program name. All the programs and routines in this book will start with a name or title. Lines 2–4 are REMarks; they contain the type of information that you may want to include in programs that you SAVE for future use. From now on we will not include such information in programs in this

book, but there will always be 1Ø–2Ø spaces between the title and the first non-REM line for any REMarks you may wish to include.

When you RUN a program, the computer does not automatically clear the screen. Anything displayed at the time you enter RUN will remain on the screen until it is overwritten by your program. To provide a clear screen at the start of a program, we enter the code CHR$(147) as in line 2Ø. The code CHR$(147) is simply recognized by BASIC as a command to clear the screen. If we had entered a number other than 147, CHR$ would have been interpreted differently.

Lines 3Ø–7Ø show you how displays are created and controlled on your computer. A clear understanding of this powerful capability will enable you to modify and expand the programs presented in this book.

Figure 1 shows you how characters and colors are stored in memory. Characters are stored in *screen memory*, and their corresponding colors are stored in *color memory*. Screen and color memory are two distinct areas separated by 54,272 character locations.

Screen memory is a set of memory locations, one location per character, numbered from 1Ø24 to 2Ø23. A character stored in a specific location in this area will be displayed in the corresponding location on your screen. In line 3Ø of our program we initialize the variable S to 1Ø24, the starting location of screen memory. Memory location 1Ø24 corresponds to the upper left-hand corner of the screen, location 1Ø25 corresponds to the second character position in the top row, and so on, to location 2Ø23, which corresponds to the rightmost position of the last row. The screen displays 25 rows of 4Ø columns each, for a total of 1ØØØ character positions. By setting S to 1Ø24 in line 3Ø, we ensure that the first character displayed will be in the upper left-hand corner of the screen.

In this and other programs, we will use C for color and K for character. All characters are stored in the computer as numbers. Characters may be letters, numerals, punctuation marks, graphic symbols shown on the keys on your keyboard, or special symbols such as < or @. Each character is assigned a unique number from Ø to 127. Appendix E in your *Commodore 64 User's Guide* shows the numbers that are associated with each character. There are actually two sets of characters, and, as you will see, our program will display both sets.

You can display a character in either *normal* or *reverse mode*. In reverse mode, background and foreground colors are interchanged. The modes allows you to switch, for example, from light characters on a dark background to dark colors on a light background. To generate characters in reverse mode, all you have to do is add 128 to the character's normal mode number. For example, the number for the letter A is 1, therefore, its inverse is 129 (1 + 128).

We can now interpret what line 4Ø in the program is telling the computer to do. It says to take all characters in the normal mode (Ø–127) and then take the same characters in reverse mode (128–255). In other words, it assigns to the variable K all values from Ø to 255.

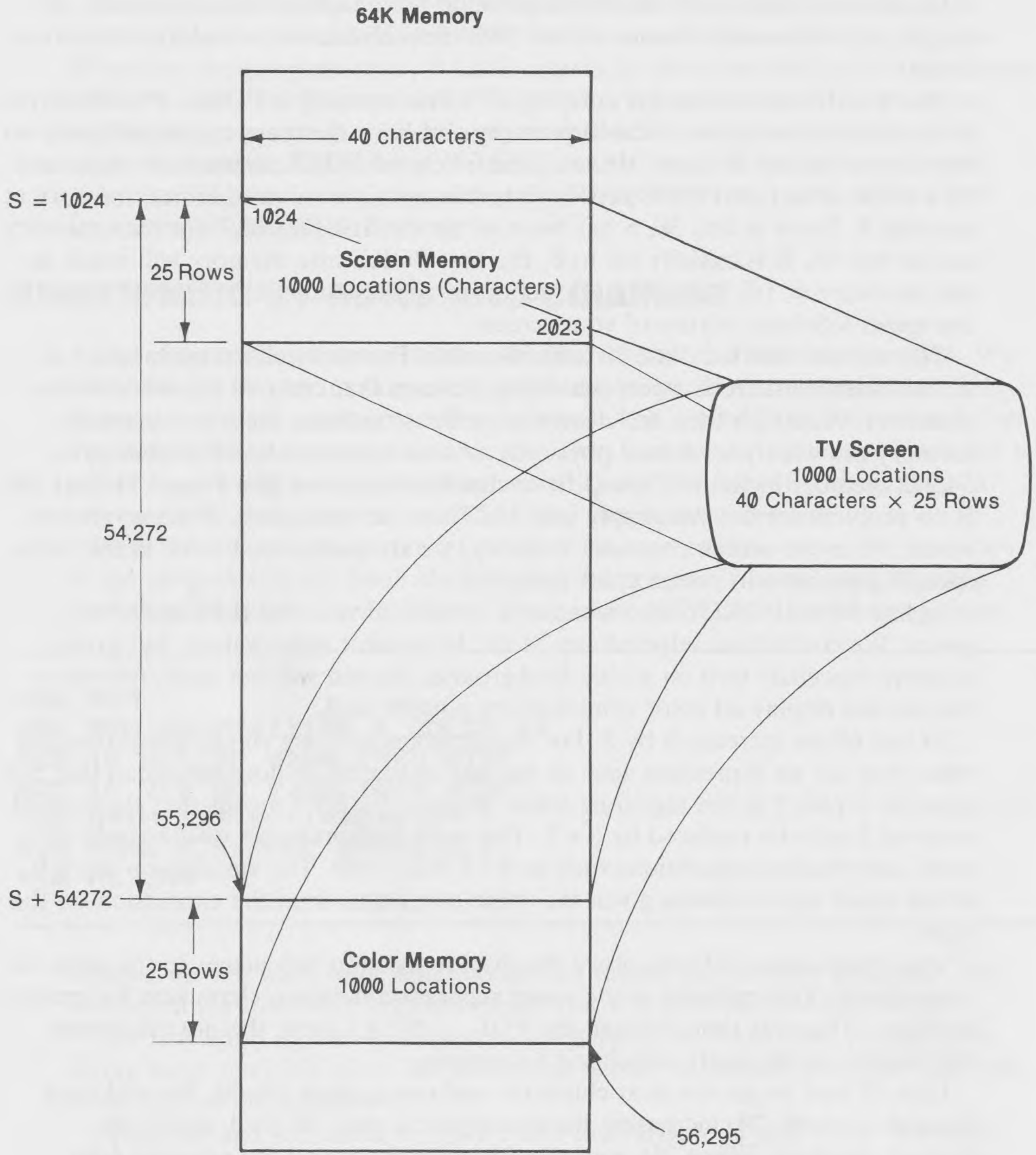


Figure 1. Display of characters and color on the screen.

In order to display any one of the possible 128 characters on the screen, we simply store its number in one of the 1000 memory locations reserved for screen display.

The BASIC instruction for entering data into memory is POKE. POKE allows us to poke characters into screen memory and have them appear immediately on the screen. In line 50 there are two closely related POKE instructions separated by a colon. The first POKE says to poke the numeric value of K into memory at location S. Since in line 30, S has been set to the first location of screen memory and in line 40, K is initially set to 0, the first POKE into memory will result in the character @ (@ has a numeric code of 0 in the computer) being displayed in the upper left-hand corner of your screen.

The second POKE in line 50 controls color. For every character location in screen memory there is a corresponding location that controls the color of the character. Although they are closely related instructions, these two areas of memory are widely separated physically. As we mentioned, color memory is 54,272 memory locations (bytes) from character memory (see Figure 1). But this is no problem for us. We simply add 54,272 to our variable S. Then no matter where we are in screen character memory, we are guaranteed to be in the corresponding position in screen color memory.

In line 50 we POKE 5 into location S+54272. Five is the color code for green. We could have selected any of the 16 possible color values, but green displays especially well on a blue background. As you will see later, television sets do not display all color combinations equally well.

In line 60 we increase S by 2. For nonprogrammers, we should point out that when you see an expression such as the one in line 60, it does not mean that S is equal to S plus 2 in the algebraic sense. Rather, S=S+2 means that the current value of S is to be replaced by S+2. The same interpretation would apply to more complicated expressions such as S=4*S/K+100. The variable on the left of the equal sign is always given the value calculated from the expression on the right.

The effect of line 60 is to move the display location two places to the right on your screen. This provides a one space separation between characters for greater legibility. The next time through the FOR. . .NEXT loop, the next character will display at the newly calculated S location.

Line 70 says to get the next character and repeat lines 40–70. We will loop through lines 40–70, increasing the character number, K, by 1 each time through the loop. When 256 is reached looping stops and the program falls through to line 80. Eventually the display will come to the end of a line at which time it will be necessary to move down one line and to the left margin of the screen. There is no provision in the program for this, and there doesn't have to be. The computer automatically maps consecutive locations in memory onto the screen in lines of 40 characters each.

Line 8Ø says to go to line 8Ø! It would seem that not much is being accomplished. But without this loop BASIC would come to the end of the program. Whenever the program ends, BASIC returns to the command mode and displays READY and a flashing cursor on the screen. This would destroy part of our display; to prevent that we need only keep BASIC occupied in the program mode until we are ready for it. To bring BASIC back to the command mode from the program mode, press either the RUN/STOP key or the RUN/STOP key and the RESTORE key together.

POSITIONING BY ROWS AND COLUMNS

Our program has a space between characters, but the lines are crowded. We would like to be able to skip every other line to create a more readable display. We could set up a counter to tell us when we had reached the 40th column, but it would be better if we could work with rows and columns directly. Since this is something we will want to do often, we will code the following program as a subroutine so you can save it and add it to programs as needed.

First, SAVE your character set program on tape or disk. Since the first lines of the programs in this book always show the program's name, save it, using CHARACTER SET for the name. Then, enter NEW and the following lines:

```
400 REM
401 REM ** POSITION & COLOR **
420 IF H<1 OR H>40 THEN STOP
425 IF V<1 OR V>25 THEN STOP
430 S=(H-1)+(V-1)*40+1024
435 POKE S,K:POKE S+54272,C
440 RETURN
```

Since we will want to be able to add this subroutine to other programs, we numbered the lines with high numbers. You will soon see how this works.

The first entry at line 4ØØ is simply a separator, or blank line, to make the listing more readable when the subroutine is part of another program. Line 4Ø1 is the title of the subroutine.

Lines 42Ø and 425 check to ensure that the variable H and V are *in bounds*. We used the letters H for *horizontal* and V for *vertical* (rather than rows and columns) to designate character position across and down the screen, respectively. Whenever POKEing around in memory, it is always a good idea to prevent POKEing into the wrong places. If you let H or V go out of bounds you might find that your program is garbled or the computer hangs up so that nothing you do works. If this happens, press the RUN/STOP and RESTORE keys

together. Or you may just have to turn off the computer, turn it back on, and reLOAD your program. To avoid these complications, we use lines 420 and 425 to guard against invalid POKEs.

Line 430 computes the same variable, S, that we used in the previous program, but now S is derived from the values of the H and V position variables. Since it is easier to work with H values 1–40 and V values 1–25 rather than 0–39 and 0–24, respectively, 1 is subtracted from both H and V to give the correct value of S. The expression for S could be simplified algebraically, but it would be less easy to read. Since in the present case there is no need for computational speed, we'll leave the expression as is.

Because there are 40 characters per line, the vertical position (V-1), which is the number of lines, is multiplied by 40 in line 430. The result is added to the horizontal position (H-1), which is the number of characters from the left side of the screen. The starting position of screen memory, 1024, is added to give the character location in screen memory.

Line 435 is the same as line 50 of the previous program, except that the variable C has replaced the constant 5. This is the key to creating fascinating color combinations. You will soon see how changing the value of C makes this simple subroutine produce all kinds of color displays.

The last line, 440, does just what it says: It RETURNs to the program that called this subroutine to the line following the GOSUB statement.

After you have keyed in this subroutine, SAVE it to disk or tape with the name POSITION & COLOR. Don't try to RUN it because it is meant to work only as part of a larger program.

HOW TO COMBINE PROGRAMS

We now have two separate programs, and we would like to combine them into a single program. The way BASIC uses line numbers makes this easy. By manipulating line numbers, BASIC programs can be assembled from bits and pieces. This is the cut-and-paste programming described in Chapter 1.

First, we will modify the Character Set program so that it works with the subroutine. Then we will graft the subroutine onto the modified CHARACTER SET program.

LOAD the CHARACTER SET program into memory and LIST it. Then make the following entries, starting with the cursor at the bottom of the screen to ensure that all entries will be made on clear lines:

```
30 H=1:V=1:C=5
40 FOR K=0 TO 255
50 GOSUB 420
60 H=H+2:IF H>40 THEN H=1:V=V+2
```

Each of the lines you just entered replaces the previous lines with the same number. You can confirm this by LISTing the program again. If any of the above lines is not in the new listing, it's probably because you forgot to press the RETURN key at the end of the line.

Place the cursor just under the new listing and LOAD the subroutine POSITION & COLOR. *Do not LIST it.* Whenever you LOAD a new program the previous program in memory is erased and is replaced by the new program. Thus, CHARACTER SET is no longer in memory, having been replaced by POSITION & COLOR. But the listing for CHARACTER SET is still displayed on your screen! This means you can add CHARACTER SET to the subroutine merely by placing the cursor on each line and pressing the RETURN key.

Move the cursor to the first program line of CHARACTER SET. Press RETURN. The cursor obligingly moves onto the next line. Press RETURN again. Press RETURN for all lines in the listing. Now enter LIST and stand back. The two programs are combined into one.

At this point you may want to eliminate some of the REM lines. To do this move the cursor to the line you want to delete, and use the space bar to remove everything but the line number. Then press RETURN. The line is eliminated. Always check the result by LISTing the lines you have been working with. To correspond to the finished program listing printed below, delete lines 2–4. LIST the program again to check that the deletions have been made properly.

Before RUNning, always SAVE the new program. Otherwise, if there is an error, you may lose the program and have to recreate it. SAVE this program under its old name, CHARACTER SET. If you are using disks, remember to prefix the name with @Ø: when writing to an existing file.

Now it's safe to RUN the program. I think you will agree that the new format of the display is an improvement.

We pointed out above that there are actually two character sets. To display the second set, hold down the Commodore key and press the SHIFT key. The Commodore key is located at the far left on the bottom row of your keyboard. Repeat this operation to restore the first set. Both sets occupy the same area of memory, but the computer interprets the character codes differently in the two cases. Note that the blank spaces listed in Appendix E of the *User's Guide* under the SET 2 column are actually filled in on your screen with the values given in the SET 1 column.

If your program doesn't work, LIST it and carefully compare it to the following listing of the full program.

```
1 REM *** CHARACTER SET ***
2 REM DISPLAY THE COMMODORE 64 CHAR-
3 REM ACTER SET IN BOTH NORMAL AND
4 REM INVERSE MODES.
```

```
20 PRINT CHR$(147)   :REM CLEAR SCREEN
30 H=1:V=1:C=5
40 FOR K=0 TO 255
50 GOSUB 420
60 H=H+2:IF H>40 THEN H=1:V=V+2
70 NEXT K
80 GOTO 80
400 REM
401 REM ** POSITION & COLOR **
420 IF H<1 OR H>40 THEN STOP
425 IF V<1 OR V>25 THEN STOP
430 S=(H-1)+(V-1)*40+1024
435 POKE S,K:POKE S+54272,C
440 RETURN
```

PATTERNS OF LIGHT

Programs don't have to be complex to generate pleasing patterns of light. Here is an example:

```
1 REM *** LINE PATTERN ***
20 PRINT CHR$(147)   :REM CLEAR SCREEN
30 K=64
40 FOR S=1024 TO 2023
50 POKE S,K:POKE S+54272,C
60 C=C+1:IF C>15 THEN C=0
70 NEXT S
80 GOTO 80
```

For the time being we have abandoned horizontal and vertical coordinate positioning. When you want to cover the screen with a simple pattern, the easiest method is just to paint the screen in consecutive positions from start to finish. When RUN, this program paints thin lines with repeating patterns of color.

In line 30, the character is set to 64, which is the code for a simple horizontal line. Line 60 is where interesting things happen. The color, C, is incremented by 1 on each pass through the loop. Since there are only 16, 0–15, allowable color values, if C becomes greater than 15 ($C > 15$), we recycle through all the color values by resetting C to 0.

By entering the following three lines you will be able to change the character K, displayed in LINE PATTERNS:

```
80 GET A$:IF A$="" THEN 80
90 K=ASC(A$)
95 GOTO 40
```

This is a very crude method of keyboard entry, because when you press a key there is no easy way of knowing what character K will be displayed. We'll soon present some real keyboard control. For now, simplicity of the above lines allows us to explain some important points.

Line 8Ø replaces the previous "do nothing" loop and does some interesting tricks. The first statement on line 8Ø, GET A$, says the following: Get a keyboard entry and store its value in A$. This is powerful because it provides a way for you to control programs directly from the keyboard. Instead of having to code all information in the program itself, you can make the program stop and ask what you want to do. The key you press will determine the program's action. This opens exciting possibilities, which we'll we using extensively in what follows.

The dollar sign wasn't tacked onto the A to make typing harder. In BASIC the dollar sign is used to designate a variable which can be any type of character—alphabetic, numeric, or special. If the dollar sign is left off, such as in the variables S, K, and C, that we have used previously, the variables can only have numeric values. Therefore, A$, called a *string variable*, is a type of variable that can store the value of any key on the keyboard.

The second statement on line 8Ø has two quote marks in succession. Don't place a space between them, but type them one after the other. Two quotes in succession represent the null character. A *null character* is a character that is and is not at the same time. The null character isn't displayed on the screen and isn't printed on the printer, yet it has a number value like any other character. It's used to signify the absence of any of the other characters. In line 8Ø if no key is pressed, A$ will be null. Thus, the second statement on line 8Ø says if A$ equals the null character, loop back and keep trying until somebody, anybody, presses a key. When, at long last, a key is pressed, the program can proceed to the next line.

Line 9Ø is the first step toward giving you keyboard control. Think of it as a preview of things to come. It tries to establish a correspondence between a key on the keyboard and the character that the computer displays on the screen. It is only partially successful. If you look at Appendixes E and F in the *User's Guide*, you will see keyboard characters in one column and their corresponding codes in the column to the right. The codes in Appendix F are those generated from the keyboard, whereas those in Appendix E are those used to POKE into

memory for display. As you can see, there is no simple relationship between all these codes, which is why line 90 is only partly successful.

When you RUN the program, you will first get a line pattern over the whole screen. Now press a key. A new pattern will be generated with the graphic symbol taken from the right side of the key. Use the SHIFT key to generate inverse patterns. Note the nice 3-D effects some patterns produce. This program is fun to play with, but eventually you will want to move on to more sophisticated processing. Try SHIFT–space bar.

COLOR CONTROL

What's the best way to control color? There are 16 possible colors. Each of these 16 colors can be used for characters, background, and border; that's 16 × 16 × 16, or 4096 possible combinations! If we are going to exploit the possibilities of the computer, we want to be able to produce any of these combinations easily and quickly. No SHIFT keys or multiple key combinations are allowed. I don't like them and I don't think anybody else does. I'll describe the best color control method I could work out and how it works. Maybe you can find a better way. Designing good key assignments is one of the most difficult and challenging parts of program design.

The available colors in the Commodore 64, their corresponding values, and the keys we will use to designate them for color control are as follows:

Color	Value	Key
BLACK	0	BLK
WHITE	1	WHT
RED	2	RED
CYAN	3	CYN
PURPLE	4	PUR
GREEN	5	GRN
BLUE	6	BLU
YELLOW	7	YEL
ORANGE	8	D
BROWN	9	C
LIGHT RED	10	E
DARK GRAY	11	Q
MEDIUM GRAY	12	A
LIGHT GREEN	13	Y
LIGHT BLUE	14	U
GRAY	15	Z

Here is the logic behind the above key assignments. The number keys, 1–9, on your keyboard will designate the same color that is printed on the front of these keys, that is black–yellow, except that no SHIFTing will be required.

Pressing one of these keys will immediately select the color of a character. Color variations will be selected from the key diagonally down and to the right of the color key. Thus, shades of gray will be down and to the right of the BLK key: Q will be dark gray. A will be medium gray, and Z will be light gray. Y will be light green and U will be light blue. All redlike colors—light red, orange, and brown—are on the diagonal down from the red key. We hope you will find this association of the color variations with the primary color easy to remember.

The f keys from f1 to f8 are called *function keys*. They can be assigned any function that we want to give them. The f1 key will control border color, and the f3 key will control background color. As you will see, both of these can have a profound effect on the appearance of a display, and you will have frequent use for them.

KEYBOARD ENTRY

We will now describe a keyboard entry subroutine that incorporates all the color control described in the previous section. Our ultimate objective is a general-purpose keyboard entry routine that you will be able to use in many ways. We will start with a simpler routine that controls color only. Later we will expand its capabilities to include controls other than color.

There is a method of entering program lines in BASIC that can save time and effort. Notice that in the program shown below many lines are similar. It is not necessary to key in each of these lines. Instead, type in one line and press RETURN. Then move the cursor to that line and type a new number over its line number. Move the cursor along the line making any necessary changes as you go. When finished press RETURN and enter LIST. You now have two similar lines displayed. Move the cursor back up to the first line and change the two lines you now have to work with. The process can be repeated as often as necessary. Remember, new line numbers can be entered in any order; BASIC sorts the lines automatically. In the program shown below, use this technique to enter the following groups of lines: 545–580; 595–600; 649, 659, and 674; 670 and 680; 660 and 675. Enter NEW and then, using the technique described for some of the lines, enter the following:

```
500 REM
501 REM  ** SUB.: COLOR KEYS **
520 GET K$:IF K$="" THEN 520
545 IF K$="D" THEN K$="9"  :REM ORANGE
550 IF K$="C" THEN K$="10" :REM BROWN
555 IF K$="E" THEN K$="11" :REM LT. RED
560 IF K$="Q" THEN K$="12" :REM DK. GRY
565 IF K$="A" THEN K$="13" :REM MD. GRY
```

```
570 IF K$="Y" THEN K$="14" :REM LT. GRN
575 IF K$="U" THEN K$="15" :REM LT. BLU
580 IF K$="Z" THEN K$="16" :REM LT. GRY
595 IF K$=CHR$(133) THEN 660 :REM F1
600 IF K$=CHR$(134) THEN 675 :REM F3
648 REM
649 REM * SET CHARACTER COLOR *
650 IF VAL(K$)>16 OR VAL(K$)=0 THEN 520
652 C=VAL(K$)-1      :REM COLOR NO.
655 RETURN
658 REM
659 REM * SET BORDER COLOR *
660 C0=C0+1:IF C0>15 THEN C0=0
670 POKE 53280,C0:GOTO 520
673 REM
674 REM * SET BACKGROUND COLOR *
675 C1=C1+1:IF C1>15 THEN C1=0
680 POKE 53281,C1:GOTO 520
695 GOTO 520
```

The subroutine COLOR KEYS cannot work by itself, but we will soon add to it so you can RUN it.

After LISTing and checking your entries, SAVE the subroutine to tape or disk with the name COLOR KEYS. COLOR KEYS will be used in the next chapter, so make a note of where you store it, especially if you are using tape.

Here's how the subroutine works. Line 52Ø is already familiar; it's a way to get a keyboard entry. Lines 545–58Ø simply translate the alphabetic keyboard entry into the color code assigned to the key. Since black–yellow are entered on number keys 1–8, there is no need for the program to contain a translation. Note that each of these numbers is one greater than the correct value. This is corrected in line 652, where one is subtracted from the value of K$.

Lines 595 and 6ØØ check to see if function key f1 or f3 has been pressed. If f1 has been pressed, we go to lines 66Ø–67Ø to set border color. Line 66Ø adds 1 to the current color value of the border, CØ. Then line 67Ø pokes the new value into the memory location that controls border color. At this point the border color is immediately changed; there is no need to return to the calling program, so the subroutine just recycles back to line 52Ø until a character color is selected. Lines 675–68Ø for the background color work in the same way.

If neither f1 nor f3 is pressed, the program will fall through to line 65Ø, which tests to see if any of the valid color keys have been pressed. If not, the program ignores the key and loops back to line 52Ø for another try. At line 652, K$ must hold a valid color. Then, VAL(k$) converts K$ to its numeric value. The subroutine returns to the calling program with C set to the color value.

The following short program will show you the use of the COLOR KEYS subroutine. After the COLOR KEYS subroutine is in memory and SAVEd to tape or disk, make the following entries:

```
1 REM *** COLOR CONTROL ***
20 PRINT CHR$(147)   :REM CLEAR SCREEN
30 K=160:S=1024      :REM REVERSED SPACE
35 GOSUB 520
40 FOR S=S TO S+C+10
45 IF S>2023 THEN S=1024
50 POKE S,K:POKE S+54272,C
70 NEXT S
95 GOTO 35
```

LIST the new program, and verify that the new lines are indeed appended to COLOR KEYS. When a program, such as this one, exceeds the length of the screen, you must specify the lines that you want displayed on the screen. Use LIST sn–en, where sn is the start number and en is the end number of lines to be listed, for example, LIST 520–600. You can use LIST –en to list all lines up to en or LIST sn– to list all lines following sn. For example, list –600 will list all lines up to line 600 and LIST 520– will list all lines from 520 to the end of the program.

In line 30 we select a reversed space character for display by setting K=160. This gives a colored square at the screen location where it is displayed.

Line 35 calls your COLOR KEYS subroutine, which returns with whatever colors you have selected for border, background, and character. Then the program enters a FOR. . .NEXT loop, which paints the inverse space character on the screen. Line 40 is interesting. We have S ranging from S to S+C+10. In other words, the positions that will be painted will range from the current position to C+10 positions to the right. Since C is the color value returned by the COLOR KEYS subroutine, the resulting pattern will be a series of bars whose lengths and colors depend on the color you select. The 10 was arbitrarily tacked on to C to extend the length of the bars. You can change this as you see fit. You might also want to try different values of K.

Save the program with the name, COLOR CONTROL. This program will be used in Chapter 3 so keep a record of where you store it, especially if you are using tape.

When you RUN this program you will first get a blank screen. Press one of the color keys we have been describing, and a bar of that color will be painted on the screen. As you press other color keys, bars will be added in adjacent spaces. By pressing f1 or f2, you can cycle through all 16 possible colors for

either border or background. If the screen becomes full, the display is erased. Press the RUN/STOP and RESTORE keys to regain control.

Now you have color control at your fingertips. The following section will describe some programs you can use with your new color control capability.

SPIRALS AND SYMMETRY

Let's make *spirals*. In addition to providing interesting displays, spirals give us a chance to demonstrate cut-and-paste programming techniques.

We are going to combine the subroutines POSITION & COLOR and COLOR KEYS and then add a main routine for painting a spiral. LOAD the subroutine POSITION & COLOR first because it is the shortest. LIST the program; then, with the cursor right under line 440 LOAD the COLOR KEYS subroutine. When the computer displays READY, move the cursor to the first line, line 400, of the displayed listing. Press RETURN repeatedly and watch the cursor move down until all lines are entered. LIST your entries and verify that the two programs have been combined.

Now add the following lines to those you have assembled:

```
1 REM *** SPIRAL ***
20 X=2:PRINT CHR$(147):REM CLEAR SCREEN
25 K=78:V=X:Y=1:H2=40-X
30 H1=1:V1=V:V2=25-X:H3=1:V3=1
34 REM
35 REM HORIZONTAL LINE
40 FOR H=H1 TO H2 STEP H3:GOSUB 400
45 NEXT H:H=H-Y
49 REM
50 REM VERTICAL LINE
55 FOR V=V1 TO V2 STEP V3:GOSUB 400
60 NEXT V:V=V-Y
64 REM
65 REM * SWAP PARAMETERS *
70 TH=H1:TV=V1:H1=H2:V1=V2:Y=-Y
75 H2=TH+X:H3=-H3:V2=TV+X:V3=-V3:X=-X
80 IF X>0 AND V2<V1 THEN 90
85 GOTO 40
90 GOSUB 520
95 GOTO 25
```

Lines 20–30 perform initialization. The character, K, is set to a slash, /. Variables X and Y are used to make small adjustments so the FOR. . .NEXT loops

position the lines properly. If you are curious about what they do, set them to 0 and run the program. The variables H and V are, of course, the familiar horizontal and vertical coordinates of the character. It is in lines 70 and 75 that interesting things take place. Here, after writing one horizontal line and one vertical line, we swap the start and end parameters of the FOR. . .NEXT loops, reversing the direction of painting on the screen. TH and TV are temporary storage locations for H1 and V1 during the swap process.

Line 80 detects the end of the spiraling process, in which case the program goes to COLOR KEYS and waits there for you to press a color key. When you press a color key, a new spiral is painted.

If your program doesn't work, LIST it in sections and compare it to the complete listing for SPIRAL shown in Appendix B.

One of the nice things about SPIRAL is the wide range of patterns you can paint just by making small changes to the program. Change the value assigned to K in line 25 from 78 to 64 and other values (refer to page 133 of the *User's Guide*). Also try changing V2 to V1 in line 55. Or how about Y=2 or y=5 in line 25?

Until now all our POKEing into memory has started in the upper left-hand corner and moved from left to right, top to bottom. For generating patterns that are symmetric about the center of the screen, a different approach works better. The next listing gives a short program that illustrates this very nicely. Enter NEW and the following:

```
10 REM *** SYMMETRY ***
20 PRINT CHR$(147)
25 C=5:K=160      :REM  GREEN SPACES
30 F(1)=1.5:F(2)=0:F(3)=-1.5:F(4)=0
35 G(1)=1:G(2)=0:G(3)=-1:G(4)=0
40 FOR V=0 TO 12 STEP 3
45 FOR N=1 TO 4:FOR M=1 TO 4
50 S=(V*F(N)+19)+(12-V*G(M))*40+1024
55 IF S<1024 OR S>2023 THEN 65
60 POKE S,K:POKE 54272+S,C
65 NEXT M:NEXT N:NEXT V
70 C=C+1:IF C>15 THEN C=0
75 GOTO 40
```

Check that minus signs have been properly entered in the program. When the program is run you will see a symmetric pattern of small green squares. When the pattern is complete the color changes and the pattern is redrawn. This is repeated until you press the RUN/STOP key.

Lines 3Ø and 35 set values into two small *arrays*, F(N) and G(M). These arrays are used in line 5Ø in the expression for the screen position S. An array is simply a way of referring to several related items, in this case numbers, by a single symbol, in this case F and G. A specific *item*, or *element*, of the array is specified by giving it a number and placing that number in parentheses after the symbol. This number is referred to as the *subscript* of the array. Arrays are powerful devices because the subscript can itself be a variable, as in line 5Ø, allowing calculation and manipulation in selecting array elements.

In line 3Ø, array element F(1) is assigned the value 1.5, F(2) is assigned Ø, and so on. Then, when N is 1 in line 5Ø, F(N) will have the value 1.5.

Lines 4Ø–65 contain three nested FOR. . .NEXT loops. The two inner loops vary N and M so that sucessive values of F and G, shown in lines 3Ø and 35, are selected.

Line 5Ø is a new expression for S that provides for symmetry about the center of the screen. Note first that H has been eliminated. You may think of H as having been replaced, or set equal to V, ensuring symmetry in the horizontal and vertical directions. By multiplying V by F(N) and G(M), points that are symmetric about the center are painted. The center of the pattern is set at the center of the screen by adding 19 for the horizontal and by subtracting vertical values from 12. Remember that counting for S starts at Ø, not 1. Therefore, 19 specifies the 2Øth, or center, horizontal position. Since we want increasing values of V to move up the screen, in the opposite direction from S, we must subtract V from 12. The result of all this is that H and V can now be used like X and Y axes with the origin at the center of the screen.

In line 7Ø we provided some automatic coloring. The color variable, C, is stepped through all 16 values and then cycled back to Ø.

Like SPIRAL, the SYMMETRY program lends itself to some interesting modifications. You can make changes to the values of F and G in lines 3Ø–35. In particular, try adding lines such as

```
32 F(5)=1.0:F(6)=1.8:F=−1.0:F(8)=−1.8
```

while extending the range of N in line 45 from 4 to 8.

In order to maintain symmetry, keep plus and minus values the same.

3

Light Show

In this chapter we will expand on the programs we have been working with so that you can create fascinating patterns of light.

GRAPHIC CHARACTERS

We will now add graphic characters and spacing capability to the COLOR KEYS program.

LOAD the subroutine COLOR KEYS. Then make the following changes and additions. You can LIST selected lines of COLOR KEYS and just change them to make similar lines. Don't forget to change line numbers also.

```
501 REM ** SUB.: KEY ENTRY **
525 K1=ASC(K$):IF K1<192 THEN 535
530 K=K1-128:RETURN   :REM GRAPHICS RT.
535 IF K1<161 THEN 545
540 K=K1-64:RETURN    :REM GRAPHICS LFT
585 IF K$="+" THEN W=W+1:GOTO 520
587 IF K$="-" THEN W=W-1:GOTO 520
590 IF W<0 THEN W=0
605 IF K$=CHR$(135) THEN 700 :REM F5
610 IF K$=CHR$(136) THEN 710 :REM F7
698 REM
699 REM * SELECT CORNERS OR CENTER *
```

```
700 F1=-F1:GOTO 520
708 REM
709 REM * SET SPACING *
710 S1=S1+1:IF S1>3 THEN S1=1
715 GOTO 520
```

SAVE the new program with the name KEY ENTRY. A complete listing of this subroutine is given in Appendix B so you can check your entries.

If you want to try out KEY ENTRY, an easy way is just to enter lines 1–95 of the COLOR CONTROL program given in Chapter 1. Better yet, use cut-and-paste techniques such as the following.

With KEY ENTRY safely SAVEd to tape or disk, LOAD COLOR CONTROL. LIST lines 1–95; with these lines displayed, LOAD KEY ENTRY. Now set the cursor on line 1 of COLOR CONTROL and hit RETURN for all lines from 1 to 95. COLOR CONTROL and KEY ENTRY are now united.

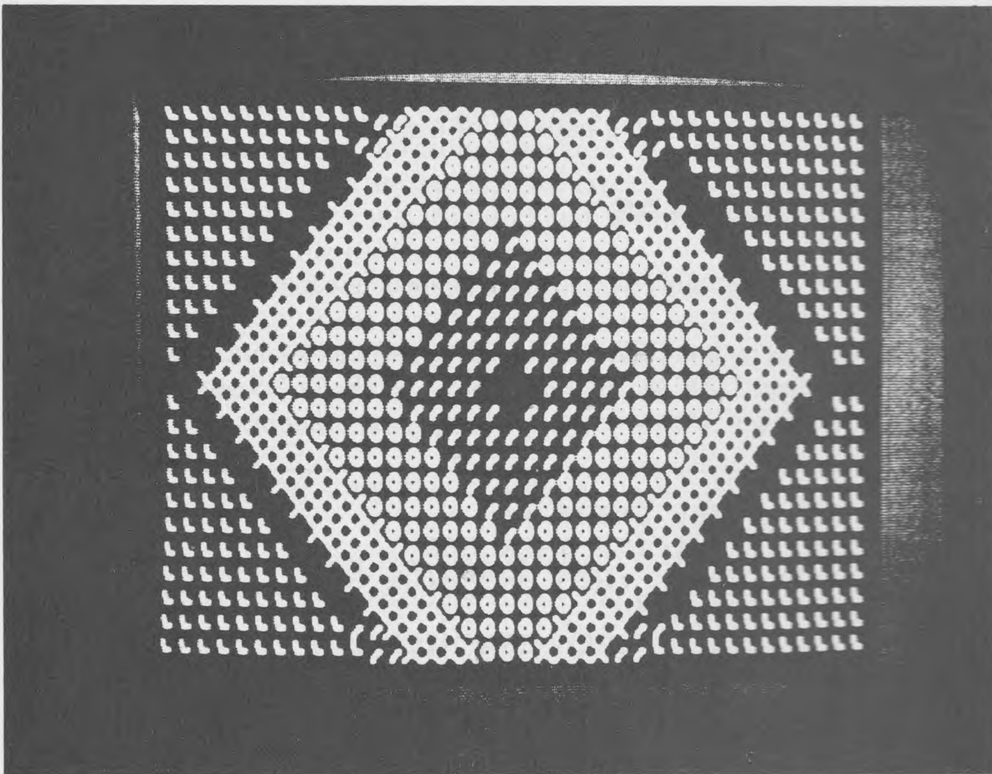


Photo 3 One of the designs that LIGHT SHOW created while operating automatically. LIGHT SHOW never runs out of new designs. You can create your own designs by selecting patterns and color directly from the keyboard.

RUN the program. Press a color key, and watch a bar of color being painted just as in the previous version of COLOR CONTROL. To select a graphic character, hold down either the SHIFT key or the Commodore key and press any key with a graphic character on the front. The SHIFT key selects the graphics character on the right side of the key; the Commodore key selects a character on the left side.

The f5, + and − keys, will be used later, but two simple additions will allow you to see how the f7 key works. Add STEP S1 to line 4Ø and a new line 32 as follows:

```
32 S1=1
40 FOR S=S TO S+C+10 STEP S1
```

When the program is RUN and you press f7 there is a space between each graphics character. Press f7 again and there are two spaces. Press f7 a third time and spacing returns to normal. As you will see, this spacing control is useful for achieving variations in designs. You can increase the number of spaces by changing the 3 in line 71Ø of KEY ENTRY to a larger number.

We will now put KEY ENTRY to better use with a much more interesting display. We will build up the program in four steps: a template, a triangle subroutine, a diamond subroutine, and an introduction screen.

TEMPLATES

A *template program* is used to build other programs. The template may work by itself as a complete program, or, as in the present case, it may require additional program lines before it can be RUN. Templates are the building blocks of cut-and-paste programming. Any group of program lines that is likely to be useful in creating new programs can be SAVEd for later use as a template.

This program's template will consist of KEY ENTRY, POSITION & COLOR, and a title. We will attack POSITION & COLOR to KEY ENTRY just as we did with COLOR KEYS.

LOAD subroutine POSITION & COLOR and LIST it. Now LOAD KEY ENTRY, but *don't* list it. Combine the two programs by setting the cursor to line 4ØØ of the displayed listing. Press RETURN repeatedly until all program lines are entered. You now have a template that can be used in building many other programs. If you want to SAVE it, you can use the following title line to indicate that it can be used for positioning and coloring:

```
1 REM *** TEMPLATE:POS COL ***
```

Now enter the title line for the program you are building, and change line 420 so an out-of-bounds condition for H will not STOP the program. This is not necessary for V:

```
1 REM ***** LIGHT SHOW *****
420 IF H<1 OR H>40 THEN H=20:W=1
```

For safekeeping, SAVE this program using the name LIGHT SHOW.

TRIANGLES

With LIGHT SHOW in memory, add the following lines. This will be easier than it looks if you use one line to create other lines, as described previously.

```
70 PRINT CHR$(147)          :REM  CLEAR
80 K=160:S1=1:W=6:F1=-1
95 REM
100 REM ***** MAIN LINE *****
101 REM
105 GOSUB 520
110 IF F1=1 THEN 180 :REM GO TO DIAMOND
115 H1=02:H3=+S1:HB=12:HE=2:HS=-1:V=2
120 GOSUB 230
125 H1=39:H3=-S1:HB=28:HE=38:HS=+1:V=2
130 GOSUB 230
135 H1=02:H3=+S1:HB=02:HE=12:HS=+1:V=14
140 GOSUB 230
145 H1=39:H3=-S1:HB=38:HE=28:HS=-1:V=14
150 GOSUB 230
170 GOTO 105
220 REM
225 REM ** DRAW TRIANGLE **
230 FOR H2=HB TO HE STEP HS
235 FOR H=H1 TO H2 STEP H3
240 GOSUB 420
245 NEXT H:V=V+1:NEXT H2
250 RETURN
```

It will be easier to understand the explanation given below if you RUN the program. First, SAVE the program with the name LIGHT SHOW and then

RUN it. Press a color key, and the program will paint triangles in each corner of the screen. All functions of the KEY ENTRY subroutine will operate except f5. Thus, you can select graphics characters with the SHIFT and Commodore keys, border and background colors with f1 and f2, and spacing with f7.

If you look closely, you will see that the right triangles are slightly larger horizontally than the left triangles. TV sets tend to squeeze objects on the right side relative to the left side. The LIGHT SHOW display compensates for this, as well as for the effect on symmetry of the even number of character positions. The result is the slight enlargement of the right side triangles.

If you have trouble with your program, check it against the listing for LIGHT SHOW given in Appendix B. This is a complete listing, so just concentrate on the lines that you have. Your program should work correctly before you go on to the next stage. If you have made any of your own changes in the program, you may want to go back and repeat the building process without the changes.

In line 8Ø, K and S1 are the now familiar character and spacing variables. The W and F1 will be explained in the next section. Line 1Ø5 calls the KEY ENTRY routine. Lines 115–15Ø set values in variables for use in the FOR . . . NEXT loops of the DRAW TRIANGLE subroutine, lines 225–25Ø. There are four such settings, one for each corner of the screen in which a triangle is drawn. The DRAW TRIANGLE subroutine contains two FOR . . . NEXT loops. The inner loop draws one horizontal line of the triangle. The outer loop controls the beginning, HB, and end, He, of each line. By incrementing V, the vertical position is stepped down one line between each outer loop.

DIAMONDS

In this section we will add a routine for drawing diamonds in the center of the screen. The program will draw an open diamond, a closed diamond, two adjacent diamonds, and other patterns.

With LIGHT SHOW LOADed add the following lines:

```
175 REM
180 CE=20:LB=5:LE=16:LS=1:V=2
185 GOSUB 280
190 CE=20:LB=15:LE=05:LS=-1
195 GOSUB 280
210 GOTO 105
215 REM
216 REM ***** SUBROUTINES *****
270 REM
275 REM ** DRAW DIAMOND **
280 FOR HL=LB TO LE STEP LS
```

```
285 FOR H=CE-HL TO CE-HL+W STEP S1
29Ø GOSUB 42Ø
295 NEXT H
3ØØ FOR H=CE+HL-W TO CE+HL STEP S1
3Ø5 GOSUB 42Ø
31Ø NEXT H
315 V=V+1:NEXT HL
32Ø RETURN
```

SAVE the new version of LIGHT SHOW and then RUN it. Press a color key to get four triangles, as before. Next, press f5 once and then another color key. An open diamond is painted between triangles. Press f5 again and then another color key. As you can see, f5 switches back and forth between the triangles and the diamond.

With f5, select the diamond. Press the + key about six times and then press a color key. The + key closes the diamond, that is, it makes the outside diamond thicker. The − key has the opposite effect. By starting with a thick diamond (use the key) and then pressing − key, color key, − key, color key, and so on, you can create patterns of nested diamonds.

You now know all the controls you need to create many fascinating designs. The most dazzling displays are those created by using the f3 and f7 keys, although all the keys do produce some very interesting combinations.

Now we can explain variables W and F1 in line 8Ø. The variable F1 is switched between +1 and −1 by f5, thereby selecting either the triangles or the diamond for painting in line 11Ø. The variable W in lines 285 and 3ØØ controls the width of the diamond. In lines 585 and 587 the + key adds 1 to W, whereas the − key subtracts 1 from W.

Lines 18Ø and 19Ø set variables for the diamond subroutine. The diamond subroutine consists of two consecutive FOR . . . NEXT loops nested in another FOR . . . NEXT loop. The two inner loops work from the center horizontal position defined by CE. The first inner loop paints the left side of the diamond; the second loops paints the right side. When called by line 185, the outer loop paints the upper half of the diamond; when called by line 195 it paints the lower half. As W increases, there is overlap of the two sides of the diamond, which creates new types of patterns such as double diamonds. The overlap explains the hesitations you will see in the painting of the diamond.

TITLE SCREEN

Until now your programs started with nothing but a blank screen. This is all right when you are experimenting and only using the programs yourself. The

LIGHT SHOW deserves better. To give LIGHT SHOW a professional look, add these lines:

```
20 PRINT CHR$(147) CHR$(5)
21 POKE 53272,21
22 PRINT "                   LIGHT SHOW"
25 POKE 53280,3:POKE 53281,2
30 PRINT:PRINT
35 PRINT SPC(9) "+=INCR. -=DECR. DIMND.
40 PRINT SPC(9) "F1=BORDR, F3=BACKGRND"
45 PRINT SPC(9) "F5=SWITCH CORNERS/CNTR
50 PRINT SPC(9) "F7=SPACE 0,1 OR 2"
55 PRINT:PRINT:PRINT
60 PRINT SPC(9)"PRESS ANY KEY TO START"
62 PRINT SPC(9)"THEN PRESS A COLOR KEY"
65 GET A$:IF A$="" THEN 65
```

AUTOMATING LIGHT SHOW

It's nice to be able to control LIGHT SHOW from the keyboard. But you will soon find that there is such a large variety of colors and patterns that you can't begin to run out of fascinating displays. Why not let the computer produce the displays for you? Then you can just let it run while you're reading, listening to music, having a party, or whatever.

Notice that in the following listing, lines 592 and 925 have a space between the quote marks. Here, we are testing for the entry of a space character, that is, the pressing of the space bar. To run in *automated*, or *auto, mode*, first RUN

```
520 IF ML=1 THEN 920
522 GET K$:IF K$="" THEN 520
592 IF K$<>" " THEN 595
593 ML=1-ML:K$="+":GOTO 920
900 REM
901 REM *** AUTOMATIC MODE ***
920 GET K$
925 IF K$=" " THEN ML=1-ML:GOTO 520
930 REM
935 REM * SET TIME DELAY *
940 IF K$="+" THEN TM=TM+900
945 IF K$="-" THEN TM=TM-900
950 IF TM<0 THEN TM=0
```

```
955 FOR J=0 TO TM:NEXT
960 REM
965 REM * GET RANDOM OPERATION *
970 J=INT(RND(0)*1000) :REM RANDOM NO.
972 IF J>255 THEN K$="+"
973 IF J>379 THEN K$="-"
975 IF J>503 THEN K$=CHR$(133)
977 IF J>627 THEN K$=CHR$(134)
979 IF J>751 THEN K$=CHR$(135)
985 IF J>875 THEN K$=CHR$(136)
990 IF J<256 THEN K$=CHR$(J)
995 GOTO 525
```

the program in the usual way. Then, whenever you want to switch to automatic, just press the space bar. The computer takes over and changes colors and designs automatically.

You have control over the intervals between changes in designs or colors. In automatic mode the + key can be used to increase the interval so that the computer will pause after it has painted the screen and before it starts the next cycle. You can press the + key repeatedly to keep increasing the pause time. Pressing the − key decreases the pause time in equal successive steps until it reaches zero.

The nice thing about auto mode is that you can let it run until you see something you like and then switch back to manual mode by pressing the space bar. Now you have full keyboard control again.

Line 52Ø is inserted before GETting the key entry in line 522. If the space bar is pressed, ML is toggled between 1 and Ø in line 925. If ML is 1, line 52Ø switches the program to the AUTOMATIC MODE routine, starting at line 92Ø.

Line 92Ø GETs any key entry that might have been made. Lines 925–945 test for entry of space bar, which will toggle ML out of auto mode, back to manual, and + or − key entries. If the latter, the counter in the FOR . . . NEXT loop in line 955, time delay variable, TM, is incremented or decremented, increasing or decreasing the presentation time of each pattern. These are the only key entries with which auto mode is concerned.

Lines 965–990 are the heart of this routine. The operation is based on *random numbers*. On each pass through the routine a random number is obtained in line 97Ø and stored in the variable J. A random number determines the operation performed on each pass.

In line 97Ø the BASIC function RND(X) is used to get the random number. If X=Ø, the system clock is used in generating the number, providing a number sequence with no repetitions. RND returns numbers between Ø and 1, so line 97Ø multiplies by 1ØØØ to get a number in the range we want and then INT con-

verts to an integer. We will thus have numbers randomly distributed from Ø to 999.

All numbers greater than 255 have no relevance as printing characters. Therefore, we use numbers 256–999 for selecting nonprinting operations such as those controlled by the function keys in lines 975–985; in lines 972–973 we use the + and – keys, as in manual mode, that is, to control the width of the diamond. These six operations have fairly equal distributions over the range 256–999. By changing this distribution in lines 972–985, you can control the relative frequency of each of these operations.

Through the function CHR$(J) in line 99Ø, the remaining values of J, Ø–999, are assigned the same function they have when operating in manual mode. K$ is set to this value, and program execution is returned to the line right after that which GETs the key entry K$ in manual mode. Now the program functions just as if a key had been pressed, since everything the program does after this point is controlled by the value in K$. Not all values between Ø and 255 result in an operation, which explains why there are periods when nothing happens on the screen for a few seconds.

If you have any trouble in getting your program to work as described, check it against the complete listing of LIGHT SHOW in Appendix B.

Table 1 summarizes the key assignments for LIGHT SHOW.

TABLE 1 KEY ASSIGNMENTS FOR LIGHT SHOW

COLOR KEYS	
Color	**Key**
Blacks	
Black	1
Dark gray	Q
Medium gray	A
Light gray	Z
Blues	
Dark blue	7
Light blue	U
Cyan	4
Greens	
Dark green	6
Light green	Y
Purple	5

TABLE 1 KEY ASSIGNMENTS FOR LIGHT SHOW (Cont.)

COLOR KEYS

Color	Key
Reds	
Dark red	3
Light red	E
White	2
Yellow	8
Cycle border	
Colors	f1
Cycle background	
Colors	f2

GRAPHICS

Key Symbol	Key
Front of key on right	SHIFT and symbol keys
Front of key on left	Commodore and symbol keys

CONTROL IN MANUAL MODE

Operation	Key
Increase width of diamond	+
Decrease width of diamond	−
Select corners or center	f5
Set spacing	f7
Switch to auto mode	Space bar

CONTROL IN AUTOMATIC MODE

Operation	Key
Increase pause between displays	+
Decrease pause between displays	−
Stop and switch to manual mode	Space bar

4

A New World of Sound and Music

Your computer can generate a great variety of sounds. These sounds can be similar to familiar sounds such as musical instruments, birds, and motors. Or they can be unlike anything you have ever heard before. We will explore this new world of sound and show you how to create computer sounds and computer music easily.

The Commodore 64 has electronic circuits designed specifically for producing sound effects. As a result, you will find that your computer can produce surprisingly accurate renderings of musical instruments. In addition, the 64 allows you to control the timbre and tone or sounds. With a little effort you can learn how to use the built-in capabilities for sound that reside in your Commodore.

In this chapter we will explain how you can use your computer to produce sound. In Chapter 5 we will show you how to turn your keyboard into a musical instrument and synthesizer. Let's begin.

Enter NEW and then the following program:

```
11 REM
12 REM ** SUB: TURN NOTE ON/OFF **
13 REM * NOTE OFF *
20 POKE M+4,WF
25 FOR Z=1 TO TP:NEXT
30 REM
```

```
35 REM * NOTE ON *
40 NT=INT(NT):IF NT>64814 THEN NT=1072
45 HF=INT(NT/256):LF=INT(NT-256*HF)
50 POKE M,LF:POKE M+1,HF
55 POKE M+4,WF+1
60 FOR Z=1 TO TN:NEXT
65 RETURN
```

This program can be used as a template for turning individual notes on and off. Therefore, SAVE it with the name TURN NOTE ON/OFF. Since this program will be used again in Chapter 5, be sure to jot down where you store it, especially if you are using tapes.

We will demonstrate the use of this template by incorporating it in a program that automatically plays all the notes on the music scale.

With the subroutine TURN NOTE ON/OFF LOADed in memory, add the following lines (don't enter NEW), all of which will be explained later.

> **NOTE: The symbol for raising a number to a power prints as a circumflex (^). Use the up-arrow key on your Commodore keyboard in place of the circumflex symbol.**

```
1 REM *** TEMPERED SCALE ***
2 REM PLAYS 8 OCTAVES, 12 NOTES EACH.
10 GOTO 820
800 REM *** MAIN LINE ***
801 REM
820 NC=268:TN=250:TP=50:NI=2^(1/12)
825 M=54272:WF=32:PRINT CHR$(147)
830 FOR J=M TO M+24:POKE J,0:NEXT
835 REM
840 REM * SET ADSR AND VOLUME *
845 POKE M+5,9:POKE M+6,0
850 POKE M+24,15
855 REM
860 REM * CALCULATE NOTES *
865 FOR N8=1 TO 8      :REM OCTAVE LOOP
870 NT=NC
875 PRINT:PRINT "OCTAVE=";N8
880 FOR J=1 TO 12      :REM NOTE LOOP
885 PRINT "NO.=";J,"PITCH=";NT*.061
```

```
890 GOSUB 20          :REM NOTE ON & OFF
895 NT=NT*NI:NEXT J   :REM NEXT NOTE
900 NC=2*NC:NEXT N8   :REM NEXT OCTAVE
905 END
```

Check your entries of N8 on lines 865, 875, and 9ØØ. It is easy to mistake the 8 for a B. The 8 refers to the fact that the variable, N8, controls the octave.

When this program is RUN, you will hear a series of notes starting at the low end of the scale at a pitch of 16 and moving up to the high end at a pitch of 3949. The program spans 8 octaves, with 12 notes per octave, for a total of 96 notes.

The program is set to produce the sound of a stringed instrument. After the first few octaves, the sound gives a pretty good approximation to a piano. Let's see what other instruments we can get from this program.

In line 845, change the Ø in the second POKE statement to 15. The result sounds like the plucking of a bass viol. To produce a bowed instrument, enter values of 85–95 in place of 9 in the first POKE statement (line 845). Experiment with other values. Any number between Ø and 255 can be entered. You might find it advantageous to reduce the number of notes; in line 865 enter 4 TO 6 in place of 1 TO 8, and in line 82Ø, enter 2145 in place of 268. Each time you increase the starting point of N8 one octave, you must double NC in line 82Ø.

The values of TN and TP in line 82Ø also alter the character of a sound. These numbers determine the timing of various parts of the sound pattern, as will be explained below.

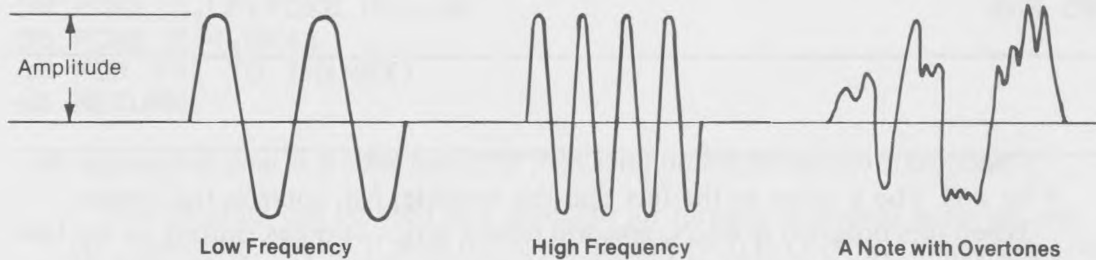
We won't be using lines 8ØØ–9Ø5 again so whether you save them or not is up to you.

WHAT IS SOUND?

Sound is transmitted as vibrations. In the air these vibrations produce sound waves. In the electric circuits in your computer sound vibrations take the form of rapidly varying voltages. But whatever the medium, sound has three characteristics that determine the overall quality of the sound—frequency, overtones, and amplitude envelope. Each of these is illustrated in Figure 2.

Frequency

Frequency is the rate of vibration. If the vibration is thought of as a wave with peaks and valleys, the number of peaks (or valleys) that passes a point in 1 sec-



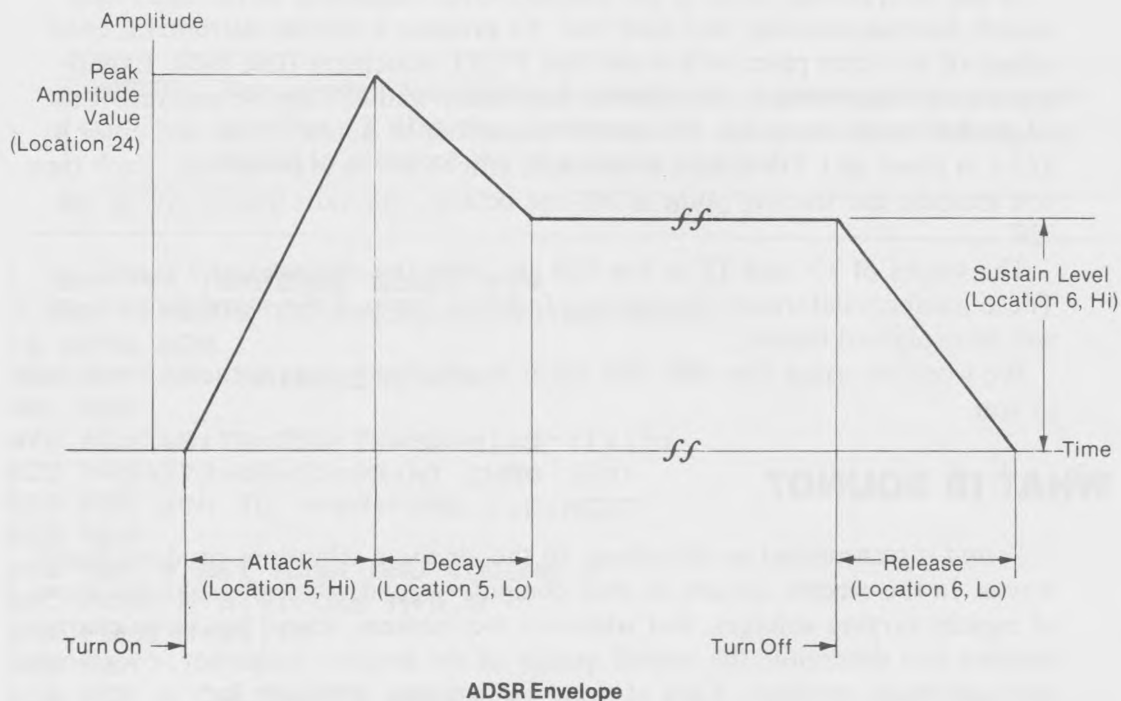


Figure 2. Components of Sound.

ond is a measure of the frequency in cycles per second. A sound wave of high frequency produces a high pitch sound; a sound wave of low frequency produces a low pitch sound. A *musical note* is a name given to a specific frequency. For example, the name of the note that has a frequency of 523 cycles per second is middle C. The human ear can hear frequencies from 2Ø to 2Ø,ØØØ cycles per second. The Commodore 64 can generate frequencies from 16 to 38Ø5 cycles per second, as shown in the TEMPERED SCALE program. Since the ear is more sensitive to frequencies near 3ØØØ, you probably noticed that the program seemed to get progressively louder, although this was only a *psychoacoustic* effect because the volume setting was constant during the program.

Overtones

Although two instruments can create sound vibrations of the same frequency, they sound different. This is partly due to the fact that they produce different *overtones.* In our discussion so far we have assumed that sounds contain only one frequency. In fact, they usually contain a number of closely related frequencies composed of a fundamental frequency and higher tones. The *fundamental* is the lowest frequency in a sound. The *higher frequencies* are whole number multiples of the fundamental. For example, if the note middle C is played, the fundamental frequency will be 523 cycles per second. The first overtone will be 2 × 523, or 1046 cycles per second. The next overtone will be 3 × 256, and so on.

The relative amplitude of the overtones determines the timber of an instrument; and it is *timber* that makes a note, played on a piano sound different from the same note played on a clarinet. The overtone amplitudes vary widely and may be zero even for frequencies close to the fundamental. At much higher frequencies, all overtones will vanish. When a violin, piano, or drum is played, very complex sound patterns are produced. But no matter now complex, it is an amazing fact that the sound vibrations are always the sum of simple vibrations: the fundamental plus all its overtones. We will refer to the totality of the overtones as the *overtone pattern.*

Amplitude Envelope

Another quality that determines sound is the *amplitude envelope.* This aspect of sound is probably less familiar than the previous two qualities, although it is of fundamental importance. The piano, guitar, and violin are all stringed instruments yet they sound quite different. This is largely due to their different amplitude envelopes.

Figure 2 shows a typical amplitude envelope. As you can see, the amplitude envelope is simply a plot of the amplitude of a sound over time. Typically, a sound rises quickly to its peak value. The rate of this rise is called the *attack.*

The sound does not necessarily stay at its *peak amplitude value* but may drop off to what is called the *sustain level.* The rate of this drop off is called the *decay.* The sustain level is maintained for a time and then stopped; in electronic music it is turned off. When the sound is terminated, the amplitude does not necessarily drop to zero immediately; it may taper off as shown in the figure. The rate of this fall is called the *release.*

It is interesting to apply these ideas to familiar instruments. The piano, for example, will have a fast attack and no sustain or release. The sound is mostly decay from the level reached by the attack phase. The organ also has a fast attack, but it has a high sustain level with little decay. The accordian is mostly attack and decay with no sustain or release.

We will follow the usual practice of referring to the four phases of the amplitude envelope taken together as ADSR for attack, decay, sustain, and release.

THE TEMPERED SCALE

Sound can vary in *frequency* (or *pitch*) through the audio spectrum. From all possible frequencies, musicians have selected specific frequencies called *notes.* By experimenting with progressions and combinations of frequencies, musicians picked those that appealed to them the most. One such combination that blends especially well is when the note frequencies are in the ratio of 2 to 1. This is called an *octave.* Another combination of notes that sounds well is the *perfect fifth,* which has a ratio of 3 to 2. A scale made up of notes having harmonious combinations, such as these, is called the *natural,* or *pure diatonic, scale.*

Because the diatonic scale didn't sound right when the keynote was shifted from, for example, C to D, a different scale, called the *equally tempered scale,* was devised. This scale is a compromise between the advantages offered by the diatonic and the desire of composers for more flexibility. In the equally tempered scale, the octave between notes with a 2 to 1 frequency ratio is equally divided into 12 intervals. Therefore, each note is related to the next higher in the scale by multiplying it by the 1/12th root of 2, which is 1.Ø5946309.

All this can be used to advantage in computer music where multiplying by intimidating numbers, such as the above, is no problem at all. We can generate all notes in the scale starting at a C note of 16 cycles per second and multiplying by the square root of 12. There is one problem, however, with respect to speed. Low-cost computers running under BASIC do not operate very fast; therefore, we will be forced to use other methods later, although the method works for the above program.

Scales other than the equally tempered scale are used in various parts of the world and by various musicians in the West. It is generally agreed that *harmony* and *dissonance* are determined in part by what we are accustomed to hearing. Even in Western music, chords that were considered dissonant in the past are

considered harmonious today. This is something to remember as you experiment with computer music. See what new combinations of sounds can do. If you follow the explanation below, you can easily create your own scales. However, all the music programs in this book will use the equally tempered scale.

HOW YOUR COMPUTER MAKES SOUNDS

From the above discussion you now know what the computer has to do to create a particular kind of sound. It must be able to create the fundamental frequency, the pattern of overtones, and the appropriate ADSR envelope. As you have already seen with the TEMPERED SCALE program, your Commodore 64 can produce a wide range of fundamental frequencies, that is, all 12 notes in each of 8 octaves.

You also saw that your Commodore can control the ADSR values when you made changes to lines 82Ø and 845. However, we have only scratched the surface of what can be done here and will return to the control of ADSR values shortly.

The generation of the overtone pattern is the most difficult of the sound qualities to produce in a computer. The flat sound of much computer music is the result of insufficient overtones. Commodore has used an indirect approach. You are provided with three waveforms and three filters. By combining waveforms and filters you can generate a great variety of overtone patterns, often coming surprisingly close to the real thing. We will describe how to do this below.

Another feature of Commodore's sound equipment is a *white noise generator*. White noise is a heterogeneous mixture of frequencies that can be manipulated in a variety of ways. You can use it in games for sound effects such as engine blast and escaping gas. Noise is one of the most fascinating types of sound because of the many surprising effects that can be produced with it.

Sound Memory

Just as the Commodore reserves memory locations for screen characters and colors, it reserves memory locations for sound and music. There are 29 consecutive locations, starting at location 54272. Although we will not use all of these locations in this book, we will use those that control the shaping of sound. These will be explained as we go along.

As a technical aside, none of which is needed for the following discussion, the sound memory locations are neither random access memory (RAM) nor read-only memory (ROM) but are 8-bit registers located on an LSI chip, the 6581 sound interface device that Commodore refers to as the SID chip. In the case of frequency these 8-byte registers are combined into a 16-bit unit; in the case of ADSR values, they are split into 2 nibbles.

The Tempered Scale Program

With the above as background, we can describe how the TEMPERED SCALE program works. We will go through that program line by line and then move on to more ambitious sound making. If you're not interested in these details, you can skip this section. You don't have to know how these programs work to use them.

The program consists of a main routine and the subroutine TURN NOTE ON/OFF. When using cut-and-paste techniques, it is advantageous to put the main program at the beginning and tack on subroutines as needed. However, in the case of music and graphics programs, speed becomes a problem. Later when you enter notes from the keyboard, the program must be able to keep up with your fingers. When BASIC calls a subroutine, it looks for it starting at the lowest numbered program line. Therefore, all subroutines, which critically impact program time, must be placed at the beginning of the program. TURN NOTE ON/OFF is placed at the very beginning. Then the *main line routine* is placed way up in the hundreds to leave plenty of room for other subroutines below it.

Lines 820–830 simply initialize things for later developments. Nevertheless, they contain some important features. In line 820, NC stands for note C. Musicians, for some reason, don't start from A, but from C. Each octave starts with note C so that for our program note C is the reference note from which we calculate all others. In line 820 we initialize it to the lowest value, 268. The 268 is a Commodore computer number used to compute frequency; it is not the frequency itself, which is 0.061 times 268, or 16.348.

In line 820 TN is the time of the note—from the time when the attack starts to when the release starts (see Figure 4-1); TP is a program delay time for the release, not the release time itself. This will be made clear later as will how to select what numbers to assign to TN and TP. In line 820 they are given values of 250 and 50, respectively.

In line 820 NI stands for note interval, that is, the factor by which each note is incremented to get the next higher frequency. In the well tempered scale each note is related to the previous by the factor 12th root of 2, written as 2^(1/12). The circumflex, ^, is the exponent sign (the up arrow key on your keyboard), and a number raised to a power of 1/12 is the same as the 12th root.

Line 825 sets M to 54272, the start of sound memory. We use M for sound memory because in previous chapters we have already used S for screen memory and don't want any conflicts creeping into our programs. Let's agree to call it music memory and remember it that way. Since all locations in music memory are consecutive locations, we will use M as the starting point, or base, and refer other locations to it. Thus, music memory location 2 is at M + 2, assuming M is music memory location 0.

In line 825, CHR$(147) clears the screen. Line 830 is a loop that POKEs 0

into all relevant music memory locations, initializing them to zero. The music locations serve several functions. If you don't clear them to zero, results may be unpredictable.

In lines 845 and 85Ø we start POKEing values into music memory. In line 845, 9 goes into the fifth location and Ø into the sixth. The fifth location controls attack and decay. The sixth location controls the sustain level and release. The values for attack, decay, and release are *time* values; the sustain value, on the other hand, sets the relative amplitude *level*—from the peak amplitude down to zero in 15 equal steps (16 levels).

To get two items into one location requires some arithmetic. Here are the formulas:

POKEd value in location 5 = A * 16 + D
POKEd value in location 6 = S * 16 + R

where

A = attack
D = decay
S = sustain
R = release

All range from Ø to 15. Line 85Ø POKEs 15 into the amplitude location, setting the peak level of the ADSR envelope to the maximum possible value.

Now all preliminaries are taken care of. Line 865 starts things rolling with a FOR. . .NEXT loop that controls the octave. The symbol N8 is used to designate a note's octave. In line 87Ø the note value NT is set equal to NC (note C), which was initialized to 268 in line 82Ø.

Lines 88Ø–895 are an inner FOR. . .NEXT loop that calls a subroutine to turn the note on and off and then calculates the next note value in line 895.

After 12 notes have been played, the loop falls through to the outer loop at line 9ØØ. Line 9ØØ calculates a new value at which note C will start the next octave. We could have calculated all 96 notes using the inner loop, but line 9ØØ gives more accurate results.

Lines 11–65 are our template subroutine for turning notes on and off. You may wonder why the note is turned off before it is turned on. This is done to give us control over the timing of the note. If the note were turned on first, all timing control would have to reside in the subroutine. In this program that's okay, but later we want to control this externally so we turn on the note, exit the subroutine, come back to turn it off, and turn on the *next* note. This way you will be able to control the timing of the note from the keyboard.

Lines 4Ø–5Ø put the note value, NT, into locations Ø and 1 of music memory.

The note value is split into two parts, HF and LF, one of which goes in location Ø and the other in location 1 to form a 16-bit number for frequency. You don't have to understand how this is done to use the program.

Although the note's value has been entered, the note must be turned on to produce a sound. This is done in line 55 by POKEing WF+1 into location 4. Location 4 does more than just turn notes on and off, so the value POKEd there must be chosen with care. In our present program, WF has been set to 32 in line 825. This takes care of everything in location 4 but the on/off switch for the voice. The latter is switched on by the added 1. As soon as 33 is POKEd, the attack phase of the ADSR envelopes starts.

Now attack and decay proceed automatically since we have already set values for A and D. When the sustain phase is reached it will continue until the note is turned off. Line 6Ø is a delay loop that keeps the program looping at line 6Ø for as long as we want sustain to last. This is controlled by time of note, TN, which we initialized to 25Ø. Later, for purposes of external timing control, we will remove line 6Ø.

When the delay loop ends it is time to exit the subroutine. The next time the subroutine is called the note will be turned off in line 2Ø. This is done by POKEing 32 into the same location that we used to turn on the note with 33. Now the sustain phase ends and the release phase starts. Although the time of the release phase was set in line 845, we still have to delay the program with the FOR. . .NEXT loop in line 25. If we did not have this delay, the program would go ahead and turn on the next note before the end of the release phase. When a note is turned on, any ADSR in process is zapped and the new note starts. Therefore, select time of program delay, TP, in line 82Ø to allow release to reach zero before starting a new note.

Table 2 gives the values to POKE in memory to get various attack, decay, and release times. Table 3 gives the TN values you can use for creating notes of various durations.

TABLE 2 ATTACK, DECAY, AND RELEASE TIMES

Value to POKE in Memory	Attack Time (in seconds)	Decay and Release Times (in seconds)
0	.002	.006
1	.008	.024
2	.016	.048
3	.024	.072
4	.038	.114
5	.056	.168
6	.068	.204
7	.080	.240
8	.100	.300
9	.250	.750
10	.500	1.5
11	.800	2.4
12	1	3
13	3	9
14	5	15
15	8	24

TABLE 3 TIME OF SOUND OR NOTE, TN

	Note Duration	Program Entry, TN
	1/16	128
	1/8	256
Dotted	1/8	384
	1/4	512
	1/4 + 1/16	640
Dotted	1/4	768
	1/2	1024
	1/2 + 1/16	1152
	1/2 + 1/8	1280
Dotted	1/2	1536
Whole		2048

5

Musical Keyboard

Is your Commodore 64 connected to your audio system? It should be because, although the sound programs will work perfectly well with just your television set, you won't be able to get the kind of reproduction that your sophisticated computer sound system deserves. Pages 6 and 7 of your *User's Guide* tell you how to connect to an audio system.

In Chapter 4 we used a simple program to explain and demonstrate some of the features of the Commodore 64. Now, to make computer music and sounds without having to enter numbers into a program, we will assign the various functions to the keys on your keyboard and let the computer do the work of translating your keystrokes into sounds and music.

When we did a similar thing with color, we had a difficult time finding a good, easy-to-use key assignment. The situation is much worse for sound. There are 96 notes on the scale, plus 24 control locations in memory, plus 2–8 things some of these registers control. With only 63 usable keys on the keyboard, something has to give. We could use the SHIFT key to extend the number of key assignments, but I don't think you relish searching for the SHIFT key with your little finger while playing Bach or rock. So here is one scheme; you will be able to use it as is and change to suit any special preferences you have.

The keys on the third row from the top will be used to play notes A, B, C, D, E, F, and G. There are 14 usable keys on that row, providing two octaves. Not all keys are usable by the program. SHIFT, for example, does not communicate with a program. On line 3, all keys except SHIFT LOCK are usable. To make the RUN/STOP key usable, the program must be operated with the SHIFT LOCK key in the depressed state, not a desirable feature, but acceptable. Line

2 can be used for sharps and flats in positions approximating those on a piano keyboard. This is the primary reason for adopting this arrangement—familiarity to many users.

My musician friends tell me that two octaves are not enough. Therefore, we will use the space bar to shift from one pair of octaves to another. The octaves just above and just below middle C are available before the space bar is pressed. When the space bar is pressed, your left hand will receive the next lower octave, and your right hand will receive the next higher octave. Remember, space bar is the only shift you will be asked to make. Also, take into account that we will be adding a playback mode so that you can separate the entering of notes from the playing of notes.

The bottom line of keys will be used to control timbre and tone. The four function keys on the right of the keyboard will be used for controlling attack, decay, sustain, and release. Most of the keys on the top row will remain unassigned.

Table 4, at the end of this chapter, shows the key assignments. Although there is no correlation of functions with the keys' capital letters or symbols, the keys have been grouped by function so that it is easier to remember what does what. You can also attach small pieces of masking tape to the keys and write each key's function on the tape.

The first program uses keys with key top letters that correspond to the notes generated.

MUSICAL KEYS

Enter NEW and the following lines:

```
5000 REM
5001 REM **** START PROGRAM ****
5010 REM
5011 REM * INITIALIZE SOUND *
5020 M=54272:AM=15:N8=1:PRINT CHR$(147)
5030 WF=32
5040 FOR J=M TO M+24:POKE J,0:NEXT
5050 POKE M+5,9:POKE M+6,0:POKE M+24,15
```

You can SAVE these lines and use them as a template for initializing your own sound programs. Next, add the following lines to the above. Remember, you can enter them as listed below and BASIC will merge them with the previous lines in the right order.

```
1 REM  **** MUSICAL KEYS ****
2 REM  HUNT AND PECK KEY ENTRY
10 GOTO 5020  :REM BY-PASS SUBROUTINES
800 REM
801 REM ******************************
802 REM *** SUB. SOUND KEY ENTRY ***
820 GET K$:IF K$="" THEN 820
830 IF K$="A" THEN NT=3608
832 IF K$="B" THEN NT=4050
834 IF K$="C" THEN NT=4291
836 IF K$="D" THEN NT=4817
838 IF K$="E" THEN NT=5407
840 IF K$="F" THEN NT=5728
850 IF K$="G" THEN NT=6430
860 RETURN
5100 REM
5101 REM *** MAIN LINE ***
5120 GOSUB 820
5130 GOSUB 20
5140 GOTO 5120
```

The next thing to do is combine these lines with the subroutine TURN NOTE ON/OFF that you wrote in Chapter 4. A good cut-and-paste technique is to SAVE a large listing in a temporary file called TEMP and then tack it onto a listing that can fit on the screen. First SAVE the above lines in TEMP; then LOAD TURN NOTE ON/OFF, and LIST it. Now LOAD TEMP, but don't LIST it. Run the cursor up to line 11 on the screen. Press RETURN repeatedly to enter lines 11–65. Now you have a complete program. First SAVE it and then RUN it.

Is the volume on your audio system or TV turned up? If so, you will be able to play the notes A, B, C, D, E, F, and G when you press the corresponding key.

Notice how simple the main line program is—two GOSUBs and a GOTO. The real work is done in subroutines. This kind of structure makes it easy to cut and paste programs. Program modules are added and then linked together in the main line program.

The only new thing in this program is in lines 820–850, which assign numbers to the note variable, NT, depending on which key is pressed. These numbers are proportional to the corresponding frequencies of the note and are the values your computer uses to generate these frequencies. A full listing of this program is given in Appendix B.

We are going to do some major surgery on this program to make it conform

to the keyboard layout shown in Table 4. If you also want to keep it as is, SAVE it as MUSICAL KEYS. You can use it as a template for sound programs that use key capital symbols. We will now modify the program and change its name to MUSICAL KEYBOARD.

MUSICAL KEYBOARD PROGRAM

With MUSICAL KEYS LOADed, enter the following lines, paying careful attention to line numbers so that later additions will merge into their proper slots. If you have forgotten the easy way of using line templates to enter repeated lines with slight variations, now is the time to refresh your memory because there will be lots of opportunity for this. (See Chapters 1–4.) The REM lines below are dummies that will be overwritten later.

```
1 REM **** MUSICAL KEYBOARD ****
2 REM  SOUND AND MUSIC SYNTHESIZER
821 A=ASC(K$)
830 REM
832 IF A=061 THEN NT=7217:RETURN
834 IF A=091 THEN NT=5728:RETURN
836 IF A=093 THEN NT=6430:RETURN
838 REM
840 IF A=141 THEN NT=8101:RETURN
850 REM
860 REM
862 IF A=202 THEN NT=4291:RETURN
864 IF A=203 THEN NT=4817:RETURN
866 IF A=204 THEN NT=5407:RETURN
868 RETURN
```

SAVE the program by the name MUSICAL KEYBOARD. As MUSICAL KEYBOARD is being built up, you'll SAVE it several times before it is complete. This will be no problem with disks. If you are using tapes, however, you must be careful to keep track of the location of the latest version. The simplest approach is to use a new tape, rewind to the beginning, and overwrite previous versions with the latest version.

With MUSICAL KEYBOARD SAVEd, enter RUN and press the SHIFT LOCK key. Keys J, K, L, :, ;, =, and RETURN will play the following notes starting at middle C: C, D, E, F, G, A, and B.

If the program doesn't work, fix it before adding more lines. From now on, whenever you add lines to the program, be sure they are working correctly be-

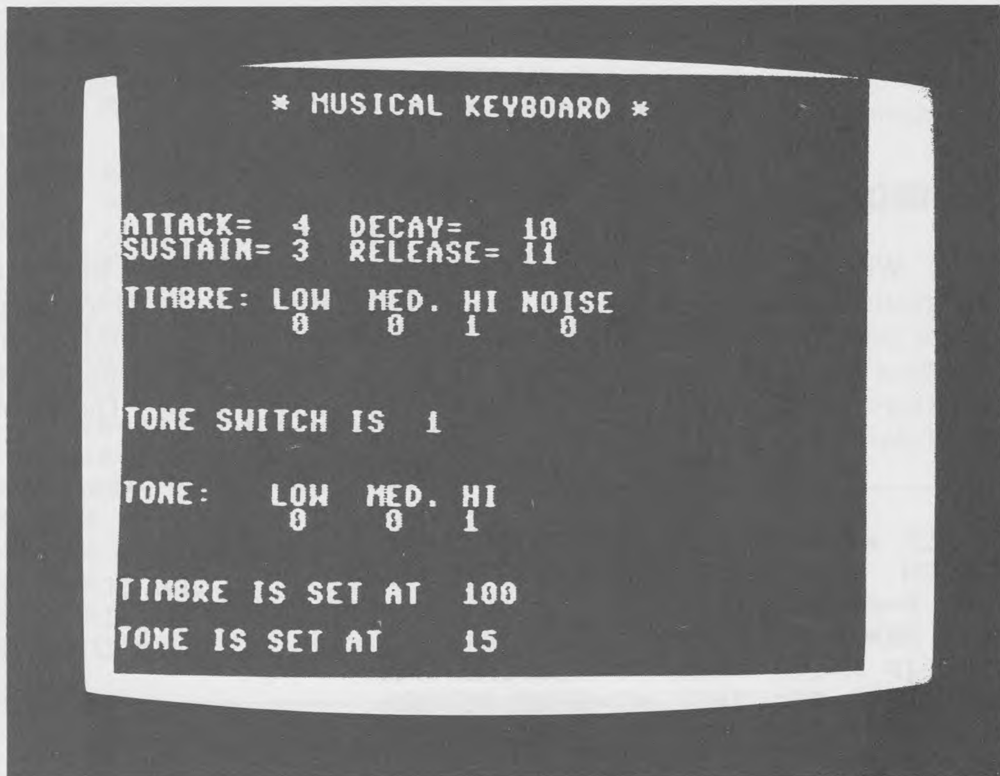


Photo 4 MUSICAL KEYBOARD turns your keyboard into a musical instrument and sound synthesizer. This illustration shows the display of those sound qualities that you control by pressing keys. Four octaves including sharps and flats are also provided.

fore proceeding to the next step. Check that the volume control is turned up. A full listing of the final program is given in Appendix B. Check the lines entered up to this point. Review Quick Chart 1 at the end of Chapter 1 for how to find typing errors.

> **NOTE: To stop the program, you have to press the SHIFT LOCK key in order to release the shift mode before pressing RUN/STOP.**

The new lines you have just added need some explanation. In this program we have to replace the easy to understand key entry routines we have been using with the lines presented above. In return for a loss of ease we gain some much needed speed and a way of avoiding the printing of upper case graphics.

In line 821, ASC(K$) gives a numeric value corresponding to whatever key you press. As we discussed in Chapter 1, each key has such a value; it's called

its ASCII number, after the standard used by manufacturers to encode characters in computer systems. By entering this value in A, we have a more concise and flexible representation than we did before. Appendix F in the *Commodore 64 User's Guide* lists ASCII codes.

More Notes

In this section we will LIST all 24 notes, covering two full octaves. You don't have to enter these right now. In fact, you will find the program more manageable if you wait until all the other features are entered. However, I know some of you are anxious to start playing, and it is quite easy to generate the following lines using lines already on the screen as templates:

```
830 IF A>186 THEN 844
838 IF A=131 THEN NT=2145:RETURN
842 IF A=186 THEN NT=6069:RETURN
844 IF A>200 THEN 858
846 IF A=192 THEN NT=6812:RETURN
848 IF A=193 THEN NT=2408:RETURN
850 IF A=196 THEN NT=2864:RETURN
852 IF A=198 THEN NT=3215:RETURN
854 IF A=199 THEN NT=3608:RETURN
856 IF A=200 THEN NT=4050:RETURN
858 IF A>209 THEN 872
860 IF A=201 THEN NT=4547:RETURN
868 IF A=207 THEN NT=5103:RETURN
870 IF A=209 THEN NT=2273:RETURN
872 IF A=210 THEN NT=3034:RETURN
874 IF A=211 THEN NT=2703:RETURN
876 IF A=212 THEN NT=3406:RETURN
878 IF A=215 THEN NT=2551:RETURN
880 IF A=217 THEN NT=3823:RETURN
882 IF A=222 THEN NT=7647:RETURN
990 GOTO 820
```

Lines 830, 844, and 858 speed up the search through all those IF statements for the note key that is pressed. The other keys are not as time-dependent as the note keys.

NOTE: Be sure to press SHIFT LOCK *after* entering RUN.

SAVEing Your Work

You now have invested some time in creating a program. It would be a shame to lose it all. At this point it might be a good idea to review what was said in Chapter 1 about SAVEing your work.

More Octaves

The simple mathematical relationship between notes of the well tempered scale make it easy to extend the two octaves that are now in your program. With MUSICAL KEYBOARD LOADed, add these lines:

```
37 IF NT>4050 THEN NT=NT*N8:GOTO 40
38 NT=NT/N8
890 IF A=160 THEN N8=2/N8:GOTO 820
```

Make sure you have entered N8, not NB, in line 890.

Now, when you press the space bar, the left keys go down an octave and the right keys go up an octave. Press the space bar again and the keys revert to the original two octaves. Thus, you can toggle back and forth using either thumb to do the switching.

Line 890 does the switching. In line 5020, NB is initialized to 1. Therefore, the first time the space bar is pressed, N8 is switched to 2. The next time the space bar is pressed, N8 becomes 1 again. Lines 37–38 double or half the note values, depending on their position in the scale.

Shaping the Sound

Now we will shape the amplitude envelope of sound using the ADSR values—attack, decay, sustain, and release.

With MUSICAL KEYBOARD LOADed enter the following lines, taking special care in lines 1247–1255 to get the proper punctuation and spacing:

```
900 IF A>136 AND A<141 THEN 1210
1200 REM
1201 REM * ADSR FROM FUNCTION KEYS *
1210 IF A=137 THEN AT=FNE(AT)
1215 IF A=138 THEN DE=FNE(DE)
1220 IF A=139 THEN SU=FNE(SU)
1225 IF A=140 THEN RE=FNE(RE)
1230 AD=AT*16+DE:SR=SU*16+RE
```

```
1235 POKE M+5,AD:POKE M+6,SR
1240 PRINT CHR$(19)
1245 FOR J=1 TO 5:PRINT:NEXT
1247 PRINT "                                 "
1248 PRINT "                                 "
1249 PRINT CHR$(145) CHR$(145);
1250 PRINT "ATTACK= " AT " DECAY=  ";DE
1255 PRINT "SUSTAIN=" SU " RELEASE=";RE
1260 GOTO 820
5035 DEF FN E(X)=X+1-(X+1)*INT(X/15)
```

With these additions, enter RUN and press SHIFT LOCK. All the notes will sound the same as before. But if you press function key f3, there is a change. Your screen displays

```
ATTACK=  Ø DECAY=    1
SUSTAIN= Ø RELEASE= Ø
```

Now when you press a note key, you get only a faint click. Alternate pressing f3 and a note key until the decay value is 9. The note keys now sound the same as before. The ADSR values are now the same as those we have been POKEing into locations 5 and 6 in line 5Ø5Ø. Continue increasing decay. When you pass 15, the value falls back to zero and recycles. All ADSR variables have values that range from Ø to 15. These are translated by the computer into times or, in the case of sustain, into an amplitude level. See Table 2 and Figure 2 for how this works.

With all values set to Ø press the f1 key, and listen to the effect the attack values have on the notes. Notice the amplitude rise to a peak and suddenly cut off because the ADSR values are all Ø.

The four function keys work in combination as well as singly. You can experiment with all four now, but we will not discuss the sustain and release keys (f5 and f7) until after entering the next program lines.

The task of cycling through ADSR values of Ø–15 is performed by the function defined in line 5Ø35. DEF FN in BASIC means define the following function, in this case E(X). For any number less than 1, INT returns a Ø. Therefore, the second term in E(x) will be Ø as long as X is less than 15. When X equals 15, the second term becomes X+1 so that the next E(X) will evaluate to Ø.

In lines 121Ø–1225, either AT, DE, SU, or RE is substituted for X in E(X), thereby incrementing attack, decay, sustain, or release, respectively, or resetting the value to Ø. Line 123Ø combines AT and DE for storage in location 5 and SU and RE for storage in location 6. This is the line to use as a template in your own programs when you want to set ADSR values.

Lines 1247 and 1248 clear the screen fields before entry of new, and possibly smaller, values, moving the cursor down one line after each PRINT. Each CHR$(145) in line 1249 moves the cursor up one line, back up to the first line of the ADSR display. Even though the cursor is not displayed, it has a location for PRINTing purposes. After PRINTing the new values, the program loops back to line 820 to get the next key entry.

Turning Notes Off

Presently, the program only turns off one note when another note is turned on. Now we will provide keys for turning off a note at any point you wish. With MUSICAL KEYBOARD LOADed, enter the line number

```
990
```

and press RETURN. This will delete an unwanted line. Then enter

```
1090 REM
1091 REM * TURN NOTE OFF *
1092 IF A<>38 AND A<>39 THEN 1098
1094 POKE M+4,WF
1096 GOTO 820
1098 REM * DUMMY BRANCH POINT *
1190 REM
1191 GOTO 820      :REM INVALID KEY
```

At this time you might want to enter the following lines to remind you to press SHIFT/LOCK and to give your program a title; they are optional but desirable.

```
5078 REM
5079 REM * START SCREEN AND TITLE *
5080 PRINT:PRINT:PRINT:PRINT
5082 PRINT " *** PRESS SHIFT LOCK ***"
5084 PRINT "  THEN THE J KEY TO START"
5086 GOSUB 820
5088 PRINT CHR$(147)
5090 PRINT SPC(8)"* MUSICAL KEYBOARD *"
```

Line 5Ø86 forces the pressing of SHIFT/LOCK, since the subroutine at line 82Ø will accept only shifted key codes.

With these additions let's return to the sound shaping variables ADSR. RUN the program, play a note, and press f5. Sustain is given a value of 1, and the display on your screen will confirm this. If you play a note again, the note stays on at a low level. Each time you press f5 the amplitude increases until it recycles to no sustain at level Ø.

Sound a note with sustain at any level other than Ø. The note remains on. Now press key 6 or 7. The sound immediately stops. Two keys are provided in the top row center to make it easier for you to find a turn off key while playing. You now have two ways of stopping a note: Either play a new note or hit the 6 or 7 key.

Set the sustain level at about 7. Turn a note on and off, each time pressing the release function key (f7) once. At first, there is no noticeable change, but gradually you will detect a trailing off of the sound. At about 8 it becomes quite noticeable; further increases will make it last several seconds.

The program is set for stopping a note, including release, as soon as a new note is played. This allows the notes to keep up with your keystrokes. To get the effect of a nonzero release time, set TP in line 25 to nonzero values. Try values 1ØØ–5ØØ.

You now have the tools to shape the amplitude of sound. The next step is to set the timbre.

Setting the Timbre of Sound

We will now use the bottom row of keys, proceeding from left to right, to control the sound. The four keys, Z, X, C, and V, in conjunction with the CRSR up/down key, control timbre. The cursor keys have the special ability of automatically repeating when held down, so we use them to sweep through a range of values.

With MUSICAL KEYBOARD LOADed enter the following:

```
1110 REM
1111 REM * WAVEFORM SELECTION *
1114 IF A=214 OR A=195 THEN 2010
1116 IF A=216 OR A=218 THEN 2010
2000 REM
2001 REM * WAVEFORM COMPOSITION *
2010 IF A<>214 THEN 2022
```

```
2015 WT=0:WS=0:WP=0:WN=128-WN:T4=1-T4
2017 T1=0:T2=0:T3=0
2020 GOTO 2040
2022 WN=0:T4=0
2025 IF A=218 THEN WT=16-WT:T1=1-T1
2030 IF A=216 THEN WS=32-WS:T2=1-T2
2035 IF A=195 THEN WP=64-WP:T3=1-T3
2040 WF=WT+WS+WP+WN:PRINT CHR$(19)
2045 FOR J=1 TO 8:PRINT:NEXT
2050 PRINT "                                "
2055 PRINT "                                "
2060 PRINT CHR$(145) CHR$(145);
2065 PRINT "TIMBRE: LOW  MED. HI NOISE"
2070 PRINT SPC(8) T1"  "T2" "T3"  "T4
2075 GOTO 820
```

RUN the program. Press a note key and then press the Z key. Press the same note key again. You will hear a gonglike sound, and your screen will display

```
TIMBRE:  LOW   MED.  HI  NOISE
           1     Ø    Ø    Ø
```

There doesn't seem to be any musical terms to specify timbre except to say it is gonglike, pianolike, and so on. For the purposes of this program, I will call the four possible Commodore values *low*, *medium*, *high*, and *noise*. The words used are not particularly important because when you hear the effects produced you will judge for yourself how to use them.

A number 1 under LOW tells you that low timbre quality is selected, and a Ø under the others tell you that they are unselected. Press Z again and the 1 changes to Ø. Each of the four keys toggles its timbre quality on and off. Press X and then a note key. You will hear the familiar timbre that we have been working with and which we now label Medium.

We'll skip the C key for the moment and jump to the noise key, V. When you press V and play a note you get a crashinglike sound. When noise timbre is combined with function key sound shaping, you can produce all kinds of weird sounds. For example, press f3 and listen to the effect on a note as you increase the decay.

In order to use the high timbre key, C, you must add certain lines. First, press SHIFT/LOCK so it releases and then RUN/STOP to return to BASIC. Then enter the following:

```
902 IF A=145 THEN 2120
2100 REM
2101 REM * SUB.: SET PULSE WIDTH *
2120 PW=PW+PX:IF PW>4095 THEN PW=100
2125 HP=INT(PW/256):LP=INT(PW-256*HP)
2130 POKE M+2,LP:POKE M+3,HP
2135 PRINT CHR$(19)
2140 FOR J=1 TO 20:PRINT:NEXT
2145 PRINT SPC(17) "      "
2150 PRINT CHR$(145);
2155 PRINT "TIMBRE IS SET AT ";PW
2160 RETURN
5030 WF=32:PX=100:DE=9
```

Now RUN the program and press C. Press a note key and then the CRSR up/down arrows key. Near the bottom of your screen you will see

TIMBRE IS SET AT 1ØØ

Hold down the CRSR key and watch the numbers change on the screen. You will hear the change in timbre of the note. What is happening is that the program is arranged so that the last note played is repeated by the CRSR key and you can therefore hear the effect of sweeping over the timbre range. Numerically, this range is from Ø to 4Ø95, and in line 5Ø3Ø the program is set for increments of 1ØØ. You can change this by changing the PX value in that line.

You can have any one, two, or three of the timbre values—low, medium, and high—active at a time. Noise, on the other hand, is only selected alone. Notice that when you press V for noise, the other three values are set to Ø and vice versa. When timbre types are played in combination, the result is not a sum of amplitudes but a complicated combination that is difficult to predict. Such combinations may result in the loss of control, in which case it may be necessary to stop and restart the program.

NOTE: At least one of the four timbre values must be selected for any sound to be played.

The timbre is generated by three waveforms, a *triangular*, a *sawtooth*, and a *pulse wave*, corresponding to our terms low, medium, and high, respectively. The triangular waveform has the least overtones. The sawtooth and pulse waves are rich in overtones. The CRSR key controls the pulse width of the pulse wave, which has the effect of varying the overtone pattern. From this it can be seen that the designations medium and high are only approximate. What the waveforms do is generate a number of overtones from which you can select an overtone pattern for producing the sound you want. This is done with tone control, as described in the next section. Figure 3 illustrates schematically how timbre and tone are produced and how they are controlled.

Lines 1114 and 1116 are used to get waveform processing out of the way of the key selection statements that will follow. We will be doing this with all our operations from now on.

Lines 2015–2020 are for processing noise, and lines 2022–2035 are for the other three timbre levels. Lines 2015, 2025, 2030, and 2035 perform a switching operation. For example, consider line 2025. If key Z is pressed, WT is set to 16 minus the previous value of WT. If the previous value was 0, then WT will be set to 16, which is the value the computer uses to select the triangular waveform. The next time the Z key is pressed WT will be 16, so the new value will be 0, turning off the triangular wave. T1–T4 work in a similar manner, switching between 1 and 0 for the purposes of displaying selection status on the screen. In line 2040 all four timbre values are added so that the combined results can be obtained. There is no POKE in this section of code because WF is the value used in lines 20 and 55 to turn the note on and off. Thus, by changing WF as described above, the selected values will be POKEd into location 4 when the *next* note is turned on.

In line 5030, DE is set to 9 so that when the decay key, f3, is pressed the new value will be the old value plus 1, or 10, instead of 0 as before.

Lines 2120 and 2125 compute the new pulse width, PW, break it into high and low components, and POKE these into the two locations, 2 and 3.

Tone Control

Tone control is performed on the overtones created with keys Z, X, C, and V in timbre selection. It works in a manner similar to a tone control in a radio or stereo amplifier, except there is greater flexibility of control. With MUSICAL KEYBOARD LOADed, add

```
905 IF A=205 THEN 2360
2358 REM
2359 REM * TURN FILTER ON/OFF *
2360 FV=1-FV:POKE M+23,FV
```

```
2365 PRINT CHR$(19)
2370 FOR J=1 TO 13:PRINT:NEXT
2375 PRINT "TONE SWITCH IS " FV
2380 GOTO 820
```

RUN the program and press the M key. Your screen will display

TONE SWITCH IS 1

If you press a note key there will be no sound. Press the M key again and the 1 switches to Ø. Now you can sound all your notes in the usual way. Each depression of the M key toggles between 1 and Ø.

The M key is a tone switch into and out of the computer's tone control circuits. When the tone switch is Ø, the tone control circuits are bypassed so everything works as before. When the tone switch is a 1, all sound is routed into tone control. Since we have not yet entered provision for tone control into MUSICAL KEYBOARD, there is no output from these circuits. The following program lines will correct this:

```
904 IF A=60 OR A=62 OR A=63 THEN 2220
2200 REM
2201 REM * FILTERS ON/OFF *
2220 IF A=60 THEN FL=16-FL:T5=1-T5
2230 IF A=62 THEN FM=32-FM:T6=1-T6
2235 IF A=63 THEN FH=64-FH:T7=1-T7
2240 POKE M+24,FL+FM+FH+AM
2242 PRINT CHR$(19)
2245 FOR J=1 TO 16:PRINT:NEXT
2250 PRINT "                                    "
2255 PRINT "                                    "
2260 PRINT CHR$(145) CHR$(145);
2265 PRINT "TONE:    LOW  MED. HI"
2270 PRINT SPC(8) T5"   "T6" "T7
2275 GOTO 820
```

With these lines added, press the < key. Your screen will display

```
TONE:  LOW  MED.  HI
        1    Ø    Ø
```

Press the < key again and the 1 toggles to Ø. Each of these tone values toggles on and off in the same way as do the timbre values. Press > and ?, the

Waveforms

Filters

Bypass Tone Control

Cutoff Frequency

Triangle Wave
Timbre Low
Key Z

Low Frequencies
Key <

Ø

1

Key M
Switches
Tone On/Off

Sawtooth Wave
Timbre Medium
Key X

Middle Frequencies
Key >

Pulse
Width

CRSR Left/Right
Arrows
Changes Cutoff

CRSR
Up/Down Arrows
Key
Changes Pulse Width

Pulse Wave
Timbre High
Key C

High Frequencies
Key ?

Timbre

Tone

Figure 3. Timbre and tone control.

other two tone control keys. You can play with any combination of these three tone values. Unlike the timbre values, combined tone values produce the sum of the parts.

With all tones set to Ø, play a note. Now press < and play the note again. You will hear no difference. But now press the M key, and you will hear quite a difference. As discussed above, the tone switch has to be set for the tone control to work. Set all tones to Ø and press a note key. Again no sound. All tones are unselected. For the tone control to work *both* the switch and at least one tone must be selected. The tone switch could have been set automatically every time a tone was selected, but this would have made switching in and out of combined tones more difficult.

You have probably noticed that the tone settings do not yet give much selectivity. To correct this, enter the following:

```
906 IF A=157 THEN 2320
2300 REM
2301 REM * SUB: SET FREQUENCY CUTOFF *
2320 CH=CH+15:IF CH>255 THEN CH=0
2325 POKE M+22,CH:PRINT CHR$(19)
2330 FOR J=1 TO 22:PRINT:NEXT
2332 PRINT SPC(18) "     "
2333 PRINT CHR$(145);
2335 PRINT "TONE IS SET AT    ";CH
2340 RETURN
```

After entering RUN, press the CRSR left/right key. You will see the following line at the bottom of your screen:

TONE IS SET AT 15

Play a note and then press the CRSR left/right key. Nothing happens because the tone setting only has an effect when the filters are active. Press M and > and then a note key. Now hold down the CRSR left/right key and hear the tone sweep from low to high. Tone increments are set at 15 in line 232Ø. You can change this to other values, keeping in mind that the range is Ø–255.

Tone is controlled by means of three filters, *low pass*, *band pass*, and *high pass*. The tone is set by POKEing a low value in location 21 and a high value in location 22. The combined low and high values determine the cutoff frequencies of the filters. MUSICAL KEYBOARD does not make entries in the low values.

AUTOMATIC PLAYBACK BY THE COMPUTER

After creating sounds or music, you can let the computer play back what you have composed. You will be able to insert changes at any point and then continue the playback. You have probably found that a computer keyboard is not ideal for a musical instrument. To compensate for inconveniences during composition, we will provide for automatic playback so your composition can be heard as it might have sounded using a more suitable keyboard.

Add the following:

```
36 IF ML=1 THEN 45
820 IF ML=1 THEN 1320
821 GET K$:IF K$="" THEN 821
825 A=ASC(K$)
1119 REM
1120 REM * NOTE SELECTION *
1123 IF A=219 OR A=221 OR A=48 THEN1710
1128 REM
1129 REM * SWITCH COMPOSE/PLAY *
1130 IF A<>95 THEN 1191
1135 ML=1:A(0,0)=1:Q=0:GOTO 820
1300 REM
1301 REM ** PLAY COMPOSITION **
1320 GET K$:IF K$<>"" THEN 1360
1325 TD=A(Q,0)+TI-7:NT=A(Q,1):Q=Q+1
1330 IF NT=0 THEN ML=0:Q=Q-1:GOTO 820
1335 IF TD-TI>150 THEN TD=TI+150
1340 IF TI<TD THEN 1340
1345 RETURN
1359 REM
1360 IF K$=CHR$(95) THEN ML=0
1370 GOTO 820
1700 REM
1701 REM * CHANGE NOTE NO. *
1710 IF A=219 THEN Q=Q+1
1715 IF A=221 THEN Q=Q-1
1720 IF A=48 THEN Q=0
1722 A(Q,0)=40
1725 PRINT CHR$(19) "            "
1730 PRINT CHR$(19) "NOTE " Q:GOTO 820
5032 DIM A(999,2)
5132 IF ML=1 THEN 5120
5135 A(Q,0)=TI-T:T=TI:A(Q,1)=NT:Q=Q+1
```

This is a little longer than most of our additions. You can break out lines 1700–1730 if you prefer, and enter them separately.

When making the above entries, make sure you press the letter I (not the number 1) to form TI. TI is not a variable; it is a clock. This is explained below.

Compose a short composition, of just play a series of notes for use as a test. Then press the left arrow key that is in the upper left-hand corner of your keyboard. Your composition will be played back. When it is over, you will be returned to the compose mode. Anything you enter now will be added to the previous composition. If you press the left arrow key again, the playback will start at the beginning of your original composition and will continue to the end of the section you added.

Start the playing of your composition; while it is playing, press the left arrow key. The playing will stop, and you will be switched back to compose mode. Now you can make any changes you want; they will overwrite your previous entries at the point you stopped the playing.

To exit the program, press SHIFT/LOCK to return to the unshifted mode. Now the RUN/STOP key is active, and you can exit in the usual way.

Controlling Playback

The three keys plus, minus, and zero are used to control your position in the composition, allowing you to move backward or forward in one note steps or to jump to the start of the composition.

You may have noticed that usually about one note is played after you try to stop playing using the left arrow key. You can correct for this by pressing the minus (−) key. This will move you back one note for each pressing of the key. It is unlikely that you will move so far back that you will be before the first note, but if you do you will get an error message and will have to restart the program. The plus (+) key can be used to move to notes after the one at which the playing stopped. The zero (0) key lets you restart at the beginning. Plus, minus, and zero are not active during playing; you must be composing to use these keys.

Each time you press plus, minus, or zero the number of the note is displayed in the upper left-hand corner of your screen. Notes are numbered as you enter them, starting with 0. However, they are displayed only when plus, minus, or zero is pressed. To find the current note number, just press plus and then minus. There will be no net change in the note, but the current note will be displayed. These numbers will help you locate parts of a composition or specific notes when you want to enter changes.

Lines 1123 and 1710–1730 perform the processing for position control. In shifted mode, 219, 221, and 48 are the codes for +, − or 0 symbols, respec-

tively. These are selected in line 1123. Q is the variable that stores the current note number, starting at Ø for the first note.

If while composing you press the space bar to shift to higher and lower octaves, this will be remembered and reproduced in the play mode. All other changes, that is, those made with the control keys, are not remembered. Instead, the playback will create sounds that are fashioned by the last settings made by the control keys for ADSR, timbre, and tone. This feature allows you to create a composition with one set of control values, save it, and then play it back with different settings, giving you an amazing diversity of sound.

The ADSR item, release, requires special consideration. Release only starts at the time a note is turned off. Therefore, when you are playing, any significant amount of release will slow the response of the notes. That is, when you play one note, the release of the previous note could still be active. The result is that you won't hear the note when you press it but will hear it after the end of the release phase of the previous note. Because of this, release, which is controlled by the variable TP in line 25, is set to Ø in MUSICAL KEYBOARD. This means that release is only active by pressing keys 6 or 7 when a note is ended not when a new note is played.

Table 2 shows that release times are very short for all POKE values less than 9. You can also see that there is no simple relationship between the POKE values Ø–15 and release times. All this means is that it not easy to incorporate these release times into keyboard playing. If you want to experiment with release times, set TP to correspond to the values given in Table 3. You might try TP=.Ø24*RE^(RE*.17), where RE is the release POKE value derived in line 1225.

How Music and Sounds are Saved

Lines 1325–134Ø and 5135 are the key lines for saving keyboard entries. During composition, notes and the time between notes are saved in an array A(Q,X). *Arrays* are simply very convenient places to store values that are closely related. The array's indexes, Q and X, are used to locate any element of the array. As its name implies, an array can be thought of as a number of elements arranged, or arrayed, in intersecting rows and columns.

The array we are using here, A(Q,X), has two rows and 1ØØØ columns. This is specified in line 5Ø32, which gives the dimensions of the array. Array numbering starts at Ø so the dimensions are 999 and 1. When BASIC sees the dimension statement, DIM, it reserves the space in memory specified by DIM, which in this case is 2ØØØ bytes. You can increase or decrease this number to fit the size of your composition and your memory requirements for other functions.

After each note is played in the TURN NOTE ON/OFF subroutine, processing returns to MAIN LINE at line 5135, where the time between notes is

stored in the first row of the array, A(Q,Ø), and the notes themselves are stored in the second row of the array, A(Q,1).

The time between notes is obtained from the system clock, TI. At all times, as long as the computer is on, TI is ticking away. This provides a very useful tool that can be used in many ways, including games, tests, and, in the present case, the timing of music. You can see what is in TI by entering PRINT TI. TI does not give time in hours, minutes, and seconds; to see time in that format use TI$. TI is just a counter that is incremented at equal intervals. This is all we need in order to time notes.

We are interested only in elapsed time, not the actual value of TI. Therefore, in line 5135, we set a time variable, T, equal to TI after a note is played. Now T remains constant at this value, but TI, being a clock, continues to increase. The next time you play a note, line 5135 computes the elapsed time, TI-T, and sets A(Q,Ø) equal to this value. Then, it resets T to the new TI for computing the following note interval.

Line 5135 also stores the note played. This is the note value after calculation in lines 38 and 4Ø, which factor in the octave. This is how the octave setting is saved with the note.

After the note interval and the note are saved in A(Q,X), Q is incremented by 1, setting the array column for the next cycle.

How Music and Sounds are Recreated

The left arrow key has a code of 95, which switches processing to line 1135. The play mode is activated in line 1135 by setting variable ML to 1.

When the play mode is active, ML is equal to 1 so line 82Ø branches to the PLAY COMPOSITION routine at line 132Ø. First, a test is made to see if the left arrow key was pressed for exiting the routine. If not, line 1325 performs the reverse process of line 5135, extracting from the array, A(Q,X), the information previously stored there and moving the array index, Q, to the next column. A time delay variable, TD, is computed by adding the original time interval stored in A(Q,Ø) to the current value of TI. A 7 is subtracted in order to adjust for differences in computer time. You can change this value to change playback speed.

Line 133Ø tests for the end of the composition, signaled by a Ø note value. Lines 1335–134Ø recreate the original time interval between notes. First, line 1335 tests the newly created time interval. If it is too large, that is, greater than 15Ø, it is set to 15Ø. This prevents long waits that would occur at the beginning or at points at which you stopped to change control keys, or answer the telephone, or whatever. The value 15Ø was selected purely on the basis of what seemed to work; you might want to change it for your own purposes.

Once TD is established, we simply wait for TI to catch up to TD. Line 134Ø

loops back on itself until TI is equal to TD, after which processing returns to MAIN LINE for the playing of the note. Since A(Q,1) already has the octave factored in, line 36 bypasses this processing in the subroutine.

SAVEING MUSIC ON TAPE OR DISK

After working hard to create music for your friends or sounds for your games, you need to be able to SAVE what you have. This can be done with a few additional lines of code. The following routines are typical write and read routines for sequential files. They can be used as templates for other programs.

Writing Sequential Files

Sequential files store items on disks or tapes one after the other as they are sent from memory. They are read back in the same order. This is just what we want to do with the notes and time intervals stored in the array A(Q,X).

Add the following write routine:

```
1124 REM
1125 IF A=33 OR A=34 THEN 1520
1500 REM
1501 REM *** DISK/TAPE TRANSFER ***
1520 PRINT CHR$(147) "PRESS SHIFT/LOCK"
1525 INPUT "FILE NAME";F$
1530 IF A=34 THEN 1620   :REM LOAD ?
1535 REM
1536 REM * SAVE *
1540 OPEN 2,8,2,"@0:"+F$+",S,W"
1545 FOR J=0 TO 999:FOR K=0 TO 1
1550 PRINT#2,A(J,K)       :REM WRITE
1555 IF A(J,K)=0 THEN 1565
1560 NEXT K:NEXT J
1565 PRINT#2,CHR$(13)
1570 CLOSE 2:GOTO 5082
```

If you are using tape, substitute the following for line 1540:

```
1540 OPEN 2,1,1,F$
```

Enter another test composition. Press the left arrow key to check that the play

mode is working as before. Then press the 1 key to save your composition. In shifted mode, the code for 1 is 33, which is used in line 1125 to branch to the DISK/TAPE TRANSFER routine. You will see these lines on your screen:

PRESS SHIFT/LOCK
FILE NAME?

You are in shifted mode so in order to enter the file name, you must return to unshifted mode. We will see later what happens if you forget to do this. For now, press the depressed SHIFT LOCK key so it pops up. Enter a file name up to 16 characters long, for example, COMPOSITION 1. Press RETURN.

If you are using tape, you will be asked by a screen message to PRESS RECORD & PLAY ON TAPE. No such message will appear for disks. Your tape or disk unit's red light comes on, the unit runs a few seconds, stops, and the light goes off. If the light blinks or does not go off on the disk unit, the write operation was not performed properly. Check that a disk is properly inserted and the latched door is closed. Be sure your disk has been initialized and the green light is on.

When transfer is complete, the following message appears on your screen:

*** PRESS SHIFT LOCK ***
THEN J KEY TO START

These lines are a reminder to reenter shift mode in order to make entries. When these operations are performed, the screen clears and you will see the title line MUSICAL KEYBOARD. This is as far as we can go now. To play back what you have written, you need a read routine. To activate the RUN/STOP key, press the SHIFT/LOCK key. The key pops up. Now exit with RUN/STOP–RESTORE.

Reading Sequential Files

Add the following read routine:

```
1600 REM
1601 REM * LOAD *
1620 OPEN 2,8,2,"@0:"+F$+",S,R"
1625 FOR J=0 TO 999:FOR K=0 TO 1
1630 INPUT#2,A(J,K)       :REM READ
1635 IF A(J,K)=0 THEN 1645
1640 NEXT K:NEXT J
1645 CLOSE 2:GOTO 5082
```

If you are using tape, substitute the following for line 162Ø:

```
162Ø OPEN 2,1,Ø,F$
```

When you have brought your program up and have the title line, MUSICAL KEYBOARD displayed, press the 2 key. The code for 2 in the shifted mode is 34 and is tested for in lines 1125 and 153Ø.

The same two lines are displayed that you saw when you wrote your file. Press the SHIFT/LOCK key and enter the name of the file you saved. Press RETURN. If you have tape, you get the PRESS PLAY ON TAPE message. The disk or tape unit does its thing. Then a message on the screen reminds you to press the SHIFT/LOCK key and any other key—the same messages as when you wrote the file.

Press the left arrow key to play the file you just read. Does it sound the same? You can now save and retrieve files for MUSICAL KEYBOARD.

In the FOR...NEXT loops in lines 1545–156Ø and 1625–164Ø, the variable J is used instead of Q. This is another nice thing about arrays: You can use different variables as indexes. In this way Q does not get changed by the write and read operations.

The statements that perform the actual write and read are PRINT# and INPUT# in lines 155Ø and 163Ø, respectively. These statements, as well as CLOSE and other aspects of writing and reading, are explained later in connection with writing and reading sprites. If you are interested now, refer to the appropriate chapter or your *User's Guide.*

THE LAST NOTE

This completes the MUSICAL KEYBOARD program. It is hoped you will have many hours of enjoyment exploring the fascinating world of computer music. There are several features in the Commodore 64 that have not been included in MUSICAL KEYBOARD. Amplitude control, filter resonance, and multiple voices were left out because their inclusion would require a book devoted solely to sound and music. If you are interested, you can learn about these in the *Commodore User's Guide* and the *Programmer's Reference Guide,* also published by Commodore.

TABLE 4 KEY ASSIGNMENTS FOR THE MUSICAL KEYBOARD

Key	Note	Cycles/Sec.
	Note Keys	
RUN/STOP	C	131
Q	C#	139
A	D	147
W	D#	155
S	E	165
D	F	174
R	F#	185
F	G	196
T	G#	2Ø7
G	A	22Ø
Y	A#	233
H	B	247
J	C	262
I	C#	277
K	D	294
O	D#	311
L	E	329
:	F	349
@	F#	37Ø
;	G	392
*	G#	415
=	A	44Ø
Up arrow	A#	466
RETURN	B	493

SPACE BAR toggles notes 1–12 to half their frequency and notes 13–24 to twice their frequency.

Control Keys

f2	Attack. Cycle values Ø–15
f4	Decay. Cycle values Ø–15
f6	Sustain. Cycle values Ø–15
f8	Release. Cycle values Ø–15
Z	Timbre low
X	Timbre medium
C	Timbre high
V	Timbre noise
M	Turns tone control off and on
<	Tone low
>	Tone medium
?	Tone high
CRSR up/down	Sweeps timbre values when timbre is set to high
CRSR left/right	Sweeps tone values

TABLE 4 KEY ASSIGNMENTS FOR THE MUSICAL KEYBOARD (Cont.)

	Playback
Left arrow	Toggles between play and compose modes
	Positioning in the Composition
Ø	Sets compose mode to the first note
+	Advances the compose mode one note
–	Moves compose mode one note back
	Disk or Tape Transfers
1	Save on disk or tape
2	Load from disk or tape

6

ELECTRONIC DRAWING BOARD

Your Commodore computer has the ability to draw dots, lines, circles, and arcs on the display screen. The process is complex, but we will sidestep the complexity and draw on the screen just by using the keyboard. We will also show you an exciting new way to control the cursor, called *cursor steering*. Instead of your moving the cursor, the cursor moves itself and you steer it in the direction you want it to go.

The Commodore has the ability to place a dot on the display screen in any position you designate. Whereas characters are formed from 8 × 8 squares of dots, individual dots, which can be turned on and off, can be laid out in sequence to form lines, arcs, circles, and all the figures that can be generated by combining these basic forms.

BIT MAPPED GRAPHICS

The technique of placing dots rather than characters on the screen is called *bit mapped graphics*. The term "graphics" refers to the display of any nontext images, not just graphs in the usual sense. Thus, we will use bit mapped graphics to generate the lines, arcs, and circles created by the DRAWING BOARD program. In this context you may think of a bit as representing one dot position on the screen.

There are memory locations for determining screen position and color in bit mapped graphics just as there is in character graphics. However, because there are many more dot locations than character locations—8 × 8 or 64 times as

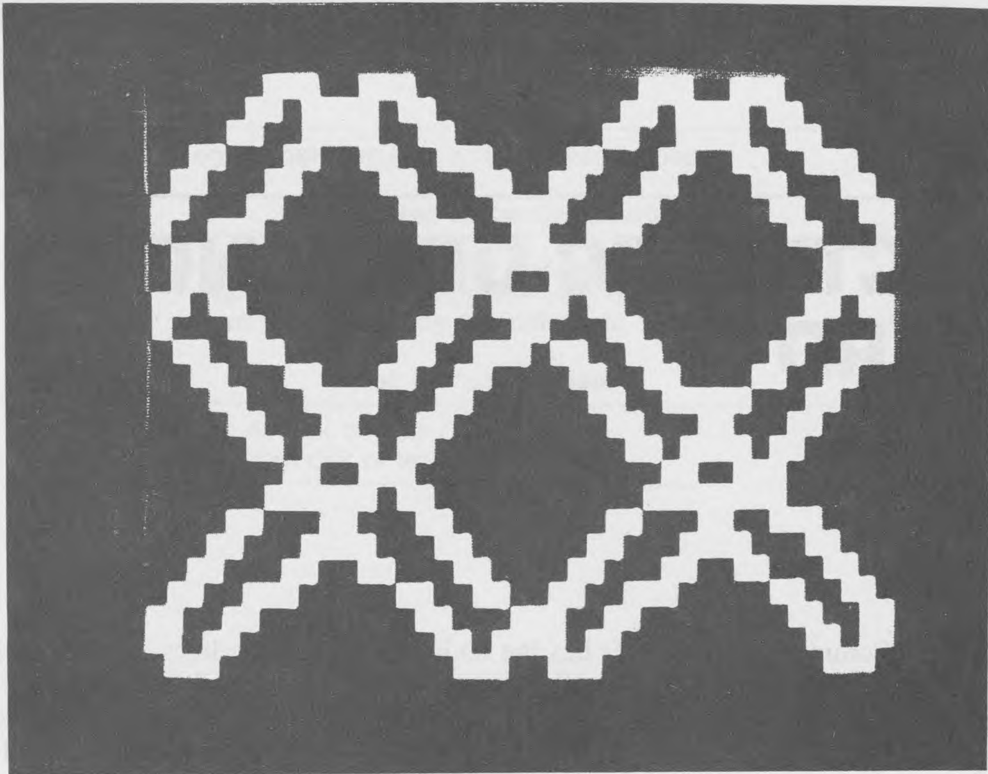


Photo 5 A design created by the "Mole" on the DRAWING BOARD. The Mole digs tunnels of light in strange ways, or, as in this illustration, in symmetric patterns.

many—bit mapped graphics works quite differently from character graphics. We will explain how bit mapped graphics works, but you don't have to understand any of this to use the DRAWING BOARD program. If you want to follow the technical details, refer to Figure 4 as we go along.

The extra memory required for bit mapped graphics is allocated out of the memory that is available to you, the user, and reduces total memory available for programs and data by 8ØØØ characters.

KEYBOARD CONTROL OF THE DRAWING BOARD

There are a number of gadgets on the market that can assist you in drawing objects and designs on your screen. One such device is a pad across which you run a *stylus*. The stylus can be used to draw and to point to objects displayed on the screen as a way of giving instructions to the computer. Another device, called a *mouse*, is rolled on the table top in the direction that you wish to move

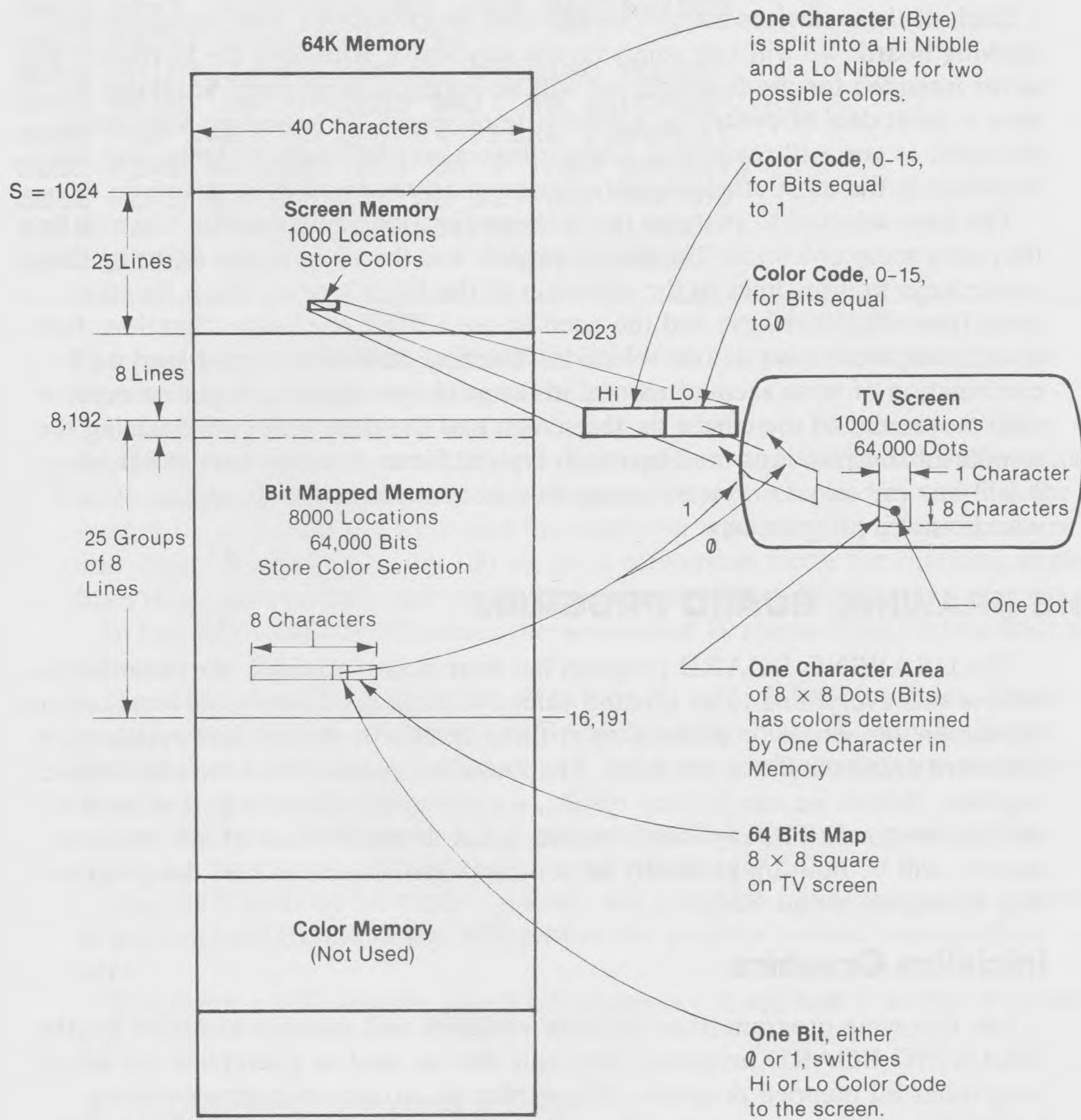


Figure 4. Bit mapped graphics. Screen memory and bit mapped memory can be moved.

the cursor or draw a line. There are also *game paddles* and *joysticks*, which can be adapted for drawing.

Each of these devices has advantages and disadvantages. For our electronic drawing board, we will rely solely on the keyboard. Although the keyboard was never intended for the functions we will be asking it to perform, it allows you to have a great deal of control at a good rate of speed. Its numerous keys act as switches, as you will see below. Other devices may be easier to learn, but the keyboard is the most efficient entry device once you know how to use it.

The keys selected to perform the drawing functions may surprise you. At first they may seem awkward. The first approach was the obvious one of using the cursor keys to draw lines in the direction of the key's arrows. Their location away from the other keys and the need to use SHIFT to change direction, however, made these keys a poor selection. The final selections were based on a combination of what seemed natural in terms of coordinating finger movement with movement of the cursor on the screen and the desirability of retaining the standard hand positions used by touch typists. Since you may have other ideas, it will be very easy for you to change key assignments in the program, even if you are not a programmer.

THE DRAWING BOARD PROGRAM

The DRAWING BOARD program has four major routines. An *initialization routine* sets everything to its starting value. A *location routine* finds locations on the screen and writes or erases dots at those locations. A *keyboard routine* gets keyboard entries and acts on them. The *main line routine* links the other three together. Before we can get any results, we will need at least a part of each of the four routines. The keyboard routine, which is where most of the program resides, will be built up gradually so you can learn about and test the program step by step as we go along.

Initialize Graphics

The following program lines initialize variables and memory locations for the DRAWING BOARD program. They can also be used as a template for initializing other bit mapped programs. If you plan to do your own programming SAVE this routine separately. You can use the routine's name, INITIALIZE GRAPHICS, to SAVE and LOAD, but the computer will truncate this to 16 characters.

Enter NEW and then these lines:

```
5010 REM
5011 REM ** INITIALIZE GRAPHICS **
```

```
5020 B=53265:BA=8192:B7=B+7:H=160:V=100
5025 C=0:C1=5:CG=5:DH=1:DV=1:RA=1
5030 AR=3.14159265/180 :REM RADIAN/DEG
5035 PRINT CHR$(147):J$="          "
5040 PRINT J$ "CLEARING DRAWING BOARD"
5045 PRINT J$ "PLEASE WAIT 30 SECONDS"
5050 FOR J=BA TO BA+7999:POKE J,0:NEXT
5055 POKE B7,PEEK(B7) OR 8 :REM BA=8192
5060 POKE B,PEEK (B) OR 32 :REM GRPHCS
5065 FOR J=1024 TO 2023:POKE J,5:NEXT
5070 GOSUB 20              :REM CURSOR
```

The program starts with high line numbers so we can put subroutines that we want to run fast at the beginning of the program.

Lines 5020–5025 initialize variables, each of which will be described below. In line 5030 you may recognize the nine digit number as pi, the ratio of the radius of a circle to its circumference. You can also simply press the key that has the symbol for pi (π) on the front and the computer will interpret this as the same nine digits. By dividing pi by 180 we get a conversion factor for changing angles from degrees to *radians*, the angular measure used by BASIC.

In line 5035, CHR$(147) clears the screen and J$ spaces the next two lines at the center of the screen.

Line 5050 clears bit memory by POKEing 0's into all 8000 locations. This is a lengthy process when performed by BASIC, taking about 30 seconds. Hence, lines 5040 and 5045 are provided to give you some evidence that the program is on its way during this initializing.

Line 5055 sets the location of bit memory at 8192. For other possible locations, see the *Commodore 64 Programmer's Reference Guide*.

Line 5060 turns on bit mode graphics, and line 5065 initializes screen memory to a green background. Line 5070 creates the graphics cursor; more on this later.

This is not a full program, but RUN it anyway to see how it works. You will see

```
CLEARING DRAWING BOARD
PLEASE WAIT 30 SECONDS
```

The program is now clearing bit memory as per line 5050. After about 30 seconds, you will see the screen painted with a green background; then two colored bars will appear. This is an error message, but since you have switched to bit map mode, instead of characters all you get are patches of color! The reason for the error message is that line 5070 says to go to subroutine at line 20. Since

there is no such subroutine, BASIC cries foul. Press RUN/STOP–RESTORE to get back to normal display.

How can you find out what is wrong if error messages don't display in readable text? There is a way: Enter the following *temporary* line:

```
5047 GOTO 5070
```

This line allows the computer to bypass the bit map instructions so the error message will print out. Of course, the program can no longer produce graphics. Once you know the problem, delete line 5047.

When INITIALIZE GRAPHICS is working as described above, SAVE it with the name DRAWING BOARD. You now have the beginning of the DRAWING BOARD program. We will add to it in the following sections, and you should SAVE it periodically. This is no problem on disks. On tape you must know where you placed the latest version. It is best to use a new tape and always work from the beginning, overwriting previous versions.

Locate a Bit

With the DRAWING BOARD program LOADed in memory, add the following subroutine.

> **NOTE: In lines 75 and 85 the printer prints a ^, the symbol for raising to a power, instead of an up arrow. You should use the key with the up arrow in place of ^.**

```
11 REM
12 REM ** LOCATE A BIT U/L **
20 Z=V*.833
25 IF H<=000 THEN H=000:MH=-MH
30 IF H>=319 THEN H=319:MH=-MH
35 IF Z<=000 THEN Z=000:MV=-MV
40 IF Z>=199 THEN Z=199:MV=-MV
45 CH=INT(H/8)
50 RW=INT(Z/8)
55 LN=Z AND 7
60 BY=BA+RW*320+CH*8+LN
65 BT=7-(H AND 7)
70 IF FE=1 THEN 85
75 POKE BY,PEEK(BY) OR 2^BT
80 POKE 1024+RW*40+CH,CG:RETURN
85 POKE BY,PEEK(BY) AND 255-2^BT:RETURN
```

The UL in the program name on line 12 signifies that position locations start in the upper left-hand corner. When the screen area is used in bit mapped graphics mode, it is made up of 200 horizontal lines numbered 0–199 and 320 vertical lines numbered 0–319. A dot can be located at the intersection of any vertical (V) and horizontal (H) lines, for a total of 64,000 dot locations. For example, when the LOCATE A BIT U/L subroutine is given location V=0, H=0, it will place a dot in the upper left-hand corner of the screen. Locations increase to the right and down so that the lower right-hand corner is location V=319, H=199, or 319/199.

Each time LOCATE A BIT U/L is called, it either writes or erases one dot at the location specified by V and H. The program then returns to the calling program. Lines, arcs, and circles are built up, dot by dot, by repeated calls to LOCATE A BIT U/L.

Aspect Ratio

Line 20 adjusts the aspect ratio of the display. The *aspect ratio* refers to the different distances between dots in the vertical and horizontal directions. On standard TV screens the distance between dots in the vertical direction is 0.833 (six-fifths) greater than in the horizontal direction. The result is that circles look like ellipses, squares look like rectangles, and diagonals do not have a 45 degree slope. This is easily corrected by multiplying, as in line 20, all vertical distances by 0.833. You can change this number to see what effect it has.

There is one drawback to correcting the aspect ratio. Diagonals will no longer be smooth lines; they get slight zigs or zags. This is something inherent in graphics produced on a TV screen, which was never designed for this type of display. Figure 5 shows why lines composed of dots appear jagged at certain angles.

Keeping the Dots on the Screen

If the dots are allowed to wander off the screen, not only would you lose them, but they may get POKEd into memory locations where all kinds of weird things can happen. Lines 25–40 keep the dots in the screen area. If they try to wander off, they get deflected off the edge like a bouncing ball. Later you may want to change this so that a dot going off one edge wraps around and enters at the opposite edge. Many interesting effects can be produced by setting the conditions in lines 25–40 to various values.

Moving, Writing, and Erasing Dots

Lines 45–65 compute the position in bit memory from H and Z (computed in line 20) values. The position of a dot depends on the ordering of bits in bit

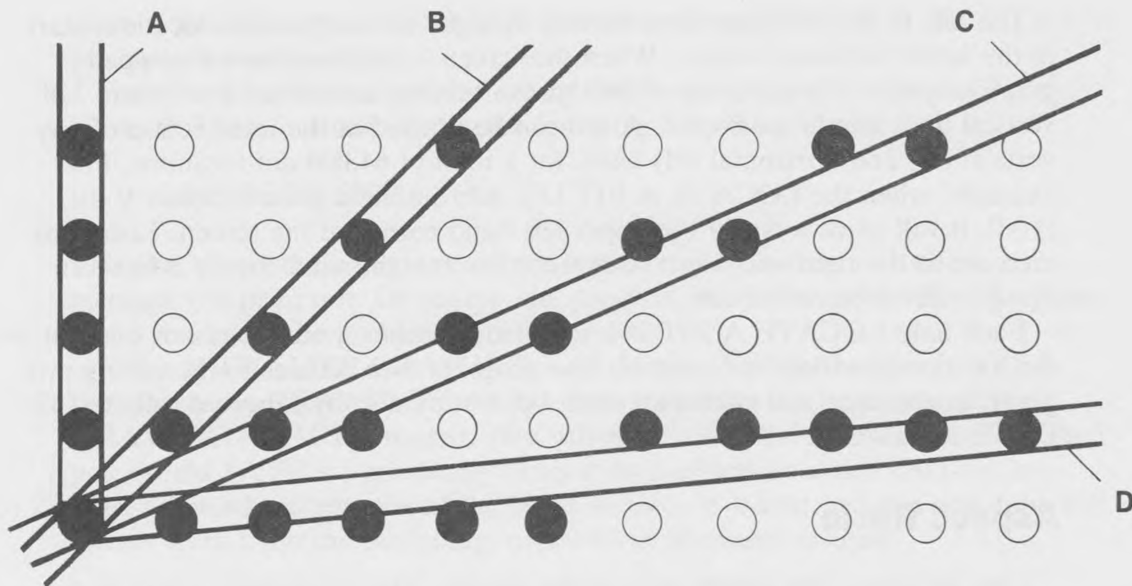


Figure 5. Lines drawn with dots on a TV screen.

memory and the ordering of dots on the screen. These must be correlated, which is what lines 45–65 do.

Figure 4 illustrates bit mapped graphics. Notice that bit mapped memory has eight times (8ØØØ) as many memory locations as screen memory (1ØØØ). For each character position on the TV screen, screen memory has one corresponding location, and bit mapped memory has eight corresponding locations. Since each location stores 8 bits, bit mapped memory has 64 bits for each of the 64 corresponding dot positions on the TV screen. On the screen, these 64 bits are arranged in an 8 × 8 square, but in bit mapped memory they are in consecutive locations. The eight character locations in bit mapped memory can be thought of as sandwiched between successive locations in screen memory. Lines 45–65 map the consecutive bit locations in bit mapped memory into an 8 × 8 dot square on the TV screen. The factor of 8 in line 6Ø takes care of the eight characters in bit mapped memory, while line 55 computes the 1 of 8 bytes that has been selected. Finally, line 65 computes the bit position in the selected byte.

In line 7Ø, the FE (flag for erase) controls writing and erasing. If FE = Ø, lines 75 and 8Ø are executed. Line 75 POKEs a 1 in bit memory at the location computed in lines 45–65. Then line 8Ø sets the color of the corresponding location in screen memory to the value of the CG (color of graphics) variable.

If you have worked with the character graphics described in previous chapters, you may be surprised to find us POKEing color into screen memory where characters are usually stored. What happened to color memory way up in locations

55296–56295? Bit mapped graphics does not use color memory (except in multicolor mode); it uses screen memory. This is possible because the dot information is stored in bit memory, leaving screen memory free for color. Screen memory has only 1ØØØ characters, nowhere near enough for 64,ØØØ dots. Therefore, it stores color for each character position, not each dot position. Further, since in each character position it stores the color of both the dot and the background, each of these has a range of only 8 colors instead of the 16 colors available in character graphics. If you think this is confusing, I agree. But referring to Figure 5 may help, and you will see how it works when you actually run the program.

If FE is a 1 in line 7Ø, lines 75–8Ø are skipped and line 85 is executed. This is the line that erases a dot by POKEing a Ø into the computed dot location. When a Ø is stored in bit memory it causes the background color to be displayed at that position instead of the foreground color, effectively erasing any dot previously displayed in the foreground color.

To summarize, Ø's and 1's are stored in bit memory. A 1 causes a dot to appear on the screen at a corresponding location. The dot is displayed in one of seven foreground colors. A Ø causes the background color to be displayed. Switching to background erases a dot. The same seven possible colors used for the dots are available for background. Foreground and background colors are stored in character locations (bytes) of screen memory. Half of each location (1 nibble or 4 bits) stores one of seven foreground colors, and the other half stores one of seven background colors. The upper half is selected by a 1 in bit memory, writing a foreground dot. The lower half is selected by a Ø in bit memory, writing a background dot, which is the same as erasing a foreground dot.

To move a dot it is necessary to erase it in its current position and then write it in a new position. This requires two passes through the LOCATE A BIT U/L subroutine. On the first pass the FE is set to 1 by the calling program so that the dot is erased. The calling program then sets the new location values in H and V and sets FE to Ø, causing the dot to appear at the new location.

POKEing and PEEKing in Graphics Memory

Lines 75 and 85 have POKEs that POKE PEEKs. They also contain the logical operators AND and OR. All this calls for some explanation.

These complex POKE statements are required because individual memory locations store information of more than one thing, namely eight screen dots. When the computer writes in memory it always writes a complete character, never single bits. We want to set an individual bit to a Ø or a 1. But the bit we want is in the same location as seven others. If we simply POKE a 1, the computer actually POKEs ØØØØØØØ1, which is the computer representation of a 1. The other seven bits at that location are set to Ø, irrespective of what they were

before. This is not what we wanted at all. We must find a way to set 1 bit without affecting the other bits at the same location.

By PEEKing at a location we get all 8 bits of its contents out of memory. If we can change just 1 bit, we can POKE it back into memory with only 1 bit altered. This is where the AND and OR operators come in. OR lets us set a single bit to a 1 without changing any other bit. AND lets us set any bit to a Ø. You can read Appendix A to see how this works.

We had a similar situation when we wrote in music memory in Chapters 4 and 5. There, for example, the same location stored the type of waveform and the ON/OFF bit of the sound. However, music memory is different because you can only write in it; you cannot read it. Therefore, we couldn't PEEK to see what was there. Instead, we had to know what had been entered, and we POKEd a number that would reproduce the old values and at the same time write the new value.

Displaying the Cursor

You now have a program with two routines. You can run the program by starting execution in the INITIALIZE GRAPHICS routine at line 5Ø2Ø. Enter

```
RUN 5Ø2Ø
```

The two-line message is displayed, and after about 3Ø seconds the screen is painted green, just as before. But then a dot appears at the center of the screen followed by a single colored bar at the left. The dot is a *graphics cursor*. Character cursors mark a character space and are 1 character wide. Since the graphics cursor must mark a single dot location, it can be no larger than a dot. In more expensive graphics systems, the cursor is often a pair of cross hairs in a circle. But a single dot will serve us just fine. You will learn later how to find the tiny dot easily if it gets camouflaged in a complex design.

The color bar is nothing but the READY message of BASIC altered by bit mapped graphics. When the program started, H and V were initialized in line 5Ø2Ø to 16Ø and 1ØØ, respectively, the center of the screen. CG was set to 5 in line 5Ø25, producing a black dot on a green background. When the LOCATE A BIT U/L subroutine is called in line 5Ø7Ø, FE is Ø so a dot is written in the center of the screen. At line 8Ø, the program RETURNs to INITIALIZE GRAPHICS. But since there are no more program steps, the program terminates normally, and BASIC returns to command mode with its READY message.

If the program works as described, SAVE it under the name DRAWING BOARD. This program can also be used as a template for other bit mapped graphics programs. If you want to be able to use it in this way, SAVE it again under another name.

The Main Line

The main line routine links the keyboard routines to the LOCATE A BIT U/L subroutines and performs some intermediate processing. With DRAWING BOARD LOADed, add the following lines:

```
1 REM ***** DRAWING BOARD *****
10 GOTO 5020    :REM BY PASS SUBROUTINES
5000 REM
5001 REM **** START OF PROGRAM ****
5100 REM
5101 REM **** MAIN LINE ****
5120 GOSUB 820
5125 IF DE=0 THEN 5140
5130 FE=1:GOSUB 20          :REM DELETE
5135 FE=0
5140 H=H+MH*INT(RA*DH)
5145 V=V+MV*INT(RA*DV)
5150 GOSUB 20               :REM WRITE
5155 GOTO 5120
```

Line 1 is the program title. Line 10 jumps the program to the initialization routine, bypassing all intermediate lines. Line 5001 helps you spot where processing starts when the program is initiated with a RUN command.

After initialization, the first thing the program does is go to the keyboard subroutine, line 820, to get a key entry. We will be adding the keyboard subroutine in the next section. For now, just notice that the keyboard subroutine is called by main line in line 5120. When this happens the program waits for you to press a key. The keyboard subroutine will process your entry and then RETURN to main line at line 5125.

Line 5125 tests to see if you have pressed a key signaling the program to move the cursor without drawing. If so, the DE (delete) variable is set to 1. If not (the default case), DE is 0. When DE=0, lines 5130–5135 are bypassed. Otherwise, the FE character is set and the LOCATE A BIT U/L subroutine is called to erase the dot at the current location. On return from the subroutine, FE is reset to 0, enabling writing of a dot. After line 5135, processing is the same whether DE is 0 or 1.

Lines 5140 and 5145 compute the new values of the horizontal and vertical dot locations from variables MH, MV, RA, DH, and DV. The values for these variables are set in the keyboard routine. Variables MH and MV determine the movement in the horizontal and vertical directions. If MH or MV is +1, movement will be to the right or down. If MH or MV is −1, movement will be to the

left or up. RA is a general-purpose size variable. It may specify radius, the distance moved, and other such variables. DH and DV are used to vary H and V, respectively, thus allowing you to set the slope of lines and the curvature of arcs. These five parameters provide considerable flexibility in what you can draw on the screen.

You cannot move between dot locations, so fractional values for H and V have no significance. The only meaningful values for H and V are whole numbers. It follows that H and V can only be incremented in whole number steps such as 1, 2, 3. However, when BASIC converts a fractional value to a whole number it makes positive numbers and negative numbers smaller (that is, the absolute value of the negative number is increased). For example, +1.6 becomes +1, and −1.6 becomes −2. In graphics this has the effect of moving dots farther left than right. If we allowed this to happen we would get all kinds of distortion in our figures. We must therefore make sure that positive and negative numbers have the same value so that circles are circles and lengths of moves don't depend on their direction. This is done in lines 514Ø and 5145 by first converting to integers with INT and only then multiplying by MH or MV to add the sign. RA, DH, and DV must be kept positive so that INT never has a chance to introduce distortion.

When the new values of H and V have been computed, the LOCATE A BIT U/L subroutine is called to write one dot. Upon return, line 5155 loops the program back to 512Ø, and the whole process is repeated. This program never stops by itself. To exit the program, press the RUN/STOP–RESTORE keys.

You can now see how main line controls the moving, drawing, and erasing of dots. If DE is Ø, a new dot is written without erasing the previous dot, resulting in the drawing of a line. If DE is 1 the previous dot is erased. If the previous dot was only the cursor, the result is to move the cursor. If the previous dot was also part of a line, the result is to move the cursor *and* erase the line. You will see how this works on the screen as we add keyboard control.

You can test the present version of your program by entering RUN; line 1Ø will take care of starting the program at the correct line. Again you will see the wait 3Ø seconds message and then a black cursor on a green background. The program ends with an error message in the language of a bit mapped graphics, which is caused by the call in line 512Ø to the nonexistent keyboard subroutine. All this test does is ensure that none of your entries have changed the previous routines; main line will not be exercised until the next section.

DRAWING WITH THE KEYBOARD

You have now reached the point where you will be able to draw with the keyboard. To do this LOAD the DRAWING BOARD, and then add the following

lines. If you have saved KEY ENTRY from Chapter 3, you can use it as a template.

```
800 REM
801 REM ******************************
802 REM *** SUB.: GRAPHICS KEYS ***
820 REM
825 GET K$:IF K$="" THEN 825
832 REM
833 REM * DIRECTION KEYS *
836 IF K$=";" THEN 1020 :REM RT.
838 IF K$="H" THEN 1025 :REM LEFT
840 IF K$="U" THEN 1030 :REM UP
842 IF K$="M" THEN 1035 :REM DN.
844 IF K$="@" THEN 1040 :REM UP./RT.
846 IF K$="N" THEN 1045 :REM DN./LEFT
848 IF K$="Y" THEN 1050 :REM UP/LEFT
850 IF K$="/" THEN 1055 :REM DN./RT.
860 IF K$=" " THEN RETURN :REM REPEAT
995 GOTO 820        :REM INVALID KEY
```

NOTE: From now on, be especially careful to type the correct line numbers so that later additions will not overwrite incorrectly numbered lines, making these errors hard to find.

Line 82Ø is a dummy REM that will be intentionally overwritten by later additions; it is needed as a target line for GOSUB in line 512Ø.

Line 825 GETs a keystroke and stores it in K$. The two adjacent quote marks signify no character and must be typed with no space in between them. If no character is in K$, that is, if no key has been pressed, then line 825 recycles until you do press a key.

Lines 836–86Ø are all very similar, thereby allowing you to enter them by using lines as templates, as described in Chapter 1. You will have many opportunities to use this procedure in the GRAPHICS KEYS subroutine.

Each of the lines 836–86Ø tests for the exact key shown in the quote marks; if the key is found, the program branches to one of eight possible drawing directions as shown in the accompanying REM statements. Thus, if you press the H key the cursor is going to move left. Now you can see how easy it is to change key assignments. Simply LIST the line you want to change, move the cursor to

the character within quotes, and press any printing key on your keyboard. That key is now assigned to perform the designated function. Don't forget RETURN.

If a key is pressed that is not recognized by any of the IF statements the program will reach line 995, which cycles back for another try.

Before this group of program lines can be tested, SAVE the latest version of DRAWING BOARD and add the following (don't type NEW):

```
1000 REM
1001 REM ** KEY PROCESSING **
1020 MH=+1:MV=+0:RETURN
1025 MH=-1:MV=+0:RETURN
1030 MH=+0:MV=-1:RETURN
1035 MH=+0:MV=+1:RETURN
1040 MH=+1:MV=-1:RETURN
1045 MH=-1:MV=+1:RETURN
1050 MH=-1:MV=-1:RETURN
1055 MH=+1:MV=+1:RETURN
```

These lines do nothing more than set the direction of drawing in lines 5140 and 5145 of main line. Since DH, DV, and RA are all initialized to 1 in line 5025, H and V will be set to +1 or −1 in lines 5140, 5145, that is, one dot position change in any horizontal, vertical, or diagonal direction, as determined by the MH, MV values.

Drawing Straight Lines

You can now RUN the program. You'll see the cursor displayed as before but with no error message messing up the screen.

Place your right hand on the keyboard in the normal typing position, that is, with your index finger on J. Press the H key and watch the cursor on the screen. It will move slightly to the left. Press several times and the cursor will draw a line. Do the same with the Y key and you will get a line sloping up and to the left. The line is jagged, but this is normal, as discussed previously in connection with aspect ratio.

Try all the keys shown in lines 836–850. Key assignments are also given in Table 5 at the end of this chapter. Before you decide a key is not working, make sure you are not overwriting a previously drawn line. The cursor will not be visible in this case.

Since you want to draw while looking at the screen, not at the keys, the key assignments have been made to facilitate *touch drawing.* With your right hand in the normal typing position, key assignment is in the direction of cursor move-

ment so you can move your fingers in the direction you want to draw. The three conveniently located keys J, K, and L have not been ignored; they will be given important assignments shortly.

You can press a key as rapidly as you like, but there is an easier way to draw than by repeated keystrokes. Press the space bar. The last key entry is automatically repeated. Hold down the space bar, and the cursor continues to draw until released. The space bar is one of the Commodore keys that repeats itself automatically. Thus, you can draw simply by holding down the space bar and using the direction keys only when you want to change direction. Remember, you have to release the space bar before pressing a direction key for it to have an effect.

When all keys are performing correctly, SAVE this latest version of the DRAWING BOARD.

Moving and Erasing

The following lines are all that is needed to let you move the cursor without drawing and to erase previously drawn lines:

```
826 A=ASC(K$)                :REM ASCII VAL.
970 REM
971 REM * FUNCTION KEYS *
972 REM  COLOR & TURN DRAW ON & OFF
980 IF A=136 THEN DE=1-DE:GOTO 820
```

RUN the program and draw a short line with the H key. Now press f7 and then the semicolon key (;). Press the space bar to move the cursor continuously. The cursor will move right, erasing the line you have just drawn. When it has come to the end of the line, it will continue to move as long as the space bar is pressed, but it will not draw. This is how you move the cursor without drawing. Notice that the cursor moves more slowly now. This is because it takes two passes through the LOCATE A BIT U/L subroutine to move or erase. The first pass erases the previous dot, and the second pass writes a new dot.

With the space bar released, press f7 again. Now the cursor resumes its drawing activity. The f7 key toggles back and forth between draw on and draw off. You can just press the space bar each time to continue. You only need to press a direction key to change direction.

Changing Key Assignments

Since the function keys are nonprinting keys, we cannot use lines such as 836–850 to find what key was pressed. Instead we must use the numerical value as-

sociated with the key. This value, or number, is specified by the ASCII standard and the numbers are called ASCII codes, as discussed in Chapter 1. Most keys have an ASCII code whether or not they are printed keys. The *Commodore 64 User's Guide* lists the ASCII codes in Appendix F, where you will find f7 with number 136, just as in line 98Ø.

First, in line 826 we place the ASCII code in the variable A by using the BASIC function for doing this, ASC(K$). For all nonprinting keys there is an IF statement to test for the pressing of any key. In line 98Ø the test is for f7. If f7 is pressed, DE is switched. You can see that if DE is Ø, then 1 − DE is 1, and vice versa.

To change the key assignment of a nonprinting key, either change the code, using Appendix F of the *User's Guide,* or, if the new key prints, use any of lines 836–86Ø as a template. To change from a printing to nonprinting key you must convert to the form of line 98Ø. ASCII codes can be used for all accessible keys; print symbols can only be used for some of the keys.

Cursor Steering

In this section we will add a method of drawing that is amusing and challenging. When used in this manner, I prefer to call the cursor, *the mole,* for reasons that will be obvious when you see it burrowing across the screen in colored tunnels.

Add the following lines to DRAWING BOARD:

```
820 IF ML=0 THEN 825 :REM SET FOR MOLE?
822 GET K$:IF K$="" THEN RETURN
824 GOTO 826
869 REM
870 REM * MOLE,ANGLES,CURVES,CIRCLES *
872 IF K$="L" THEN ML=1-ML:GOTO 820
```

When you RUN the program and the cursor appears, press H and then L and watch the cursor move across the screen. Press Y, N, or any other key and the cursor changes direction without stopping. You are steering the cursor in the direction you want it to go. To stop the cursor, press L again. The L key toggles back and forth between the two modes of operation.

Try drawing rectangles, triangles, animals, and faces. For serious drawing, use the L key for speed, the space bar for short distances, and the direction keys for very short distances of only a few dots.

Start the cursor in any direction with the L key, and just let it run. It will bounce off the edges of the DRAWING BOARD. Many interesting patterns

can be formed in this way. You can get more complex patterns by changing .833 in line 2Ø to 1 and other values.

Notice how inexpensively we get cursor steering. Four lines of executable code does it. Line 822 is the key. Remember how we looked for a key entry in line 825? In this case, instead of looping back to the GET statement if no key is pressed, we loop back to main line, which goes ahead with its assigned tasks using the same *unchanged* values of MH, MV, DH, DV, and RA as previously.

COLOR FOR THE DRAWING BOARD

What are graphics without color? With the routines presented in this section you will be able to set the color of the dots (foreground), the border, the screen background, and the character background. Rather than try to explain the difference between the latter two, not an easy task, let's just add the program lines and see how it works. We'll start with dot colors and than add the other three together. Add these lines to DRAWING BOARD:

```
982 REM
983 REM * DOT COLOR *
984 IF VAL(K$)<1 OR VAL(K$)>8 THEN 995
986 C=VAL(K$)-1
990 CG=16*C+C1:MH=+0:MV=+0:RETURN
```

When the cursor comes on the screen press the 2 key. The color of the cursor changes to white, the color printed on the 2 key. Any line drawn by the white cursor will of course, be white. Similarly, any of keys 1–8 can be used to select a color for the dots.

In line 99Ø MH and MV are set to Ø so you can see the new color without having to draw in any direction. This means, however, that if the cursor is set for steering with the L key, it will stop and you must press a direction key to get it started again. If the cursor is set to green it disappears because green is the background color.

Press the 2 key and notice how much is switched to white. Move the cursor one dot position up. Press the 1 key and notice how much is switched to black. Repeat these operations seven times. You will see that the amount of color change varies from one to eight dots. This is because the color in bit mapped graphics is controlled by character position not dot position. If the cursor is just entering a character position, only one dot will change. As the cursor moves through a character position, which is 8 × 8 dots, successively more dots change color until the maximum of eight is reached.

Line 984 filters out the color keys from all others; line 986 then computes the foreground color value, which is simply the key value minus 1. Line 990 puts the foreground color in the high nibble of the CG variable and retains the current background color, C1, in the lower nibble.

For more color control, add the following lines:

```
974 IF A=133 THEN 1070      :REM F1
976 IF A=134 THEN 1110      :REM F3
978 IF A=135 THEN 1094      :REM F5
1068 REM
1069 REM * BORDER *
1070 C0=C0+1:IF C0>15 THEN C0=0
1072 POKE 53280,C0:GOTO 820
1090 REM
1091 REM * CHARACTER BACKGROUND *
1094 C1=C1+1:IF C1>7 THEN C1=0
1098 CG=16*C+C1:MH=+0:MV=+0:RETURN
1100 REM
1101 REM * CLEAR AND CYCLE BACKGROUND *
1110 C1=C1+1:IF C1>7 THEN C1=0
1115 CG=16*C+C1
1120 FOR J=1024 TO 2023:POKE J,CG:NEXT
1125 GOTO 820
```

Now, with the program RUNing, you can change the border color by pressing function key f1. The border color is not affected by switching to bit mapped graphics so you still have the full range of 16 colors that we worked with in previous chapters. Repeated pressing of f1 will cycle through all 16 colors.

By pressing f3 you can cycle through all eight background colors of the graphics color set. This color is stored in the variable, C1, and is the low nibble of the CG variable.

By pressing f5 you can change the background color of only one character area rather than the whole screen. The character area that changes color is the one that the cursor is in. See the above discussion on dot colors for an explanation of the relationship of the cursor position to a character area. If you change the character background and then move the cursor along a line, you will see the cursor push the color in front of it as it moves from one character area to another. This can produce a number of interesting effects. Notice what happens when you draw a diagonal. Color changes on both sides of the cursor's path unless the cursor is located right on the diagonal of the character areas. However, the cursor will only stay on a character area diagonal if the aspect ratio in line 20 is changed to a 1.

Press L and let the cursor run. Now you can see why I call this the mole.
Changing the dot color or background is a good way of finding a lost cursor. The change in color will make the cursor stand out from its surroundings.
Now that you have rather extensive color control, perhaps you would like to be able to control the length of cursor moves.

Setting the Length of Cursor Moves

There is no reason the cursor has to move only one dot position on each move. To allow you to change the length of cursor movements, add these lines to DRAWING BOARD:

```
852 REM
853 REM * CONTROL KEYS *
856 IF K$="+" THEN 1420
858 IF K$="-" THEN 1425
1400 REM
1401 REM * INCR/DECR MOVEMENTS *
1419 REM * PLUS *
1420 RA=RA+1:GOTO 820
1424 REM * MINUS *
1425 RA=RA-1:IF RA<1 THEN RA=1
1430 GOTO 820
```

First, draw a line to make sure your program is working as it should. Then, press the + key four times and continue drawing. The cursor jumps ahead, and all distances will be covered much faster. Press the − key and the distances get shorter. Each pressing of the + or − key changes the distance moved by one dot. However, if the distance reaches the length of one dot position, further pressing of the − key will have no effect.
Set the length small, that of three or four dot positions is good, and then set the mole to work with the L key. As the mole moves along, press the + key a few times and then the − key. The mole changes its pace without even bothering to stop.
So far you have been limited to drawing straight horizontal, vertical, and diagonal lines. Now you will learn how to change the angles of lines and draw arcs.

ANGLES AND ARCS

Back in the 193Øs when TV was being developed, TV engineers would have been quite surprised if they had been told that people in the 198Øs would be

sitting at keyboards drawing lines on the tube. Television technology is poorly suited for this, and consequently we are forced to accept only rough approximations to the smooth straight lines and circles that we can get from simple paper and pencil drawings.

As you now know, lines in computer graphics are not lines at all but a succession of dots. If you try to draw a line at an angle, the number of dots in the horizontal direction will be different from the number in the vertical direction. Figure 5 illustrates this. Line B can be formed from dots that are spaced one dot position apart in both the horizontal and vertical directions. If the horizontal and vertical spacing is equal, as in Figure 5, the line will have a 45 degree slope. But as we have seen in the discussion of aspect ratio, these distances are not equal on the TV screen. The vertical spacing is 0.833 (six-fifths) greater than the horizontal spacing, so line B would have a slope greater than 45 degrees.

Now look at line C. It requires two horizontal dots for each vertical dot. The more a line moves away from the 45 degree angle, the more jagged it becomes and the longer it must be to appear sloping. If line C were only two horizontal dots long, it would appear as a horizontal line with no slope at all. Line D has to move five horizontal dots for one vertical dot. Lines that are exactly horizontal or vertical result in smooth lines on the screen. Understanding these things is not necessary to use the following entries, but it will help make clear the form lines take at various angles.

```
874 IF K$="J" THEN 1220  :REM ROT. CCW
876 IF K$="K" THEN 1230  :REM ROT. CW
878 IF K$="=" THEN 1260  :REM RESTORE
1200 REM
1201 REM * SET ANGLE *
1220 AN=AN+10:IF AN>360 THEN AN=0
1225 GOTO 1235
1230 AN=AN-10:IF AN<0 THEN AN=360
1235 DH=COS(AN*AR):MH=+SGN(DH)
1240 DV=SIN(AN*AR):MV=-SGN(DV)
1245 DH=ABS(DH):DV=ABS(DV)
1250 RETURN
1255 REM
1260 DH=1:DV=1:AN=0:GOTO 820
```

When the cursor is displayed, move it to the right by pressing the semicolon key three times. Now press the + key five times and the J key once. The cursor jumps to a position to the right that is slightly raised. Press the diagonal key, @, three times. The cursor draws a line with a 10 degree slope. Press the J key once again and the @ key five more times. The cursor now draws a line with a

2Ø degree slope. Each pressing of the J key increases the slope 1Ø degrees, rotating the line in the counterclockwise direction.

Press the K key and then the @ key five times. The slope is back to 1Ø degrees. Each pressing of the K key decreases the slope 1Ø degrees, rotating the line in the clockwise direction. Thus, the J and K keys change the slope of all diagonals the same amount.

Press the diagonal Y key five times. It draws a line with the same slope but in the opposite direction. N and / diagonal keys also draw lines with the same slope.

Draw a line to the right with the semicolon key and then down with the M key. The vertical dot separation is reduced relative to the horizontal. This is to be expected since the slope of the diagonals was changed by changing the relative cursor movement in the horizontal and vertical directions.

Press the J key six times, giving the line a slope of 6Ø degrees plus the 1Ø degrees already entered. The cursor rotates to 7Ø degrees; the @ key will now draw a line at 7Ø degrees. Press H five times and then U five times. Now the vertical dot spacing is greater than the horizontal.

To restore the initial settings of 45 degree slope and Ø degree angle of the diagonals, press the = key. Verify that initial conditions have been restored by drawing with some of the direction keys.

To draw circles, press J or K repeatedly. The radius of the circle can be set by the + and − keys.

Now that you know how to control the cursor with the J and K keys, let's do some cursor steering. With the cursor at least halfway up the screen, set the cursor moving by pressing the M and then the L keys. As the cursor moves, rapidly press the K key. The cursor will describe an arc. If you press at the proper rate, you can draw circles, spirals, and other shapes. You can switch between the J and K keys whenever you want; if you're quick enough, you can change the curvature of arcs with the + and − keys.

Clear the screen with RUN/STOP–RESTORE keys. Then enter RUN. Sorry, with BASIC there is no faster way to clear the bit mapped screen for this task. Start, as before, by pressing the semicolon key three times, but don't press the + key. Press J and then the @ key several times. Nothing happens. The line is too short. Recall our previous discussion about how the length of line affects the slope. Press the + and @ keys repeatedly. At first the dot spacing is so short the cursor will only move horizontally. Then it moves up at a steeper slope than 1Ø degrees, and finally it settles at a 1Ø degree slope.

This behavior can sometimes get you into situations in which the cursor seems to be stuck, for example, if by mistake you press the J and K key when the angle is set at Ø. Pressing the + key a few times will usually clear this up. Or, you can always press the = key to return to normal.

As you have seen, when the J or K keys are pressed the cursor actually draws

the next point. This is good for drawing arcs and circles, but it is not good for drawing a line at some desired angle. To do the latter, turn on the move mode with the f7 key. Set the desired angle with the J or K key. Move the cursor to the point at which you want the line to start. Then turn the draw mode back on by pressing f7 again, and draw the line. If you plan to draw many such lines, an easier way is to change line 125Ø is as follows:

```
1250 GOTO 820
```

This will bypass the writing of the cursor position, but you will have to keep track of what angle you are at.

If you followed the discussion of aspect ratio earlier, you will recognize that changing angles is similar to changing aspect ratio. The difference is that the aspect ratio setting in line 2Ø applies in all cases, whereas the angle setting is only for lines and is in addition to the setting in line 2Ø.

In lines 1235 and 124Ø the conversion factor is applied to AN (angle variable) to convert from degrees, which is most natural for us to work with, to radians, which the computer must work with. The signs are stored separately in MH and MV, so the whole numbers will be the same for plus and minus values. MV gets a negative sign because V increases from top to bottom.

DRAWING CIRCLES

The routine given below will enable you to draw circles just by pressing the C key. The dot spacing of the circles will not increase with the radius as it did in the previous section.

> **NOTE: The two circumflex characters, ^, in line 134Ø stand for raising RD and Y to the power of 2. They should be entered on the Commodore keyboard by pressing the up arrow key.**

```
890 IF K$="C" THEN 1320  :REM CIRCLE
1300 REM
1301 REM * DRAW CIRCLE *
1320 FE=DE               :REM DELETE?
1325 SH=H:SV=V           :REM SAVE PARMS
1330 RD=RA*6:X=RD:Y=0
1335 FOR Y=0 TO RD/SQR(2)
1340 X=SQR(RD^2-Y^2)
1345 FOR J=1 TO 2
```

```
1350 H=SH+X:V=SV+Y:GOSUB 20
1355 H=SH-X:V=SV+Y:GOSUB 20
1360 H=SH+X:V=SV-Y:GOSUB 20
1365 H=SH-X:V=SV-Y:GOSUB 20
1370 T=X:X=Y:Y=T          :REM SWAP X,Y
1375 NEXTJ:NEXT Y
1380 H=SH:V=SV:GOTO 820
```

After the above lines are added, get the cursor on the screen and press the C key. A small circle is drawn with the cursor at the center. Press the + key four times and then the C key again. A larger concentric circle is drawn. Press the − key twice and the C key again. A smaller concentric circle is drawn. Each pressing of the + or − key changes the radius by six dot spaces. This is determined in line 1330 by the multiplier 6 of RA. You can change this if you like.

That's all there is to it. You can draw circles anywhere on the screen simply by moving the cursor to the desired center position, setting the radius, and then pressing C. If you make a mistake, use the f7 key to erase the circle just as you did lines. Try it by pressing f7 and then C.

The routine for drawing circles consists of two nested FOR. . .NEXT loops. The outer loop uses the Pythagorean theorem to compute X for steps in Y that are increased by one dot space each. The loop stops when the angle reaches 45 degrees. All the remaining points are drawn in the inner loop using the symmetry of the circle. The inner loop is run twice, one time drawing all points symmetric with the computed point and a second time with X and Y values swapped to use the symmetry about the 45 degree diagonals.

Note that DRAW CIRCLE does not RETURN to main line but calls Locate a Bit U/L; then processing returns to GRAPHICS KEYS to get the next key entry.

TABLE 5 KEY ASSIGNMENTS FOR THE ELECTRONIC DRAWING BOARD

Type	Key	Operation
Direction		
	;	Move right
	H	Move left
	U	Move up
	M	Move down
	@	Move up and right
	Y	Move up and left
	/	Move down and right
	N	Move down and left
Control		
	+	Increment spacing or radius
	−	Decrement spacing or radius
	Space bar	Repeat last operation
	L	Continuous move
Angles and circles		
	J	Rotate counter clockwise
	K	Rotate clockwise
	=	Restore slope to 45 degrees
	C	Draw a circle
Color		
	f1	Border: cycle Ø−15
	f3	Background: cycle Ø−7
	f5	Character: cycle Ø−7
	1−8 keys	Dot color as per color label on the key
Move or erase		
	f7	Toggle between the draw mode and the move/erase mode

7

THE WORLD OF SPRITES

SPRITE 1. a spirit; a shade; an apparition or ghost
2. an elf, pixie, fairy or goblin

—Webster's Unabridged Dictionary

Either of the above definitions can be applied to *computer sprites*. They are the weird figures, people, animals, or what have you that populate the screens of arcade-type games. Sprites can be used in many ways, and the following programs will give you the tools to do so.

Computer sprites are made up of dots within a prescribed rectangular area. By turning individual dots on or off, figures can be created within the rectangle. You can draw faces, missiles, letters, animals, patterns—anything that can be represented within the rectangular area. Once entered in the computer's sprite memory, the sprite can be moved about the screen, moved over or under other sprites, turned on and off; its color can be changed; collisions with other sprites can be detected and acted on. Up to eight sprites can be displayed simultaneously, and many more can be stored in memory. Sprites give animation and variety to your displays.

When you were working with the DRAWING BOARD program, you no doubt found it to be rather slow compared to commercial programs. This is because all the DRAWING BOARD operations were performed using the BASIC language. Sprite operations, on the other hand, are performed with special circuitry built into the computer. BASIC is still used to control these circuits, but with sprites, 5ØØ dots can be moved across the screen faster than one dot can move on the DRAWING BOARD.

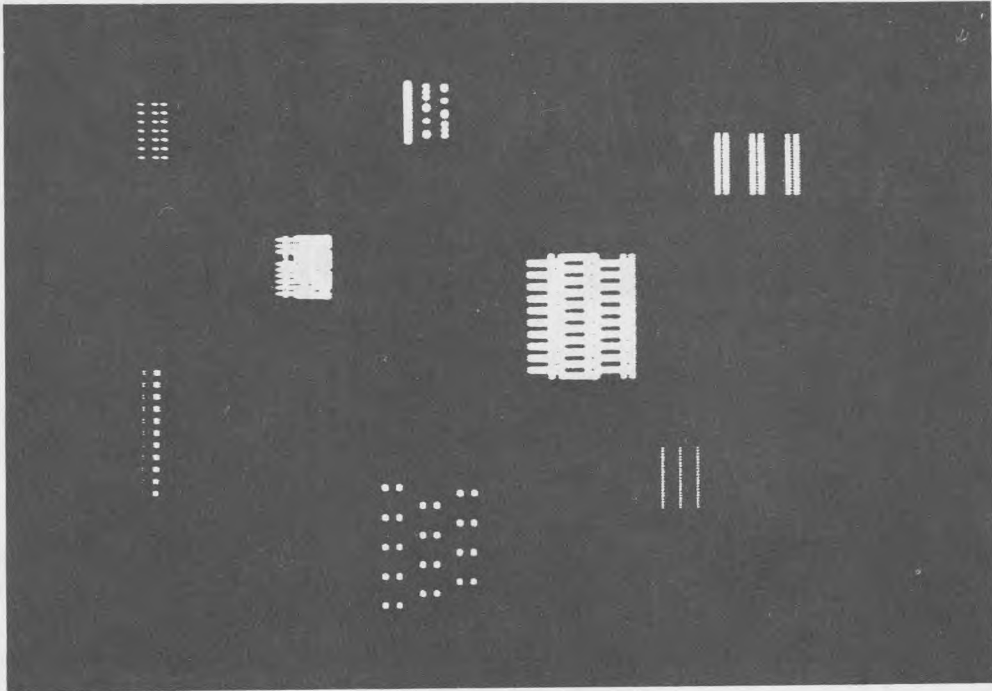


Photo 6 Abstract forms created in SPRITE ART float across the screen and interact visually with one another.

SPRITES THE EASY WAY

Each sprite is made up of 5Ø4 dots. There are eight dots (bits) per character so a sprite requires 63 characters. To see all eight sprites available on the Commodore 64, you would have to enter 5Ø4 characters! Nobody wants this kind of task. If you did the sprite demo program of the balloon in the *Commodore 64 User's Guide*, your appetite for sprites is probably whetted, but the entry of all those numbers!

Rather than entering myriads of numbers to define sprites why not let the computer do the work? This is the approach we will take in the following programs. Then, instead of an unpleasant task, you will probably find sprite drawing one of the most enjoyable and challenging uses for your personal computer.

Three sprite programs are presented below that will make it easy to draw and use sprites. The same methods can easily be transferred to your own programs to create games, computer art, computer stories, or any other projects your imagination can devise.

The first program introduces you to sprites and tells you how to use them. The sprites will be interesting patterns drawn by the computer. Although this

first program is enjoyable in its own right, it is a template program that will be used to build the next two.

The second sprite program will let you draw and display up to eight sprites on your screen. You can SAVE your sprite creations for use in other programs, and you can retrieve previously SAVEd sprites to make changes. Although primarily a utility program, you will find that the displays are interesting in themselves.

The third sprite program combines much of what has gone before. You will be able to create a background with the DRAWING BOARD program, add sprites of our own creation, move the sprites about the screen, change colors, and listen to the sprites as they crash, touch, or kiss.

Good news! Sprites come cheap. You can use the DRAWING BOARD program as a template for sprite programs. Even though sprite graphics use different facilities of the computer than dot graphics, many of the operations are the same. With cut-and-paste techniques a few changes and additions will transform your drawing board to a sprite theater. If you didn't enter the DRAWING BOARD program, you can copy the sprite programs directly from the listings in Appendix B. The following descriptions will assume that you are cutting and pasting the DRAWING BOARD program.

SPRITE ART

The first sprite program is called SPRITE ART because of the artistic possibilities it presents. Serious computer artists work with expensive, sophisticated computer equipment. Some use a cursorlike marker as a paint brush. Quite striking results have been obtained with this technique. However, you have learned in working with the DRAWING BOARD that inexpensive computers that use TV type displays are not good at emulating conventionally drawn art. Instead, why not take advantage of what the computer does best—the creation of patterns and movement. Sprites can move past one another to create shifting patterns of color. Eight sprites can be displayed simultaneously, each with a pattern of your choice. Here is a new artistic medium worth investigating.

DISPLAY EIGHT SPRITES

LOAD the DRAWING BOARD program into your computer. Protect DRAWING BOARD by changing to a working disk or tape. Then make the following entries:

```
5021 S0=2040:U=53248  :REM BASE LOC.
5022 H=30:V=55:C=5:LC=12288:RA=1
```

```
5023 P1=1:P3=1:DH=1:DV=1
5045 FOR J=0 TO 7     :REM POINTERS
5050 POKE U+39+J,C+J:POKE S0+J,192+J
5055 POKE U+1+2*J,V:POKE U+2*J,H+J*30
5060 NEXTJ
5065 POKE U+21,255    :REM SPRITES ON
5068 REM
5069 REM * SET ALL DOTS ON *
5070 FOR J=0 TO 7:FOR K=0 TO 62
5072 POKE LC+J*64+K,255:NEXT K:NEXT J
```

LIST line 5Ø21 and change it to 5Ø2Ø. You will temporarily have two identical lines, since changing a line's number does not delete the previous line, it only adds a new line.

To name your new program and to remove unwanted lines enter

```
1 REM ***** SPRITE ART *****
5021
5025
5040
```

Since you do not need circle processing, very carefully delete lines 87Ø and 89Ø and lines 13ØØ–138Ø.

Save your handiwork with the file name SPRITE ART. When you RUN the program seven squares made up of wavy patterns will appear along the top row. Starting with the leftmost square, a solid color is painted over the patterns. Each square has a different color. There are really eight squares, but the second is background blue and therefore invisible. By pressing function key f1, you will be able to change the border color.

If your program does not work as described, check your entries against the following listing of INITIALIZE GRAPHICS:

```
5010 REM
5011 REM ** INITIALIZE GRAPHICS **
5020 S0=2040:U=53248   :REM BASE LOC.
5022 H=30:V=55:C=5:LC=12288:RA=1
5023 P1=1:P3=1:DH=1:DV=1
5030 AR=3.14159265/180 :REM RADIAN/DEG
5035 PRINT CHR$(147):J$="          "
5045 FOR J=0 TO 7     :REM POINTERS
5050 POKE U+39+J,C+J:POKE S0+J,192+J
```

```
5055 POKE U+1+2*J,V:POKE U+2*J,H+J*30
5060 NEXTJ
5065 POKE U+21,255   :REM SPRITES ON
5068 REM
5069 REM * SET ALL DOTS ON *
5070 FOR J=0 TO 7:FOR K=0 TO 62
5072 POKE LC+J*64+K,255:NEXT K:NEXT J
```

As you have probably guessed, the eight squares are eight sprites who will do your bidding if only you give them the proper instructions. You know from previous programs that these instructions must specify color and screen position. For SPRITE ART you will want to add instructions to control size and pattern. Before entering the program lines that will provide these features, let's learn a little about how sprites work.

SPRITE GRAPHICS

Figure 6 shows how sprites are painted onto your TV screen. Four basic items are involved in the display of sprites. There is the *sprite* itself, that is, the pattern of 1's and Ø's that determine the sprite figure. Figure 6 shows two sprites for illustration—sprites 1 and 6. Sprites may be stored at any memory location that is a multiple of 64. Two other sprite items are *position* and *color*. Both of these are entered in sprite locations high up in memory. Finally, there is the *background*, that is, what you see on the screen at all of the positions at which there is no sprite. This information usually comes from screen memory. It can also come from bit mapped memory (not shown in Figure 6).

Sprites have memory locations reserved for their control just as sound and bit mapped graphics have memory locations reserved. Each of the eight sprites has its own memory locations for control. Sprite color and position are controlled by POKEing appropriate values into these locations.

Sprite memory locations start at 53248; therefore, in line 5Ø2Ø, we set the base value, U, to 53248 and relate all higher locations to this value. In line 5Ø55 the horizontal and vertical positions, H and V, are POKEd into memory locations starting at U and U + 1, respectively. The FOR. . .NEXT loop steps through succeeding locations, two per sprite. The V variable is constant, which places all sprites on the same line. The H variable is incremented by J*3Ø so that each sprite is to the right of its predecessor.

In line 5Ø5Ø the color, C, is POKEd into color control locations starting at U + 39. By adding J to C, each sprite is given a different color.

The mapping of sprites onto the screen from memory is much simpler for sprite graphics than for bit mapped graphics. Sprites are composed of 21 rows of

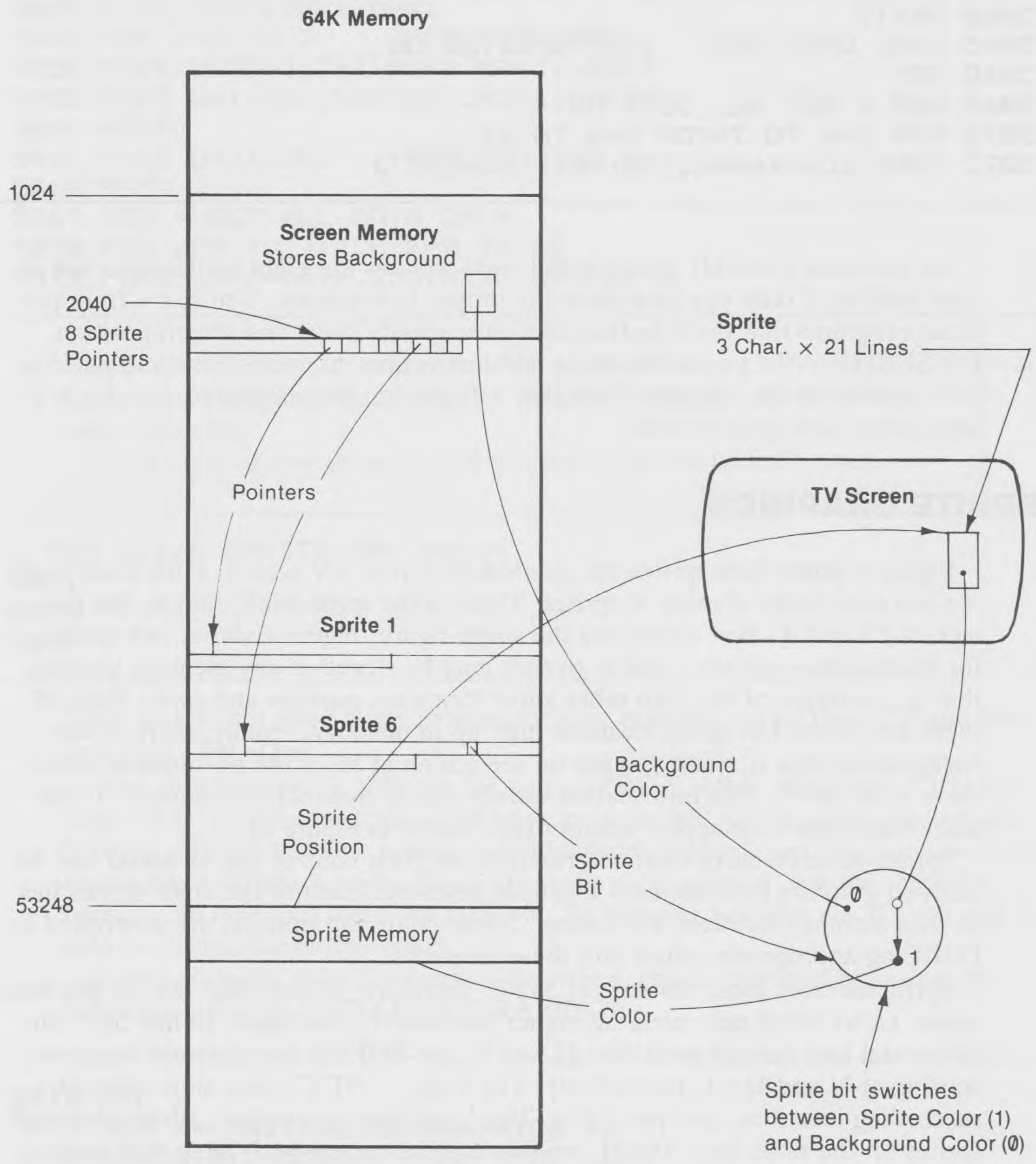


Figure 6. Sprite Graphics.

dots. Each row has 24 dots. Each memory location can store one character, and each character can store eight dots (8 bits). Therefore, each row of a sprite contains three characters. Since there are 21 such rows, a sprite requires 63 characters, that is, 63 memory locations. These locations can be any place in memory where they are not likely to be overwritten by programs or other data. In line 5Ø22 we set the start of the sprites at location 12288, with LC as the base location variable.

To summarize storage of sprite information, all sprite controls are located at fixed locations starting high up in memory at 53248. The actual sprite design that is displayed on the screen is stored lower down at any convenient location. How does the computer know where this is? We must let it know by POKEing pointers to the location of the sprite in memory (not its screen position) into an area starting at 2Ø4Ø, which is reserved just for this purpose. To do this, in line 5Ø2Ø we set another base variable, SØ, to 2Ø4Ø; then in line 5Ø5Ø, we do the necessary POKEing. But the computer doesn't want the actual memory location, it wants only multiples of 64. So we divide sprite location 12288 by 64, come up with 192, and POKE that into 2Ø4Ø. The FOR. . .NEXT loop takes care of all eight sprites.

If a sprite dot is set to Ø with a POKE to its memory location, it will appear to be transparent, that is, the background will be displayed. If it is set to 1, it is displayed in the color POKEd into memory locations starting at U+39, as we did in line 5Ø3Ø. If that color is the same as the background, the dot will appear to vanish as did the dots in sprite 2.

When the program is RUN, you first see the sprites displayed with whatever values are in memory. When the program reaches line 5Ø7Ø, a FOR. . .NEXT loop POKEs 255 into all sprites. The number 255 is the largest that can be entered into a memory location (1 byte), which has the effect of setting all dots to 1's and the sprite display to solid squares of color.

The transparent mode, that is, when a sprite dot is set to a Ø, is most interesting. Background images or other sprites can be seen through openings in a sprite as it moves across the screen.

Sprites require only 63 characters. But in line 5Ø72 the base location is incremented by 64 for each sprite, because the computer used the 64th character for housekeeping functions; thus, the sprites must have at least 64 characters of separation.

The sprite's position on the screen is set by specifying the horizontal and vertical position of the upper left-hand corner of the sprite. Sprites can be moved to invisible locations off the screen. Consequently, sprite locations start and end off the screen, as shown in Figure 7. As you can see, our initial values of H=3Ø, V=55 place the upper left-hand corner of the first sprite just inside the visible sprite area.

Sprites can be turned off and on. This is done in a single memory location at

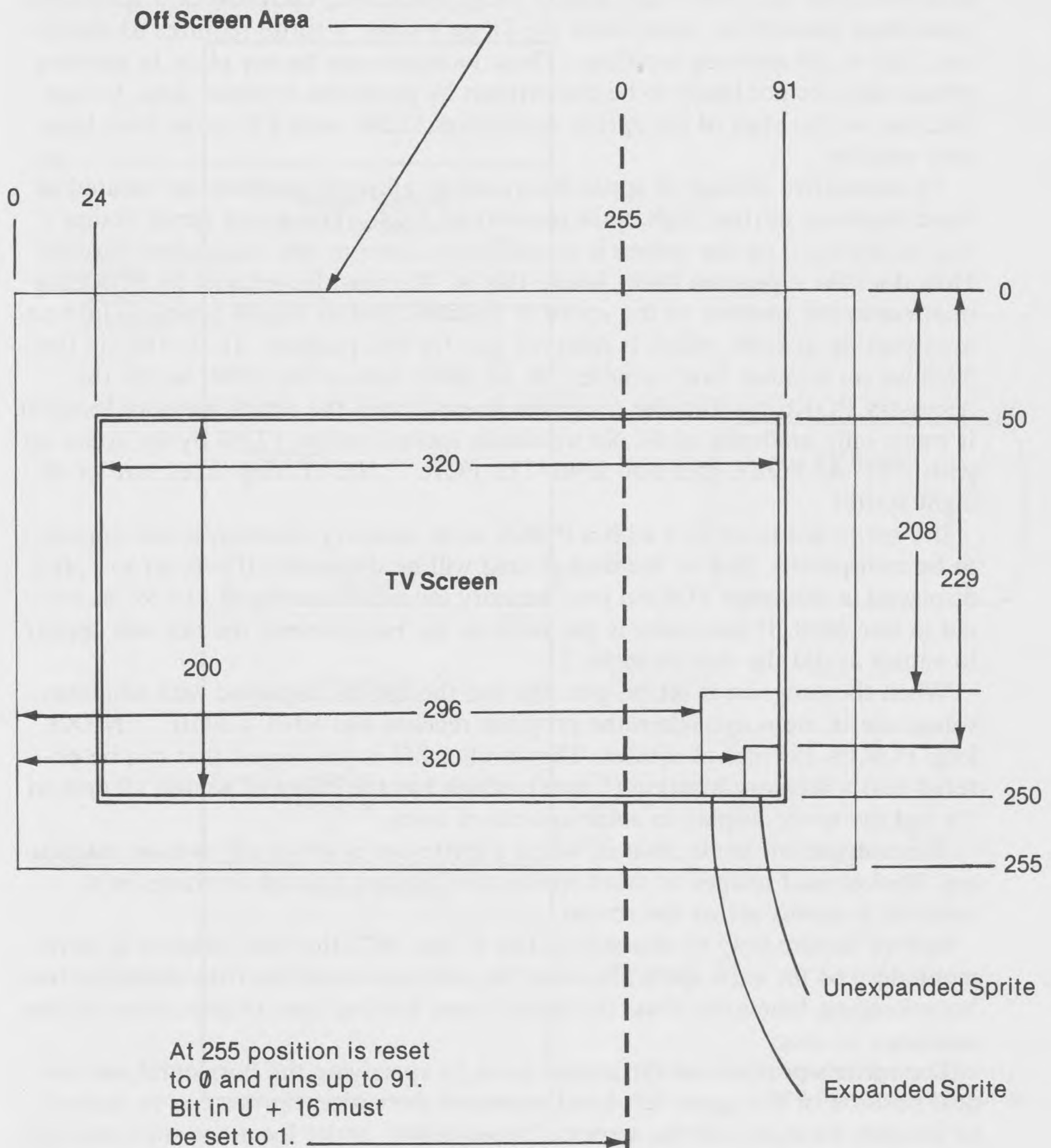


Figure 7. Sprite Positioning. (Not to scale.)

U + 21. Each sprite has 1 bit in this location that is reserved for turning it off and on. In line 5Ø65, by POKEing 255, location U + 21 is set to all 1's and all sprites are turned on.

We will now add program lines to SPRITE ART that will let you change, under keyboard control, the sprite variables we have been discussing.

Setting Sprite Colors

LIST lines 1Ø9Ø–1125. Now make changes, deletions, and additions so that your program looks like the following (line 112Ø gets deleted):

```
976 IF A=134 THEN 1094      :REM F3
978 IF A=135 THEN 1110      :REM F5
1090 REM
1091 REM * BACKGROUND *
1094 C1=C1+1:IF C1>15 THEN C1=0
1098 POKE 53281,C1:GOTO 820
1100 REM
1101 REM * SPRITE COLOR *
1105 C=PEEK(U+39+SP) AND 15
1110 C=C+1:IF C>15 THEN C=0
1115 POKE U+39+SP,C
1125 GOTO 820
```

SAVE it and RUN it. If you haven't turned off your computer since the last time you ran SPRITE ART, the sprites are immediately displayed as solid squares. The computer remembers the last sprite entry! Nevertheless, it's not so smart because it goes ahead and paints in solid colors even though they're already there. This means that you must sit through this useless process before you can make key entries; but it only takes a few seconds.

Press f5 several times. Sprite 1 cycles through all 15 possible colors. Press f3. The background cycles through all 15 colors. Location 53281 in line 1Ø98 is our old friend for controlling background color.

Sprite color goes into locations starting at U + 39. To get into the right location, we add the sprite number SP in line 11Ø5. Selection of sprites will be the subject of the next section.

Selecting Sprites

Since we want to be able to work with all eight sprites, we enter the following lines:

NOTE: In line 988 the circumflex, ^, prints, instead of the up arrow that is used on the Commodore 64 to designate the raising of a number to a power. In line 988 and in those that follow later, press the up arrow on your keyboard in place of the circumflex (^).

```
982 REM
983 REM * SPRITE SELECTION *
984 IF VAL(K$)<1 OR VAL(K$)>8 THEN 995
986 SP=VAL(K$)-1:J=PEEK(U+21) :REM TURN
988 POKE U+21,J OR 2^SP
990 V=PEEK(U+1+2*SP):HL=PEEK(U+2*SP)
992 HH=PEEK(U+16) AND 2^SP:H=HL+256*HH
995 GOTO 820       :REM INVALID KEY
```

Some of these lines replace DRAWING BOARD lines that we don't want.

When you RUN the program you will find that by pressing any of keys 1–8, you can select the corresponding sprite. Test this for each sprite by changing its color with f5.

Line 984 checks for a valid key entry. If valid, in line 986 the sprite number SP is set to the value of the key entry minus 1. K$ is a string that can't be used in calculations. Since we will often use the sprite number to calculate memory locations, we convert the string variable K$ to a numeric value with the VAL functions in lines 984–986. Once in the computer, it is more convenient to work with values starting at Ø, so we subtract 1 from the value entered.

Lines 99Ø–992 pick up the current position of the selected sprite on the screen, so that when you press a direction key the newly selected sprite will be the one to move. You will see how to move your sprites in the next section.

How to Move Sprites

The screen position of the upper left-hand corner of each sprite is stored in locations starting at U=53248. To move a sprite it is only necessary to POKE new values in the location corresponding to the sprite you want to move. This is quite a different method from that used to move about in bit mapped graphics. Accordingly, we will have to modify our locate subroutine to work with sprites. The following lines will do it:

```
12 REM ** LOCATE A SPRITE **
20 IF H<=024 THEN H=024:MH=-MH
```

```
25
30 IF H>=HS  THEN H=HS :MH=-MH
35 IF V<=050 THEN V=050:MV=-MV
40 IF V>=VS  THEN V=VS :MV=-MV
45 HL=H-256*HH
50 REM
55 REM * PROCESS HH BIT SET/RESET *
60 IF HH=0 AND H<256 THEN 105
65 IF HH=1 AND H>255 THEN 105
70 IF HH=1 THEN 90
75 HH=1:HL=0
80 POKE U+16,PEEK(U+16) OR 2^SP
85 GOTO 105
90 HH=0:HL=255
95 POKE U+16,PEEK(U+16) AND 255-2^SP
100 REM
105 POKE U+1+2*SP,V:POKE U+2*SP,HL
110 RETURN
5125 HS=320-H(SP)*24:VS=229-V(SP)*21
5130
5135
```

Note that line 25 is first moved to line 2Ø; line 25 is then zapped. Slight modifications are also required to main line. Line 5125 gets new data and lines 513Ø–5135 are zapped.

When you RUN your program now, you will find that your direction keys and your + and − speed control work just like they did with the DRAWING BOARD. Start sprite 1 in a diagonal direction and press the L key. The sprite goes floating across the screen bouncing gently off the walls. Press the + key and the pace quickens. Give the J key a few taps and the sprite changes angle.

> **NOTE: Remember when using the J and K keys, if the + and − settings are too low the computer can't cope. Give it a sufficient angle to make its move. If your sprite gets stalled, the = key will reset angles to normal.**

Press any of keys 1–8 and a new sprite starts an excursion about the screen. As the sprites fly about, notice how sometimes a sprite will pass in front of another sprite and sometimes it will pass in back of another sprite. There is a definite rule for this: Lower numbered sprites pass in front of higher numbered sprites. Sprites are numbered from their starting positions from left to right and

from 1–8. The sprite on the far left is sprite 1, the sprite next to the right is sprite 2, and so on.

The lines you have just entered give you all kinds of possibilities for working with sprites. Lines 2Ø–4Ø are the familiar border conditions we have been using to reflect objects off the walls of our two-dimensional stage. We had to modify them to work with sprites, but as you can see from the sprites behavior, the job they do is the same. These are the lines you will change if you want sprites to glide off the screen, wrap around from one side of the screen to the other, collide with invisible barriers placed on or off the screen, and so on. We are not going to pursue all of these areas, but feel free to make changes in these lines on your own. To help you do this, here is an explanation of what is going on behind the scenes as the sprites move about.

Lines 2Ø and 35 simply set up reflective boundaries at the edges of the visible screen area, as shown in Figure 7-1. In lines 3Ø and 4Ø we had to enter some new variables to take care of the fact that sprites can change size. We will be changing sprite sizes shortly, but for now imagine that the sprite has increased in the horizontal direction to the right. Clearly, the upper left-hand corner will be farther left when the right side of the sprite hits a right boundary. The same applies for vertical changes. It is this change in position of the upper left-hand corner at the boundary for which variables HS and VS provide. They are set in line 5125 of the main line routine.

The horizontal and vertical sizes of the sprites are stored in array variables H(SP) and V(SP). The sprite has two sizes, *expanded* and *unexpanded*. When expanded the sprite's dimension is doubled. The horizontal dimension increases from 24 to 48 dots; the vertical dimension increase from 21 to 42 dots. These are represented by 1 or Ø being set in the corresponding array element for each sprite. As can be seen from line 5125, if H(SP) is Ø, HS is 32Ø, and if H(SP) is 1, HS is 296. Thus, H(SP) is like a switch, switching HS between two values having a difference of 24 dots, the exact amount needed to change the position at which the sprite will be reflected. The vertical dimension is adjusted in a similar manner.

From all this, you can see that to change boundaries all that is needed is to change the numbers 24, 5Ø, 32Ø, and 229 in lines 2Ø, 35, and 5125.

So much for the placement of boundaries. What happens when a boundary is hit is determined by the THEN clause of the IF statements in lines 2Ø–4Ø. Here you can make the sprite jump across the screen, change color, disappear, expand, contract, and so on.

Lines 5Ø–95 involve a technical detail that can be skipped if you're not interested in the specifics of how sprites are positioned. The sprites can be positioned horizontally at any one of 344 points. But the VIC II chip that provides sprite control has position registers of only 1 byte, hence a capacity for positioning only 255 points. The solution? Store a high bit in another register shared by all

sprites and let the programmer take care of setting and resetting the high bit. That is exactly what is done in lines 55–1Ø5. Line 6Ø tests for moving from 255 to 266 and line 65 tests for moving from 266 to 255. If this boundary is crossed, the high bit stored at U+16 is either set in line 8Ø or reset in line 95, depending on the direction of motion.

HL stores the lower 4 bits of H and ranges from Ø to 255. HH stores the fifth bit of H and ranges from Ø to 1. After HL and HH are determined and the high bit is set to Ø or 1, V and HH are POKEd into sprite registers U+1+2*SP and U+2*SP, respectively.

Patterns for Sprites

It is now time to give some variety to our sprites. There are many ways that this can be done, and we will show you a few of them. Try the following lines and see what you can come up with:

```
1600 REM
1601 REM ** CHANGE PATTERNS **
1610 FOR J=0 TO 62       :REM CLEAR
1615 POKE LC+64*SP+J,0:NEXT:GOTO 820
1619 REM
1620 IF P1>7 THEN P1=0    :REM CHANGE
1622 IF P3>9 THEN P3=1
1623 IF 2^P1+P2>255 THEN P2=0
1625 FOR J=0 TO 62 STEP P3
1630 POKE LC+64*SP+J,2^P1+P2
1635 NEXT:GOTO 820
```

The above lines are activated from your keyboard by the keys, A, S, D, and F, so add the following:

```
861 IF K$="F" THEN P1=P1+RA:GOTO 1620
862 IF K$="D" THEN P2=P2+RA:GOTO 1620
863 IF K$="S" THEN P3=P3+RA:GOTO 1620
864 IF K$="A" THEN            1610
```

RUN the program and wait a few seconds for the initialize routines to finish. Then press the F key. Sprite 1 changes to three vertical bars. Press F again. The three bars shift left. The D key also produces bars, but the separation and width of the bars change. Press the S key. Nothing may happen, in which case press

the A key and then the S key again. The A clears the sprite for the S, D, and F keys. The S key produces short horizontal bars of various configurations. Some look like a school of fish or a squadron of missiles. As you continue pressing the S, D, and F keys to make pattern changes, these keys change what they do. For example, instead of making vertical bars, the F key may make short horizontal bars.

Try different combinations of keys. The pattern may not change much for a few keystrokes, and then it may suddenly shift to a new design. Use the A key to start on a new sequence. The + and − keys also affect the patterns produced. If you change a pattern while a sprite is in flight, the sprite will stop momentarily for the change and then resume its flight.

These patterns are created by numbers generated in lines 861–863, where variables P1, P2, and P3 are incremented by RA. The magnitude of RA is controlled by the + and − keys. As a result, with each keystroke you have control over the size of the P variables and their rate of change.

In line 163Ø, P1, set by the F key, raises 2 to a power, which has the effect of determining individual bits of a character. P2 is added to the result, generating a number for POKEing into the sprite. To prevent simple repetitions, P3, set by the S key, varies the steps by which the P1/P2 combo is POKEd into the sprite.

Expanding Sprites

To control the size of your sprites from the keyboard, enter the following:

```
866 IF K$="T" THEN 1710   :REM EXPAND
868 IF K$="G" THEN 1750
1708 REM
1709 REM * EXPAND VERTICALY *
1710 V(SP)=1-V(SP):IF V(SP)=0 THEN 1725
1715 POKE U+23,PEEK(U+23) OR 2^SP
1720 RETURN
1725 POKE U+23,PEEK(U+23) AND 255-2^SP
1730 RETURN
1748 REM
1749 REM * EXPAND HORIZONTALY *
1750 H(SP)=1-H(SP):IF H(SP)=0 THEN 1765
1755 POKE U+29,PEEK(U+29)  OR 2^SP
1760 RETURN
1765 POKE U+29,PEEK(U+29) AND 255-2^SP
1770 RETURN
```

The G key expands the sprite horizontally, and the T key expands it vertically. These keys were selected simply because they are easy to use with the left index finger in the direction in which they expand the sprites. If you have other ideas, you can easily change these key assignments, as you can all others, by placing your own choices in lines 866–868.

Expand one of the sprites in both directions. Make a pattern with the pattern keys, and notice how much more detail can be seen with the expanded sprite.

Horizontal expansion is controlled by location U+29; vertical expansion by U+23. These locations are shared by all sprites, so we use the now familiar AND and OR operators in lines 1715, 1725, 1755, and 1765 to turn individual bits on and off at these locations.

Turning Sprites On and Off

If you want to turn individual sprites on and off, add the following lines:

```
865 IF K$="Z" THEN 1650
1648 REM
1649 REM * TURN SPRITE OFF & ON *
1650 J=PEEK(U+21)
1655 IF (J AND 2^SP)=0 THEN 1665
1660 POKE U+21,J AND 255-2^SP:GOTO 820
1665 POKE U+21,J OR      2^SP:GOTO 820
```

By pressing the Z key you can zap the selected sprite. Press the Z key again and the sprite reappears. A zapped sprite will also reappear if it is selected with any of selection keys 1–8.

SPRITE ART will be used as a template in Chapters 8 and 9 so be sure to SAVE it out of harm's way.

TABLE 6 KEY ASSIGNMENTS FOR SPRITE ART

Type	Key	Operation
Direction		
	;	Move right
	H	Move left
	U	Move up
	M	Move down
	@	Move up and right
	Y	Move up and left

TABLE 6 KEY ASSIGNMENTS FOR SPRITE ART (Cont.)

Type	Key	Operation
	/	Move down and right
	N	Move down and left
Control		
	+	Increment speed or pattern steps
	–	Decrement speed or pattern steps
	Space bar	Repeat last operation
	F	Change sprite pattern by bit
	D	Change sprite pattern by character
	S	Change sprite pattern by position
	A	Clear sprite pattern
	Z	Toggle sprite off/on
	T	Toggle sprite vertical expand/contract
	G	Toggle sprite horizontal expand/contract
Angles		
	L	Continuous move
	J	Rotate counter clockwise
	K	Rotate clockwise
	=	Restore slope to 45 degrees

NOTE: If sprites don't move, press =.

Type	Key	Operation
Color		
	f1	Border: cycle Ø–15
	f3	Background: cycle Ø–15
	f5	Sprite: cycle 0–15
Select sprite		
	1–8	Select sprite by number, counting left to right

8

Sprites from Your Keyboard

Let your fingers do the drawing! In this chapter you will learn how to draw sprites the fun way—using your computer keyboard. You can display up to eight of your sprite creations at a time, save them for entry into other programs, and recall them for changes. Sprite symmetries are drawn automatically, and with one keystroke you can reverse background and foreground colors.

We'll call our sprite drawing program SPRITE STUDIO, and we'll use SPRITE ART as a template to create it. With some changes and additions to SPRITE ART, you will soon be drawing sprites on your screen.

SPRITE STUDIO

With SPRITE ART safely backed up on another disk or tape, LOAD a copy into your computer. First let's shorten the program. All the lines from 1200 to 1635 deal with angles, patterns, and increments, none of which we will need in the STUDIO. Delete them and also delete lines 856, 858, and 878.

Enter

```
LIST 5011-5072
```

Lines 5023, 5030, 5060, and 5069 are not needed so they can be deleted too. Make changes to the remaining lines so that they correspond to the following:

```
5011 REM ** INITIALIZE SPRITES **
5020 S0=2040:U=53248  :REM BASE LOC.
```

```
5022 H=48:V=125:C=1:LC=12288:HL=H
5035 PRINT CHR$(147):J$="          "
5045 FOR J=0 TO 7     :REM POINTERS
5050 POKE U+39+J,C:POKE S0+J,192+J
5055 POKE U+1+2*J,V:POKE U+2*J,HL
5065 POKE U+21,255    :REM SPRITES ON
5068 REM
5070 FOR K=0 TO 62
5072 POKE LC+J*64+K,255:NEXT K:NEXT J
5090 POKE U+21,255:POKE LC+31,247
```

Since this initialization routine can be used as a template for your other sprite programs, it has its own title in line 5Ø11. Use this title to SAVE the routine as a separate template, although the last two letters, ES, will be truncated since a maximum of 16 characters are accepted for file names.

Line 5Ø22 has four changes, with the variable HL replacing RA. In line 5Ø5Ø

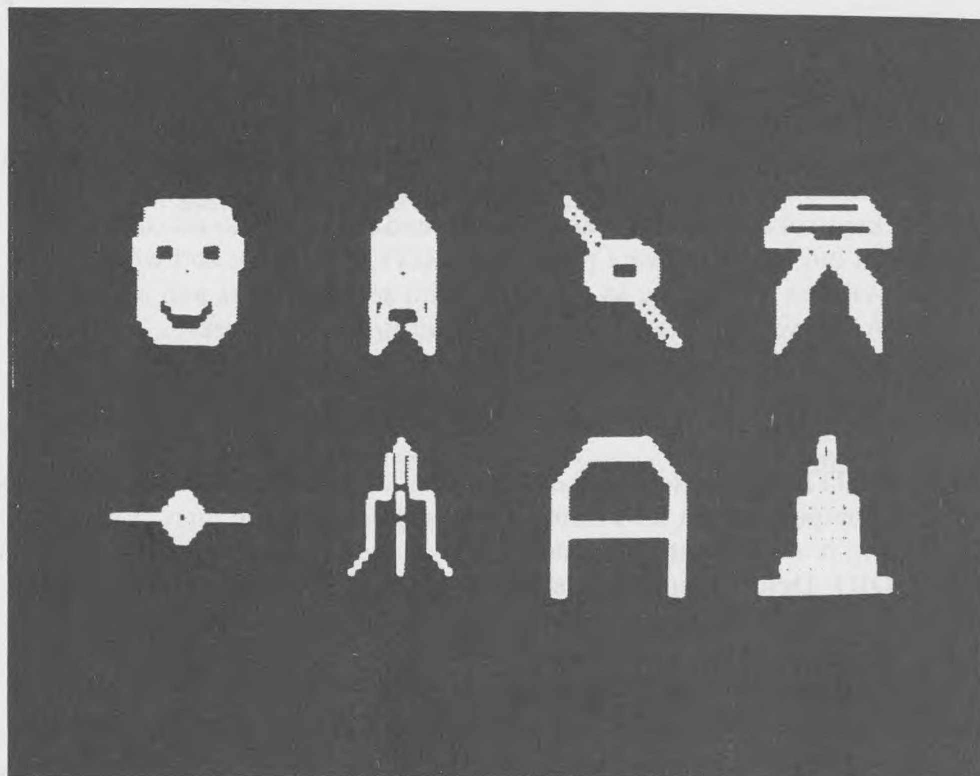


Photo 7 These eight figures show the wide range of objects that can be drawn directly on your keyboard in SPRITE STUDIO. Sprites are objects that can be moved about the screen to provide animation for your programs.

the J has been deleted and is no longer added to the color variable C. In line 5055 the variable HL has been substituted for H+J*30. In line 5070 the first FOR statement is deleted. Line 5090 is added.

LIST 5011–5090 and check your results against the above lines. Line 5065 gets overwritten when the following lines, which are necessary to initialize for drawing, are added:

```
1 REM ***** SPRITE STUDIO *****
5037 PRINT J$ " INITIALIZING SPRITES"
5040 PRINT J$ "PLEASE WAIT 10 SECONDS"
5042 REM
5060 H=H+73:IF H<278 THEN 5067
5065 V=V+64:H=48:GOTO 5068
5067 IF H>255 THEN HL=H-256:GOTO 5070
5068 HL=H
5074 POKE U+16,136
5079 REM
5080 REM ** ENTRY POINT FOR RESTART **
5082 PRINT CHR$(147) SPC(8);
5084 PRINT "**** SPRITE STUDIO ****"
5086 U=53248:LC=12288:C=1
5088 POKE U+23,255:POKE U+29,255
5095 TB=1:H=12:V=10  :REM NEW H AND V
```

After SAVEing your program with the name SPRITE STUDIO, you can enter RUN to see how the studio looks on your screen. You will see two rows of four white squares in the lower half of your screen. The top line of the screen displays the title SPRITE STUDIO. The white square in the top row on the left has a small square at its center.

Each of the white squares is an expanded sprite with all its dots turned on. The one black dot in the first sprite is a cursor used for drawing. You cannot move the cursor yet, but you can change the colors with the three function keys f1, f3, and f5; these keys work the same as with SPRITE ART, as do the selection keys 1–8. Press key 8 and then key f5. Sprite 8 changes color. You can select any one of the eight sprites with keys 1–8.

In the STUDIO, you will draw directly on the sprites with the cursor. Instead of moving sprites about the screen as in the SPRITE ART program, you move the cursor within the sprites. This means that the location subroutine will have to be modified to perform cursor movement rather than sprite movement. Before adding the lines for this, let's look at the sprite initialization routine. It has some interesting new programming devices.

The changes you made to lines 5011–5072 were mostly concerned with the

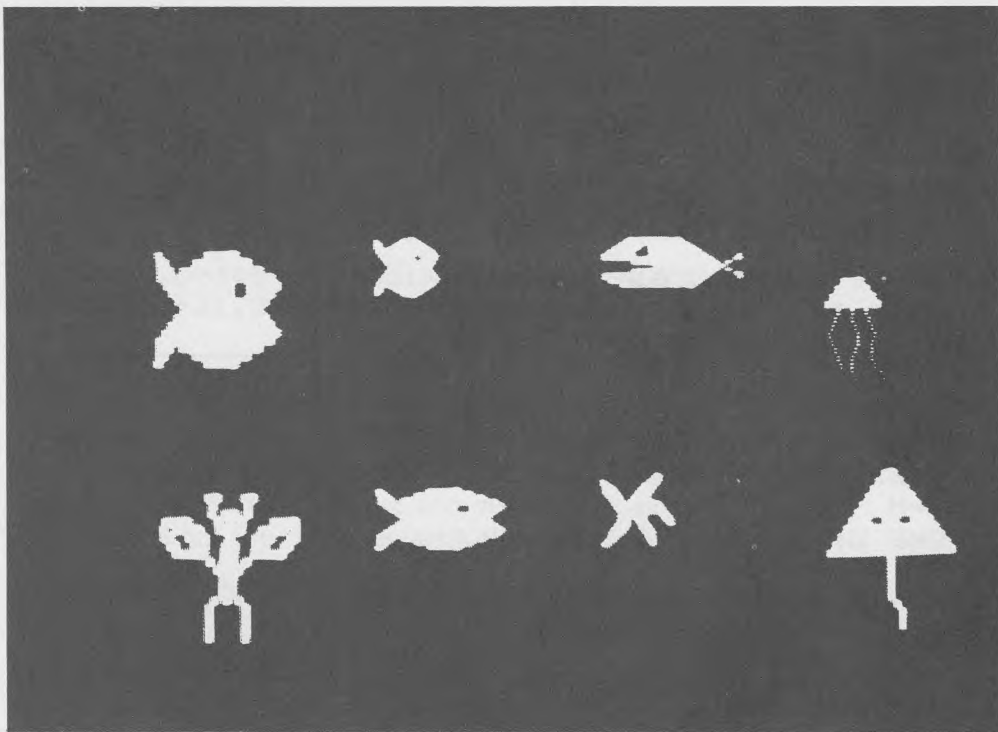


Photo 8 Denizens of the deep are drawn in SPRITE STUDIO. Up to eight sprites can be drawn, saved on tape or disk, and then recalled for editing.

placement of sprites in two rows in the lower half of the screen rather than, as in SPRITE ART, a single row at the top. The variable HL in line 5022 is needed because we are placing sprites 4 and 8 past horizontal position 255. Remember from our previous discussion of sprite positioning that sprites positioned past 255 required special handling. This is done in lines 5067–5074. The variable HL is the four lower bits of the horizontal position. If the horizontal position H goes beyond 255, then in line 5067 HL is set to the difference, and in line 5074 the high bit is set in memory location U+16. Since we know that only the bits for sprites 4 and 8 have to be set, we POKE 136 (128 (sprite 8's bit) + 8 (sprite 4's bit) = 136) into U+16. This is the only time in the STUDIO program that we have to concern ourselves with this complicated positioning scheme. From now on, we will be positioning cursors, not sprites.

Line 5080 designates a second entry point at which the program can be started. Usually you start a program simply by entering RUN. This causes the program to start at the lowest numbered line. Sometimes you want to start the program at a point that will not destroy what you have already entered. You can do this by adding a line number to the RUN statement—in this case, RUN

5Ø8Ø. The program will begin execution at line 5Ø8Ø, bypassing the part of the initialization that would wipe out what already is in the sprites. Line 5Ø86 ensures that necessary variables are set on reentry. You will see how all this works later.

Line 5Ø88 initializes the sprites to expanded mode. This is the most convenient mode for drawing. You will be able to contract and expand the sprites, just as you did with SPRITE ART, and thus see how the sprites will actually look, expanded or unexpanded, while you are drawing them.

Line 5Ø95 sets the values of H and V to the starting position at the center of the sprite. From now on, H and V no longer refer to sprite positions; they will refer to the cursor position. The new variable on line 5Ø95, TB, stands for this bit. By setting TB to 1, the bit at the cursor position is turned off. This turns on the cursor in the first square. If this sounds wrong, remember that all the sprite dots are initiated to on. Therefore, turning the cursor on requires that the dot at the cursor position be turned off. You will see how this works when you make the following program changes for cursor movement and drawing. (Make sure you SAVE new versions of SPRITE STUDIO as you go along.)

LIST 12–4Ø and 5125–5145. With these lines displayed on your screen, make the necessary changes so that the lines on your screen are the same as the following; add lines not already present and leave the lines not listed here as they are:

```
12 REM ** ONE DOT ON/OFF **
20 IF H<00 THEN H=23:V=V-1
30 IF H>23 THEN H=00:V=V+1
35 IF V>20 THEN V=20
40 IF V<00 THEN V=00
5102 REM IF FE=1 AND DE=0, ZAP CURSOR
5103 REM IF FE=1 AND DE=1, DRAW A DOT
5104 REM IF FE=0, WRITE NEW CURSR. POS.
5125 FE=1:GOSUB 20  :REM ZAP/DRAW
5130 IF DE=1 THEN DE=0:GOTO 5170
5135 FE=0
5140 H=H+MH
5145 V=V+MV
5150 GOSUB 20       :REM WRITE CURSOR
5155 GOTO 5120
5170 GOTO 5120
```

Lines 2Ø–4Ø are the now familiar border conditions as they apply to individual sprites; they prevent the cursor from leaving the selected sprite. Since you probably will most often be drawing from left to right, line by line, when the cursor

hits the right side of the sprite it is programmed to wrap around one line down. It will wrap around one line up when it hits the left side. It will stay in the same position when it hits the top or bottom of a sprite.

Lines 5125–5155 are slightly modified versions of what we had in the DRAWING BOARD. As you will see, the processing is similar in both cases. We won't be using arcs or angles other than 45 degree diagonals. Therefore, RA, DH, and DV have been removed from lines 5140–5145. Line 5170 is a temporary GOTO that will be overwritten later.

To provide for cursor movement, delete line 45 and add the following lines:

```
50 CH=INT(H/8)
55 LB=(LC+64*SP)+(V*3+CH)
60 BT=7-(H AND 7)
65 REM
66 REM * IF FE=0, WRITE NEW CURSOR POS.
67 REM IF TB=1 DOT IS A 1;IF 0 THEN 0
70 IF FE=0 THEN 85
75 IF TB=1 THEN 100
80 IF TB=0 THEN 105
85 TB=SGN(PEEK(LB) AND 2^BT)
90 IF TB=1 THEN 105
95 IF TB=0 THEN 100
100 POKE LB,PEEK(LB) OR 2^BT:RETURN
105 POKE LB,PEEK(LB) AND 255-2^BT
110 RETURN
```

NOTE: In lines 85, 100, and 105, use the up arrow on your keyboard for the circumflex character, ^. The same applies to all program lines that follow.

Again, you will notice a similarity to DRAWING BOARD processing. We will explain these lines in detail after adding the lines for drawing. For now, test what you have entered by RUNing SPRITE STUDIO.

When the eight squares appear on your screen, press the H key. The cursor moves left one dot position. Press U and the cursor moves up. Press N and it moves diagonally down and left. All the direction keys, the space bar, and the L key work exactly as in the previous programs. Press the L and semicolon keys and the cursor goes scanning across the sprite, wrapping around one line down until it hits bottom, and then stays there.

If you select another sprite with the keys 1–8 and then press a direction key,

the cursor will appear in the selected sprite but will be at some border position. We can improve on this with a few changes. LIST 988–992 and then alter them to correspond to the following lines:

```
988 POKE U+21,J OR 2^SP:TB=1
990 V=10:H=12:J=LC+64*SP :REM SET CRSR.
992 POKE J+31,PEEK(J+31) AND 247
```

Line 988 gets an additional statement, and lines 990 and 992 are rewritten.

When this latest version is RUN, each time a new sprite is selected, the cursor will appear at the center of the sprite. If the cursor is moved off center and the sprite is deselected and then reselected, the old cursor as well as the new cursor will be present, although only the new cursor will be active. This is no problem, because with drawing capability old cursors are easily removed.

You can turn the selected sprite on and off by pressing Z, the zap key. If a sprite is off when selected, it is automatically turned on when reselected, and the cursor appears dutifully at the center, even if an object was previously drawn in the sprite.

Drawing Sprites

We can now add drawing capability with just two simple changes. LIST 874–876. Change them to conform to the following:

```
874 IF K$="J" THEN DE=1:TB=0:RETURN
876 IF K$="K" THEN DE=1:TB=1:RETURN
```

That's all there is to it. Bring up SPRITE STUDIO, and let's start drawing.

Press the J key and then the H key. The cursor moves to the left as before, but now it leaves a black dot behind. Press the J and H keys several more times. You have drawn a line. Each time you press the J key you turn off a dot at the position of the cursor, leaving it in background color after the cursor moves away.

Draw about three horizontal lines, one on top of the other. They don't have to go all the way across the sprite, just make them long enough to get a solid color patch. Move the cursor either above or below the color patch, and then run the cursor right through it. As the cursor runs through the patch it turns on, becoming the color of the foreground, that is, the color of the rest of the sprite. This makes it possible always to see the cursor irrespective of whether it is moving through turned on or turned off dots.

Move the cursor to inside the dark area again. Press the K key. Move the cursor away. It leaves behind the foreground color, a turned on dot. The K key turns dots on; the J key turns dots off. The color of turned on dots, the foreground color, is selected by function key f5. The color of turned off dots, the background color, is selected by function key f3. Notice that as you press f3, the patch you created takes on the color of the background, whereas as you press f5, the dot within the patch and the sprite itself change color.

All the J and K keys do is set the dot off or on, respectively. The color of the dot is determined independently so that sprites created in SPRITE STUDIO can be given any color in subsequent programs. The purpose of including color control in the STUDIO is so you can see how the sprites will look with various color combinations, as well as allowing you to turn your studio into a gallery for displaying your sprite creations in a variety of colors.

To get rid of the cursor, use the J or K key to set it to the same color as its surroundings. The cursor is still there, just not visible.

The space bar and L key move the cursor, as before. However, they do not draw; you still must press the J or K keys to draw when the cursor is in continuous motion.

Erasing is simply a matter of using the K key to reverse what the J key has drawn and vice versa.

How the Program Draws Sprites

If you are not interested in the details of how the program draws, skip to the next section.

Sprite drawing is performed by lines 5Ø–11Ø. The first requirement is to convert dot positions that we humans like to use to those that the computer uses. This is the function of lines 5Ø–6Ø. We specify a dot position in a sprite by giving its horizontal position, H, and its vertical position, V. Measurement starts in the upper left-hand corner of the sprite, so the dot in the upper left-hand corner is at position H=Ø, V=Ø. A sprite has 21 rows of 24 dots each, so H ranges from Ø to 23 and V ranges from Ø to 2Ø. Compare this to the situation in DRAWING BOARD where H and V measured dot positions on the whole screen rather than a restricted area as now.

We cannot read or write individual dots (bits) from or to memory; only complete characters consisting of 8 dots can be read or written. In terms of characters, the sprite consists of 21 lines of characters (the same as for dots) with 3 characters per line (3 characters × 8 dots = 24 dots per line).

Line 5Ø converts H, the number of horizontal dots, to the number of characters, or the closest integer value computed. Then line 6Ø finds the actual dot (bit) position within that character. Line 55 computes the character location in memory corresponding to the H and V values. First, starting with the base

value, LC, the selected sprite's location is determined. We have located all sprites consecutively, so we simply multiply sprite number, SP, by the 64 characters in each sprite (21 lines × 3 characters + 1 character for housekeeping). Then we add in the number of lines, V, times the 3 characters per line, plus CH, the character position on the last line.

The resulting memory position, LB, is PEEKed and POKEd in lines 100 and 105. Once the character is removed from memory with a PEEK, we can access any of its 8 dots (bits) with the BT value from line 60. If we want to turn the bit on, we use the OR function in line 10. If we want to turn it off, we use the AND function in line 105.

Two different types of operations are performed: (1) moving the cursor with the direction keys and (2) setting dots off or on with the J and K keys. First, we'll examine cursor movement. Then you'll see that the setting of dots is just a simple variation of what happens when the cursor is moved.

To move the cursor, it must be turned off at its present location and turned on at its new location. But to turn the cursor off we cannot just set it to 0. Before the cursor arrived at its present position the dot at that position could have been either turned on or turned off. When the cursor leaves for a new position it must restore the dot to its original state. The original state is stored in the TB (this bit) variable. If TB is 1, we must set the dot on, which is done with a POKE in line 100. If TB is 0, we must turn the dot off which is done in line 105.

Look for a moment at the main line program, lines 5120–5155, which is retained from the SPRITE ART. The first thing main line does after a return from the key subroutine is to set the FE flag to 1 in line 5125 and branch to line 20. If FE is not equal to 0 in line 70, lines 75 and 80 will be executed. These perform the dot restoration functions described above. The cursor is now erased, and the program returns to main line at line 5130. Line 5130 tests to determine if a J or K key was pressed. If neither J nor K was pressed, DE is 0, and lines 5135–5150 are executed in order to write the cursor in its new position.

Next, in line 5135 flag FE is set to 0, in lines 5140–5145 the new position of the cursor is computed, and in line 5150 a second branch is made to line 20, this time to write the cursor in its new position. At line 70 the program will branch to line 85 in which the status of the dot in the new position is saved in variable TB. First, we get the current character with PEEK(LB). Then, we get the current dot's numerical value through the AND function. The sign function, SGN, returns a 0 if the numerical value is 0, and a 1 if the numerical value is other than 0, which will be the case if the dot is turned on, irrespective of the dot's position within the character.

Lines 90 and 95 examine TB to determine whether the cursor should be turned off. Remember, to be visible the cursor must be set opposite to the pre-

vious setting. This is exactly what lines 90 and 95 do by branching to lines 105 and 100, respectively, in the opposite way from lines 75–80. The cursor is thus made visible at the new location, and processing returns to main line.

If drawing is performed with the pressing of the J or K keys, TB is set to 0 or 1, respectively, in lines 874 and 876. This is all that is needed. Lines 70 and 75 will cause the desired dot setting to be POKEd, depending only on the value of TB. When execution returns to main line, the setting of DE to 1 in line 874 or 876 will cause execution to bypass the second GOSUB 20 used for cursor movement in line 5150. The new dot value has been displayed, and main line loops back to line 5120.

You can use these program lines as templates whenever you want to program the turning on and off of individual dots in a sprite.

Expansion and Contraction of the Sprites

If you have tried using the T and G keys to contract and expand the sprites as you did in SPRITE ART, you found that things didn't work quite right. This can easily be corrected. Enter LIST 1709–1770. Make changes to lines 1710, 1720, 1730, 1750, 1760, and 1770 so they look like the following:

```
1710 V(SP)=1-V(SP):IF V(SP)=1 THEN 1725
1720 GOTO 820
1730 GOTO 820
1750 H(SP)=1-H(SP):IF H(SP)=1 THEN 1765
1760 GOTO 820
1770 GOTO 820
```

SPRITE ART starts with contracted sprites and uses continuous motion. SPRITE STUDIO starts with expanded sprites and does not want the cursor moved after changing dimensions. Hence, you will find that the above changes make the T and G keys do just what they should, and no more.

Why is the expansion and contraction of sprites accomplished so easily? Why don't size changes require changes to the ONE DOT ON/OFF subroutine?

When a sprite expands the number of dots remain the same, but each dot is displayed as twice its original size. Actually, each dot in memory is displayed as two dots on the screen in the direction of the expansion. Expansion in both directions results in four dots on the screen for each dot in memory. But the sprite is still composed of a 24 × 21 dot rectangle. The expansion is all performed by hardware, with POKEs to memory locations U+23 and U+29. No adjustments to the program are required.

SEMIAUTOMATIC DRAWING

You now have the capability of drawing any possible sprite by turning each dot on the 24 × 21 dot sprite rectangle on or off with the J and K keys. However, there are a number of ways the program can assist you. The following additions are optional, but you will find they can greatly add to your enjoyment in drawing sprites.

Reversing Colors

Since sprites start with all dots turned on, you will often find that it is easier to draw sprite figures by turning dots off rather than by turning them on. However, you may want to display them with the dots set with on/off values interchanged, that is, with the foreground and background dots reversed. The following lines perform this function with one press of the X key:

```
880 IF K$="X" THEN 1810  :REM REVRS C'S
1800 REM
1801 REM * SWTCH BACKGRND. & FORGRND. *
1810 FOR J=0 TO 62
1825 K=PEEK(LC+64*SP+J)
1830 POKE LC+64*SP+J,255-K
1835 NEXT J:TB=1-TB:GOTO  820
```

With the sprites displayed, press the X key. The background color is painted over the first sprite, except the cursor is switched to foreground. Press X again, and the sprite toggles back to its original colors. Select a sprite and draw a figure. Reverse it with the X key. You now have a useful tool that makes it easier to draw sprites.

Drawing Symmetric Sprites

Many sprite objects have symmetries that can be used to reduce the number of drawing steps. The following program lines will draw symmetric dots automatically. In addition to reducing drawing effort, you'll find that *autosymmetric drawing* will allow you to create objects that never would have occurred to you otherwise.

```
300 REM
301 REM ** AUTO-SYMMETRIC PROCESSING **
```

```
302 REM SYMMETRIES: HORIZNTL.,VERTICAL,
303 REM          DIAGONAL, HORIZ. & VER.
320 IF XS<>1 THEN 330
325 H=24-H:GOSUB 20:H=24-H
330 IF YS<>1 THEN 345
340 V=20-V:GOSUB 20:V=20-V
345 IF DS<>1 THEN 360
350 T=22-H:H=22-V:V=T:GOSUB 20
355 T=22-H:H=22-V:V=T
360 IF AS<>1 THEN 375
365 H=24-H:V=20-V:GOSUB 20
370 H=24-H:V=20-V
375 RETURN
861 IF K$="F" THEN XS=1-XS:GOTO 820
862 IF K$="D" THEN YS=1-YS:GOTO 820
863 IF K$="A" THEN DS=1-DS:GOTO 820
864 IF K$="S" THEN AS=1-AS:GOTO 820
5170 GOSUB 320   :REM AUTO-SYMMETRIC
5175 GOTO 5120
```

With AUTO-SYMMETRIC PROCESSING added, press the F key once, then the J key, then the H key. Repeat the J and H combination several times. A line expands from the center. Press the U key instead of the H key. Two lines are drawn up the screen, symmetric about the center. The U key moves the cursor. Only the J key draws so you won't see the symmetric dot appear until you press J (or K, if that is the drawing key you are using). Use the @ key in place of the U key. Two diagonals are drawn that slope in opposite left-right directions.

Press the F key again to turn off autosymmetry. Press the 2 key to select sprite 2. Press the D key once. Then press J followed by U several times. A vertical line expands symmetrically about the center. Switch from the U key to the H key and two lines are drawn symmetrically about the center.

Press the D key again to turn it off; then try the S key on sprite 3. Use a combination of the J and Y keys. A diagonal line will expand that is symmetric about the center. By switching from Y to N, a zigzag line forms, still symmetric about the center.

Press S again and then A. Select sprite 4. Press the J and Y key combination. The result is the same as for the S key, a line sloping from upper left to lower right. As before, change from the Y key to the N key. Instead of a zigzag line, two lines are drawn in the same direction, symmetric about the diagonal.

To summarize, the four autosymmetric keys draw dots that are symmetric in the following ways:

F key	Symmetric about a vertical center line.
D key	Symmetric about a horizontal center line.
S key	Symmetric about both horizontal and vertical center lines. This is like a symmetry through the center point. It is an interchange of horizontal and vertical dot positions.
A key	Symmetric about a diagonal running from the upper right-hand corner to the lower left-hand corner.

You can use the autosymmetric keys singly, as above, or in any combination. See what happens when two or more symmetry keys are active at one time.

This completes the drawing features we will describe in this book. Perhaps other drawing aids will occur to you. We hope you will be able to incorporate your ideas into SPRITE STUDIO by following the methods described above.

DISK AND TAPE OPERATIONS

In order to use the sprites you create in SPRITE STUDIO, you must be able to write them on tape or disk. You can then read them into other programs and use them in any way you like. You may also want to reLOAD them into the STUDIO in order to make changes or copies. We will now show you SPRITE STUDIO routines for these operations. The additional program lines will be added in three stages: the menu, the SAVE routine, and the LOAD routine.

Menu for Disk Operations

The following program lines will allow you to tell the computer the name of the file and the sprites that you wish to store:

```
950 REM
951 REM * DISK SAVE OR LOAD *
955 IF A=211 OR A=204 THEN 1910
1900 REM
1901 REM * MENU FOR SAVE/LOAD *
1910 PRINT CHR$(147)    :REM CLEAR/HOME
1912 PRINT "YOU CAN SAVE OR LOAD 1-8"
1914 PRINT "SPRITES AT A TIME."
1916 INPUT "START SPRITE NO.";SS
1918 IF SS<1 OR SS>8 THEN 1916
1920 INPUT "END SPRITE NO.";ES
1922 IF ES<1 OR ES>8 THEN 1920
```

```
1924 IF ES<SS THEN 1920
1926 POKE U+21,0:REM SPRITES MST BE OFF
1928 PRINT "ENTER Q TO QUIT"
1930 INPUT "FILE NAME OR Q";F$
1932 PRINT CHR$(147)
1934 IF F$<>"Q" THEN 1938
1936 POKE U+21,255:GOTO 820
1938 IF A=204 THEN 2010 :REM LOAD?
```

You can RUN this to test your entries. With the eight sprites displayed on your screen, enter SHIFT/S, that is, hold down the SHIFT key and press the S key. The S stands for SAVE so the program responds with the message

YOU CAN SAVE OR LOAD 1–8
SPRITES AT A TIME.

It then asks the following question:

START SPRITE NO.?

Press the 1 key and then press RETURN. The program responds with a second question:

END SPRITE NO.?

Press the 8 key. The sprites will be turned off. The program will display

ENTER Q TO QUIT
FILE NAME OR Q?

Press the Q key and then RETURN. The menu will disappear and the sprites will reappear.

Repeat all the above steps, substituting SHIFT/L for SHIFT/S in the first step. L stands for LOAD, and all steps should work the same as for SAVE.

As you can see, the program allows you to store and retrieve from one to eight sprites in any file you care to name. If you were interested in only one sprite, you would enter the same start and end sprite numbers.

Press SHIFT/S again. Press the 9 key and then RETURN. You entered an invalid character, so the program repeats the question. Press the 8 key and RETURN. The program will accept this and ask for the end sprite number. Enter a 7. The program repeats the question because the end number is less than the start number, an invalid response. Enter an 8 for the end number and then Q to quit. Exit the program so you can enter the SAVE routine.

Notice the use of the INPUT statement in lines 1916 and 192Ø. In previous programs we used the GET statement for keyboard entries. Both statements can be used for transferring keyboard entries into the program. The GET statement provides better control: You can examine each individual key entry and proceed in any way you want. The INPUT statement is more suited to entries of more than one keystroke, as when entering a file name. With INPUT all the editing and cursor controls on your keyboard can be used, therefore, if there is a mistake in keying, it is easily corrected. This is not possible with GET. GET can be used without the RETURN key. INPUT always requires that you press RETURN to signal the completion of an entry. INPUT also allows you to include a prompt message as in line 193Ø.

Saving Sprites on Disk or Tape

The following lines will save sprites on a disk:

```
1940 REM
1941 REM * SAVE SPRITE *
1945 OPEN 2,8,2,"@0:"+F$+",S,W"
1950 FOR SP=SS-1 TO ES-1
1955 FOR J=0 TO 62
1960 PRINT CHR$(147)"SPRITE IS     "SP+1
1965 PRINT "CHARACTER IS "J+1
1970 S$=STR$(PEEK(LC+64*SP+J))
1975 PRINT#2,S$      :REM WRITE CHAR.
1980 NEXT J:NEXT SP
1985 PRINT#2,CHR$(13)
1990 CLOSE 2:SP=SP-1:POKE U+21,255
1995 GOTO 820
```

If you are using tape, you need only change line 1945 as follows:

```
1945 OPEN 2,1,1,F$
```

The second number in the OPEN statement tells the computer which device you are using. *Device numbers* are usually 1 for tape units and 8 for disk units. If you are using different device numbers, you must enter that number in the OPEN statement, but if you have not changed these numbers since acquiring your equipment, just copy the lines, as is.

To test SAVE SPRITE, draw a simple object in the first sprite. Enter SHIFT/

S, then enter a 1 for the start sprite number and a 1 for end sprite number. For *file name* enter SPRITE 1. Do not put the name in quotes or add the device number as you do when SAVEing a *program*.

When the RETURN key is pressed several things happen. If you are using a disk, the screen is cleared, the disk unit goes to work, and the following information is displayed on your screen, where XX stands for continuously changing numbers:

```
SPRITE IS       1
CHARACTER IS XX
```

If the program has been entered correctly, the disk unit stops running, the red light on the disk goes off, XX is displayed as 63, the sprites are turned on, and the object you drew is displayed in the first sprite.

If you are working with tape, the action is similar but with these differences: After you enter the file name and press RETURN, the screen clears and the computer displays PRESS RECORD & PLAY ON TAPE. When this is done, the tape unit starts up and the screen clears and remains blank for several seconds. After the sprite and character number messages described above are displayed for a few seconds, a blank screen appears again. This is repeated a couple of times. Then the tape stops and the sprites reappear. The object you drew is in sprite 1 and the sprite and character numbers are displayed.

NOTE: Because of differences in products and changes to equipment, your equipment may not perform exactly as described above. If the final results are the same, the program is probably working correctly.

When we work with sprites we use dots. However, these dots must always be assembled into groups of eight for the computer. Each of these groups is a character, which is what XX displays. As the characters are written to the storage device, they are counted and the count is displayed in XX. Since a sprite has 21 rows of dots with 24 dots per row, there will be 63 characters transferred to disk or tape per sprite. If at the end, XX displays 63, you know that the entire sprite has been saved.

Saving Eight Sprites at a Time

Draw test objects in all eight sprites. Save them all by entering a 1 for the start sprite number, an 8 for the end sprite number, and the file name SPRITE

1–8. Now when XX reaches 63, instead of stopping, XX returns to 1 and the sprite number is incremented to 2. The process is repeated until the sprite number is 8 and XX is 63.

You now have two sprite files that you can use to test the next program lines for retrieving sprites from disk or tape.

Retrieving Sprite Files from Disk or Tape

Here are the final entries for SPRITE STUDIO:

```
2000 REM
2001 REM * LOAD SPRITE *
2010 OPEN 2,8,2,"@0:"+F$+",S,R"
2015 FOR SP=SS-1 TO ES-1
2020 FOR J=0 TO 62
2025 PRINT CHR$(147)"SPRITE IS     "SP+1
2030 PRINT "CHARACTER IS "J+1
2035 INPUT#2,S$:SV=VAL(S$)
2040 POKE LC+64*SP+J,SV
2045 NEXT J:NEXT SP
2050 CLOSE 2:SP=SP-1:POKE U+21,255
2055 GOTO 820
```

As with SAVE SPRITE, the OPEN statement in line 2010 must be changed for tape units. Substitute the following line:

```
2010 OPEN 2,1,0,F$
```

First, try retrieving sprite 1. (If you are using tape, be sure either to position the tape ahead of where the sprite was saved or to rewind the tape all the way.) Clear all sprites by pressing the RUN/STOP–RESTORE key combination. Enter RUN. When the cleared sprites are displayed, enter SHIFT/L to initiate the LOAD operation. Enter 1 for both the start and end numbers and a file name of SPRITE 1. Everything that happens is the same as for the SAVE program except at the end: When the sprites are turned on, sprite 1 appears in the first square. Again, the screen displays the last sprite and character transferred, that is, sprite 1 and character 63.

Let's get all eight sprites back from disk or tape. Enter SHIFT/L again. The first sprite number is 1, the end sprite number is 8, and the file name is SPRITE 1–8. The screen will display each sprite number as its characters are read into memory. At the end, presto! All eight sprites appear just as you drew them.

> **NOTE: When LOADing sprites from tape, the error message, STRING TOO LONG, may be displayed occasionally. When you see it, rewind the tape and repeat the LOAD operation; this usually solves the problem.**

You don't have to retrieve sprites the same way that you SAVE them. For example, if you SAVE sprites 1 to 8, you can retrieve them as sprites 5 to 7 from the same file. The original sprite 1 will be retrieved as sprite 5, sprite 2 as sprite 6, and sprite 3 as sprite 7. Then the operation will stop.

If you try to retrieve more sprites from a file than you saved, the system will hang-up, that is, stop. Press the RUN/STOP–RESTORE combination and enter RUN. And, of course, rewind the tape. Then reenter your read request with fewer sprites.

Recovery from Error

Suppose that you have spent time creating sprites that you like and for some reason there is an error—BASIC leaves your program and displays an error message. All is not lost. Depending on the problem, you can reenter your program by entering RUN 5Ø82, the entry point for restart. This is possible even if the cursor is hidden by a sprite. Just enter RUN 5Ø82 as if you could see the entries, and press RETURN. You will be in control again. Be careful *not* to press the RESTORE key. You can press RUN/STOP at any time and then reenter the program with RUN 5Ø82.

Tips on SAVEing and LOADing Files

The key lines for SAVEing and LOADing are 1926, 1945, 1975, 199Ø, 2Ø1Ø, 2Ø35, and 2Ø5Ø. All of these lines except 1926 are required in any disk or tape transfer. They can be used as templates when you write your own programs. Although here we are concerned only with sprites, the computer doesn't know, and doesn't care, what's in a file. It can be sprites, stock tables, or phone numbers; it's all the same to the computer. Of these key lines, only line 1926 is required for sprites.

Line 1926 turns off all sprites by POKEing Ø into sprite memory location U+21. Without this POKE, operation becomes unpredictable. The system may hang-up with the disk's red light on, forcing you to enter the RUN/STOP–RESTORE combination. This is unfortunate, because it is much nicer to watch the sprites being painted on the screen as they are read in. You can see this happen by inserting REM at the beginning of line 1926. The program will probably

work for a while, but eventually it will hang-up. I don't know the reason for this, but there may be a bus contention problem between the sprite display logic and the disk/tape transfers.

The OPEN statements for disks in lines 1945 and 2010 act like gates, opening the path for data transfer. Both lines are identical except for the last characters, R and W. These are the characters that tell the computer the mode of data transfer for disks, that is, whether the transfer is a read or a write. R stands for read and W for write.

OPEN statements work somewhat differently for each type of equipment, and may even vary for different models of the same equipment. The following description applies in general, but you may find differences for the equipment you are using.

The first character of the OPEN statement is used by the computer as a short hand for the file name. It is called the *logical file number*. You can use any value between 1 and 127 for the logical file number. The choice of 2 in lines 1945 and 2010 was arbitrary. If you have more than one OPEN statement active at a time, each must be given a different logical file number, otherwise you are free to choose any number between 1 and 127.

The second character in the OPEN statement is the *device number*, which has already been explained. It is usually 8 for disks and 1 for tape. This character can be left out if you are using the Commodore tape unit.

The third character in the OPEN statement specifies a *channel* on disks and an *operation* on tape. Your Commodore 64 has 16 channels, numbered 0–15, with which it can communicate with external devices such as disks and printers. These channel connections are not permanent connections. Rather, they are established by you in the OPEN statement. You must tell the computer which channel to use. There are certain restrictions. When OPENing for a disk transfer you can only specify channels 2–14. In lines 1945 and 1920, channel 2 was arbitrarily selected from the 13 available channels.

Tapes work somewhat differently. The third character of the OPEN statement, called by the confusing name, *secondary address*, specifies an operation. If the character is 0 or is omitted, a read operation is performed. If it is a 1 or a 2, a write operation is performed. The 2 will cause an end-of-tape marker to be written on tape, effectively sealing off the tape for reading past the marker. Use a 2 if you want to receive the error message "?DEVICE NOT PRESENT" if you read past the end of the data you are currently writing.

The next item in the OPEN statement is the file name. In line 1930 the file name you entered was stored in the variable F$ and is now entered in the OPEN statement. The computer now knows what file to associate with the file number 2, which is the first character in OPEN. In subsequent statements, whenever the computer sees the file number 2, it will know that this is a code for the file name. That is all that is needed in a tape OPEN statement. Disks

require more. The file name must be preceeded by Ø:. If you want to be able to change an existing file, you must use @Ø:, as is done in lines 1945 and 2Ø1Ø. If you want to protect your sprite files from being overwritten, delete the @.

The S in the OPEN statement stands for *sequential file*. All tape files are sequential so there is no need to specify file type. Disks, on the other hand, can also have *random* and *relative files*. Sequential files are the easiest to work with and are suitable for storing sprites.

The @Ø:, file name, file type, and file mode must all be entered as one string. But we obtained the file name stored in F$ as a separate string. We must combine it with the others into a single string. This is done with the + sign, which is used to combine, or concatenate, strings into a single string. All substrings must be in quotes or end in $, as can be seen in lines 1945 and 2Ø1Ø.

The OPEN statement sets the stage, but the real work is done in lines 1975 and 2Ø35. Line 1975 uses the PRINT# statement to transfer anything stored in the sprite string variable S$ to disk or tape. PRINT# is a close relative to PRINT with which you are familiar. The difference is that instead of transferring or "printing," data to the screen, PRINT# transfers data to tape or disk. The character # is part of the word, so no space is allowed after the T in PRINT. Any variable can follow PRINT#. The variable can contain any type of information and can be either a numeric or string variable.

The character 2 in line 1975 is the file number that is the code for the file name F$. This relationship was established in the OPEN statement.

In line 197Ø, variable S$ gets its data from a PEEK into memory where the sprite is stored, as determined by the sprite base address LC. The character read by the PEEK is converted to a string by the STR function, and the result stored in S$.

Lines 194Ø–198Ø contain an inner and an outer FOR...NEXT loop. The inner loop scans memory for all characters in a sprite. The outer loop moves to successive sprites. It starts by setting sprite number SP equal to the start sprite SS. It then steps to each successive sprite until the end sprite number ES is reached. Variables SS and ES are set in lines 1916 and 192Ø, respectively, when you make these entries on the keyboard.

Here we have transferred one character at a time directly from the sprite area in memory. In other applications you can LOAD the variable in the PRINT# statement with many characters. The practical limit is 8Ø characters, since that is the maximum allowed when reading with INPUT#. You could also use an array variable directly instead of first LOADing from memory.

In line 2Ø35 INPUT# performs the inverse operation of PRINT#, transferring data from tape or disk to a variable, in this case S$. We have used the same variable in the PRINT# and INPUT# statements, but this is not required. Again, the character 2 tells the computer from which file to read. The two FOR...NEXT loops perform the same operation as when data are saved. Char-

acters are read one at a time and converted back to numeric values by the VAL function in line 2Ø35. Whether the data are numeric or string, it is usually safer to LOAD into a string variable just in case string data are read in.

Lines 199Ø and 2Ø5Ø contain the CLOSE statement. CLOSE is always required after OPENing a file and completing operations. This frees the files and channels for other operations. It also clears out the buffers, a requirement for successful LOAD and SAVE operations. The computer does not write each character it receives from the PRINT# statement immediately, This would be wasteful of time and disk space. Instead, it assembles the characters into blocks and transfers a complete block to disk or tape. When the computer is reading a similar process is performed in reverse. This accounts for the intermittent activity of the disk or tape unit that you probably noticed when you SAVEd and retrieved sprites. When an operation is completed, the block may not necessarily be full. The CLOSE statement forces the transfer of the remaining data of the last block. Again, the character 2 in the CLOSE statement tells the computer the name of the file to CLOSE.

The two statements following CLOSE in lines 199Ø and 2Ø5Ø apply only to this program. The sprite number SP is set to the last sprite transferred. At the end of the FOR...NEXT loop, SP is one greater than this value. Finally, when all work is completed, the sprites are turned back on with a POKE into sprite memory location U+21.

Lines 196Ø, 1965, 2Ø25, and 2Ø3Ø let you know what's happening during that uncertain interval of data transfer. You can speed things up by deleting them or moving them outside the FOR...NEXT loop.

TABLE 7 KEY ASSIGNMENTS FOR SPRITE STUDIO

Type	Key	Operation
Direction		
	;	Move right
	H	Move left
	U	Move up
	M	Move down
	@	Move up and right
	Y	Move up and left
	/	Move down and right
	N	Move down and left
Draw		
	J	Set dot to Ø (background)
	K	Set dot to 1 (foreground)
Control		
	Space bar	Repeat last operation
	L	Continuous move
	G	Expand or contract horizontally
	T	Expand or contract vertically
	Z	Turn sprite on/off
Autosymmetric		
	F	Horizontal symmetry
	D	Vertical symmetry
	S	Horizontal/vertical symmetry
	A	Diagonal symmetry
Color		
	f1	Border: cycle Ø–15
	f3	Background: cycle Ø–15
	f5	Sprite: cycle Ø–15
	X	Reverse foreground and background colors
Sprite selection		
	1–8	Sprite selected as per number on the key
Disk and tape transfers		
SHIFT/S		Save on tape or disk
LOAD/S		Load from tape or disk

9

Sprite Theater

Now that you can draw sprites, perform dot mapped graphics, and create computer sounds, let's put it all together in one program, which we'll call SPRITE THEATER. You'll be able to control sprite movement and sound from your keyboard and can use SPRITE THEATER to put on sprite shows or as a workship for creating computer games.

SPRITE THEATER will be mostly SPRITE ART, some SPRITE STUDIO, a smattering of MUSICAL KEYBOARD, and as much of DRAWING BOARD as you care to put in the background. We'll explore the fine art of switching memory banks so you can expand the working space for your graphics programs.

BUILDING THE THEATER

SPRITE ART will be our template. We'll start by cutting the LOAD SPRITE program out of SPRITE STUDIO and then pasting it into SPRITE ART.

In preparation, have copies of SPRITE ART and SPRITE STUDIO programs available, and also the sprite files SPRITE 1 and SPRITE 1–8 from Chapter 8, or their equivalent, so you can test your program. Also have available a working tape or disk for storing SPRITE THEATER as it is being built. With tape, it is easiest to start SPRITE THEATER on a new tape so you can overwrite old versions without danger to other files.

Sprite files are different from program files in that you cannot LOAD and SAVE them in order to copy them. They are sequential files; therefore, if you

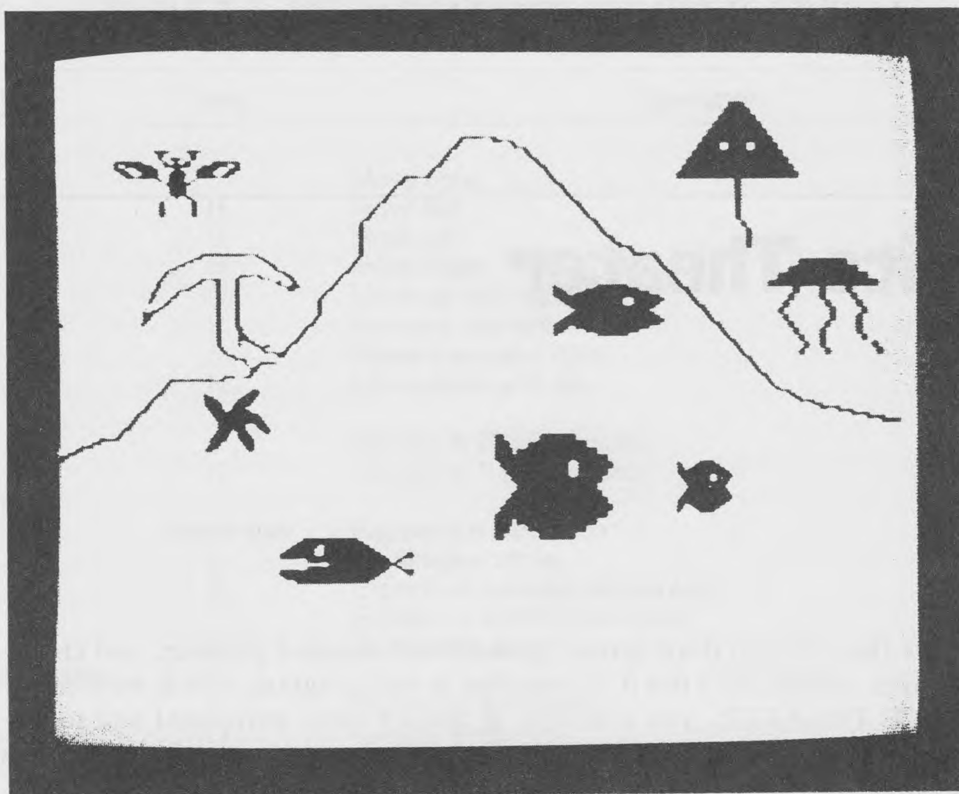


Photo 9 An electronic fish tank is created in SPRITE THEATER from the fish drawn in SPRITE STUDIO and the background drawn in DRAWING BOARD. You manipulate the fish and control sounds and colors from your keyboard.

want to copy, for example, SPRITE 1 or SPRITE 1–8 to a working tape or disk, you must first LOAD them into SPRITE STUDIO with SHIFT/L and then SAVE them with SHIFT/S.

The following cut-and-paste steps are quite easy with disks. However, with tapes you may find that repeated SAVEs and LOADs are more work than making entries directly from the keyboard. It all depends on your relative adeptness with tapes and keyboard. Always remember that cut-and-paste techniques are meant to save time. If they don't, then use the keyboard.

Cutting and Pasting the LOAD Routine

LOAD a copy of SPRITE STUDIO into memory and enter LIST 1910–1938. With these lines displayed, insert a tape cassette or disk that has SPRITE ART recorded on it into your drive unit. Set the cursor right under line 1938 and LOAD a copy of SPRITE ART into memory. This, of course, erases SPRITE STUDIO, which is exactly what we want to do. If you are using tape, the top

lines may be pushed off the top of the screen. You can enter these manually, reLIST them on the next paste operation, or reLIST and paste them individually. Place the cursor on 1910, the top line, and hit RETURN repeatedly until lines 1910–1938 are all entered; be careful not to enter LOAD again on the way down. We have now pasted the LOADing SPRITE menu into SPRITE ART. SAVE this new program as SPRITE THEATER.

LOAD another copy of SPRITE STUDIO into memory, this time entering LIST 2000–2055. LOAD your SPRITE THEATER and then enter lines 2000–2055, as before, so they are pasted into SPRITE THEATER. SAVE SPRITE THEATER.

Continue this cutting and pasting with lines 950–955, lines 1800–1835, lines 1900–1901, and line 880 of SPRITE STUDIO and any other lines that might have been pushed off the top of the screen in the process of LOADing SPRITE ART. You can LIST as many of these lines at one time as will remain on the screen after LOADing SPRITE THEATER. The new lines are pasted in as before. SPRITE THEATER must be SAVEd after each addition.

Lines 1800–1835 add the useful reversal of foreground and background, using the X key, which SPRITE ART did not have.

LIST line 5022, and change V=55 to V=225, moving the initial sprites to the bottom of the screen.

When you RUN your program, you will find that you can control the sprites in the same way that you did with SPRITE ART, which is not surprising since this was our template.

> **NOTE:** **Remember that if you press the J or K key you must press the + key a couple of times to get proper movement. Return to normal with the = key.**

Enter SHIFT/L to test the lines you have added for LOADing sprites from tape or disk. The process is identical to that of SPRITE STUDIO. You'll recognize the program prompts as being the same as before. First, retrieve SPRITE 1. At completion of the retrieval operation, you'll see your sprite drawing in sprite 1. Move the sprites around. Then call up file SPRITE 1–8 with SHIFT/L. All sprites will be LOADed at the positions where you last left them.

At this point you will probably want to create various sprite objects in the STUDIO and move then into the THEATER. First, add the following lines so that the LOAD routine messages will be cleared and the program will have a proper title line:

```
1 REM ***** SPRITE THEATER *****
2052 PRINT CHR$(147)     :REM CLEAR
```

You can delete lines 5Ø68, 5Ø69, 5Ø7Ø, and 5Ø72. This will eliminate the initializing of the sprites to all dots on. Your sprites will be displayed as you left them if you leave the program with RUN/STOP or with the combination RUN/STOP–RESTORE.

Remember that you have the expansion keys G and T from SPRITE ART. Just because the sprites in SPRITE THEATER start small they don't have to remain small.

Also left over from SPRITE ART is the use of the A, D, S, and F keys to create patterns on the sprites. This has been left in to provide added variety, but there will be times when pressing one of these keys by mistake will cause you to destroy a sprite object that you want. If this is a problem for you, delete the pattern-making lines 861–864 and 16ØØ–1635.

THEATER SOUND

Your Commodore 64 has a built-in capability for *detecting collisions* between sprites. When sprites collide the fact is recorded in sprite memory location U+3Ø. You can PEEK into this location to see if there has been a collision. What you do when a collision has occurred is entirely up to you, enabling you to create all types of games based on sprite collisions. SPRITE THEATER provides a template program from which game programs can be derived.

One of the most interesting uses of collision detection is the production of sound based on the sprite that is hit. We will show how this is done by adding sound to SPRITE THEATER. You will learn how to use sprite collisions to create game programs, to explore the world of sprite sound making, or for any other application you can dream up.

In order to add sound, we will make additions to initialize music (sound) memory, detect collisions, and provide keyboard control of sound. Add the following lines to SPRITE THEATER:

```
5080 REM
5081 REM ** INITIALIZE SOUND **
5084 M=54272:FOR J=M TO M+24
5086 POKE J,0:NEXT  :REM INIT. CHIP
5088 POKE M+5,060:POKE M+6,060
5090 POKE M+24,15
5095 POKE M+1,017:POKE M,037
5098 HF=16:N8=1:WF=129
```

These lines are similar to those used to initialize MUSICAL KEYBOARD. Refer to Chapters 4 and 5 for an explanation. Line 5Ø98 initializes to octave 1 and a noise waveform.

The following lines will let you hear sprites crash.

> **NOTE:** Use the up arrow on your keyboard in place of the circumflex, ^, in this and all following program lines.

```
400 REM
401 REM ** COLLISION DETECTION **
402 REM SPRITE HITS SPRITE,HS
420 IF WF=0 THEN 495 :REM SOUND OFF ?
425 SH=PEEK(U+30):J=SH AND 2^SP
430 IF J=0 THEN BC=0:GOTO 495
432 IF BC=1 THEN 495
433 BC=1:MV=-MV
435 SH=SH-2^SP :REM REMOVE SEL. SP.
440 REM
445 REM * COMPUTE NOTE BASED ON SPRITE*
450 SJ=0:FOR J=0 TO 7
455 SJ=SJ+SGN(SH AND 2^J)*J:NEXT J
460 IF SJ>255 THEN SJ=255
470 POKE M+1,(1+SJ)*N8  :REM HI VAL.
475 REM
480 REM * TRN ON, DELAY, TRN OFF, DELAY
485 POKE M+4,WF  :FOR J=1 TO 250:NEXT
490 POKE M+4,WF-1:FOR J=1 TO 002:NEXT
495 RETURN
5070 POKE U+30,0
5152 GOSUB 420
```

Turn up the volume on your amplifier. Then bring your sprites onto the stage, and set sprite 1 moving to the right with the semicolon key followed by the L key so it collides with another sprite. You'll hear the sprites crash. Each sprite produces a different note. The sound is controlled so that they are produced only after a sprite has cleared away from all other sprites. To hear a sound produced by each sprite as it is hit, separate the sprites so the moving sprite clears one before hitting another.

The moving sprite sometimes bounces off the sprite it collides with; other times it passes in front of or behind the sprite that it hits.

If the variable WF (waveform) is set to Ø, no sound is produced because line 42Ø will immediately branch to line 495, the exit. A sprite hit is loaded into the variable SH by a PEEK to sprite memory location U+3Ø. Each sprite is assigned a bit at this location, which is turned on by the computer if the sprite is involved in a collision. Any combination of the bits may be set, but if there is no collision all bits will be Ø.

Every PEEK to U+3Ø clears it to Ø, leaving it ready to record the next colli-

sion and those sprites that have collided. We store its value in SH, after which that particular value will be available until the next PEEK to U+3Ø. In line 425 variable SH is ANDed with 2^SP to determine if the selected sprite has collided. We are not interested in other sprites that may overlap and, therefore, be recorded as colliding. Line 43Ø tests for collision.

The bounce/collision variable, BC, is used to prevent a note from sounding repeatedly when sprites collide. Without BC, movement slows and the same note repeats monotonously until the sprites separate. If BC is 1, then the current collision has had a note sounded and processing at line 432 exits to line 495. Otherwise, MV is set to −MV to create a bounce. We could do the same with MH, but the resulting motion is much less interesting. Reversing only one component of the motion makes the sprites bounce in some cases and not bounce in others. Here is an area with which you can experiment to get different effects.

Once the sprite that is moving has cleared a sprite that is hit, J will be set to Ø in line 425 and BC will be set to Ø in line 43Ø, enabling sound on the next collision.

Line 435 subtracts out the currently selected sprite from SH so the sound computed in the next lines will vary widely. There is a great deal that can be done here, including making the waveform or amplitude depend on which sprite is selected.

Lines 45Ø–47Ø compute the tone of the sound based on which sprite(s) is(are) hit. If more than one is hit, that will affect the note. A FOR. . .NEXT loop cycles through all bits of location U+3Ø, which are now stored in SH. The number SJ is computed with each sprite that is hit, thus contributing an amount related to its sprite number through its position in U+3Ø. In line 47Ø, SJ is combined with the octave N8 to produce a note. Octave N8 is initialized to 1 in line 5Ø98, entered previously.

Lines 485–49Ø set the waveform and turn the note on and off. Both of these functions are performed at the memory location M+4. The low bit of M+4 gates the sound. An even number will be set a Ø in the low bit, turning the sound off. Since in line 5Ø98 WF is initialized to 129, an odd number, it turns on the sound in line 485; in line 49Ø, WF−1, or 128, turns off the sound after a delay determined by the FOR. . .NEXT loop in line 485. The FOR. . .NEXT loop in line 49Ø determines the release time. By changing the values of WF, 25Ø, and ØØ2, you can change the envelope of the sound.

Lines 4ØØ–435 can be a template for collisions in your own programs. Once you have SH you can AND it with a power of 2, as done in line 455, to determine which of the 8 bits are on. (See Appendix A for a description of how this works.) This will tell you the sprites that have collided. An easier situation occurs when you are only concerned if there has been a hit irrespective of the sprites hit. Line 425 is all you need for this.

The following lines will give you some control over the sounds sprites make when they hit. If you are clever, you may even produce some music.

```
882 IF K$="W" THEN 3020
884 IF K$="O" THEN 3060
3000 REM
3001 REM * WAVEFORM CONTROL *
3020 SU=SU+1:IF SU>3 THEN SU=1
3025 IF SU=1 THEN WF=33
3030 IF SU=2 THEN WF=0
3035 IF SU=3 THEN WF=129
3040 RETURN
3050 REM
3055 REM * OCTAVE CONTROL *
3060 N8=N8+1:IF N8>7 THEN N8=1
3065 RETURN
```

The W key controls the waveform, and the O key controls the octave. When the program starts, you will hear the same crash sounds as before. To change the sound, press the O key. Each time you press O the octave increases up to octave 8 then recycles back to Ø.

Change the waveform with the W key. You will get everything from stringlike instruments to the sound of falling pebbles. This is a *sawtooth waveform* (see Figure 3). Since each sprite has its own note, you have in effect a sprite music maker. Press the W key again and all sounds stop. You can do this when you want your sprites to move silently through space. A third press of W cycles back to the noise waveform.

See the key assignments in Table 8 at the end of this chapter for all the things you can do with your keyboard.

THEATER SCENERY

The DRAWING BOARD program can be used to create scenery and other background objects for your SPRITE THEATER. Before adding the following lines, SAVE a copy of SPRITE THEATER in a safe place and work with another copy. Adding dot mapped graphics to sprite graphics gets a little tricky, and you don't want to lose SPRITE THEATER. In order to avoid confusion, we will change the name of SPRITE THEATER when we modify it for dot graphics so you won't have two versions of the same program.

With a copy of SPRITE THEATER in memory add these lines:

```
1096 FOR J=1024 TO 2023
1098 POKE J,(PEEK(J) AND 240) OR C1
1099 NEXT:GOTO 820
```

```
2055 RETURN
5031 GOSUB 1910  :REM GET SPRITES
5037 REM    ENABLE BIT GRAPHICS
5038 POKE U+24,PEEK(U+24) OR 8
5040 POKE U+17,PEEK (U+17) OR 32
```

Line 5031 contains something new: a GOSUB in the initialization to the LOAD SPRITE routine. If you are to use bit mapped graphics with sprites, you have to give up the flexibility of calling up new sprites at any time. With bit mapped graphics you must call up the sprites you are going to use only at the start of the program, before switching to bit mapped graphics mode. This is not such a severe restriction since you are allowed up to eight sprites at a time, any of which can be turned on or off. Also, you can exit the program and reenter RUN if you want to change sprites. This will not wipe out the scenery created by DRAWING BOARD.

Lines 1096–1099 provide for background color changes in bit mapped mode. Ordinarily, background color changes simply require one POKE to memory location 53281. In bit mapped graphics, background color comes from the screen area 1024–2043. Since the same character stores both foreground and background information, an AND function is required to separate out background data.

Line 5038 tells the computer to place the storage area for bit mapped graphics at starting location 8192. The operation OR 8 has the effect of setting the high bit of the low nibble at location U+24 to a 1.

Line 5040 sets bit mapped mode by setting the sixth bit at location U+17 with a POKE of 32.

Note that bit mapped graphics use the same base U as do sprite graphics. This is because they are both controlled by the same device, the VIC chip, which has all the memory locations (implemented as registers) for controlling sprites and bit mapped graphics.

Now let's see how bit mapped graphics works with sprites. Save the program in memory with the file name THEATER HI. This will let you distinguish this program from SPRITE THEATER. The significance of HI in the name will become clear later.

LOAD DRAWING BOARD into memory, and draw a simple object. Exit drawing board with RUN/STOP–RESTORE. LOAD THEATER HI and RUN it. The load menu will be displayed. Make entries for file SPRITE 1–8 just as you did previously. When all sprites are LOADed, they will be turned on, as expected. The surprise is that your object drawn with DRAWING BOARD also appears. It was not erased from memory when you exited DRAWING BOARD, only the screen image was erased. Both sprites and dot graphic objects are retained until overwritten or until the computer is turned off. This is

what allows us to combine objects created with different programs without having to pass variables between them.

A not so pleasant surprise that may leave you thinking that your program is bad is that the screen has a strange colored area running across it near the center. Other marks may appear on your screen that shouldn't be there. You can set your mind at rest because what you are seeing is not necessarily the fault of your program. It is due to the overlapping of the storage area for dot graphics and sprite graphics. Test this by selecting sprites and pressing the X key. Not only does the sprite change, but a section of the center area also reverses color. What you are seeing is the mapping of the sprites by bit mapped graphics. They are the same sprites, but the different mapping results in their being displayed in consecutive lines.

Press the f3 key. The background is painted a new color. You now have control over bit mapped background, which also becomes the background for the sprites.

SWITCHING MEMORY BANKS

By POKEing an 8 into U+24 the location of bit mapped graphics is set to start at 8192. Bit mapped graphics requires 8000 characters, so bit mapped graphics will extend up to 8192 + 8000, or memory location 16,192. In line 5022, LC, the starting location for sprites, is set at 12,288. It is easy to see that there is an overlap. If you are working with only one or two sprites, the few extraneous objects may be acceptable. But if you want proper displays, you are going to have to separate these memory areas.

Moving the sprites down in memory so they clear the dot graphics area would result in conflicts with BASIC, which is stored in low memory. The solution is to go to higher memory. But this requires *memory bank switching*. The computer can access only 16K of memory at a time. To extend its reach requires a switch to another memory bank of 16K characters. There are four such banks, providing a total of 64K of memory. Not all of this memory is equal. Some of it conflicts with the computer's requirements to do its own processing, such as getting the characters that it displays on the screen.

The bottom line is that we can get more memory but to do so we will have to give something up. We'll select the next higher bank of memory, and in doing so we'll give up the ability to display characters on the screen. The second memory bank that we will be using cannot access the ROM memory in which characters for display on the screen are stored. Since we have not included characters as part of our programs, this will not be a serious handicap. But it means working in an environment in which messages, including error messages, are displayed as gobbledygook and exiting the program becomes a special problem.

Here are the lines that will switch memory banks and allow you to exit the program gracefully:

```
1 REM ***** THEATER HI *****
892 IF K$="Q" THEN 1084   :REM QUIT
1080 REM
1081 REM * QUIT GRACEFULY *
1084 POKE 56578,PEEK(56578) OR 3
1086 POKE 56576,(PEEK(56576)AND252)OR3
1088 POKE 648,4:END
1096 FOR J=17408 TO 18407
2053 SP=0
5020 S0=18424:U=53248  :REM BASE LOC.
5022 H=30:V=225:C=5:LC=16384:RA=1
5032 REM
5033 REM * SWITCH MEMORY BANK *
5034 POKE 56578,PEEK(56578) OR 3
5035 POKE 56576,(PEEK(56576)AND252)OR2
5036 POKE 648,68
5042 FOR J=17408 TO 18407:POKE J,5:NEXT
5050 POKE U+39+J,C+J:POKE S0+J,J
```

The Q key is used to *quit the program*. If you tried to quit using the RUN/STOP–RESTORE combination that has worked up until now, you discovered that you lost control, and all screen messages from the computer were meaningless. Lines 1080–1088 let you quit gracefully by switching back to the normally used lowest bank of memory.

Lines 1084 and 5034 are identical. Their function is, in effect, to provide a path for the next POKE. (More specifically, they set port A of the complex interface adapter—CIA—chip for output.) Now lines 1086 and 5035 can POKE the bank of memory to be selected into memory location 56576 in the CIA. Here are the values to POKE for bank selection:

BANK SELECTION

Bank No.	Start Location	POKE Value
0 default	0	3
1	16384	2
2	32768	1
3	49152	0

As you can see, by POKEing 2 in line 5Ø35 we are switching to bank 1 so we will be working in the memory range from 16384 to 32767. Line 1Ø84 switches back to the original memory bank in which everything works normally.

One more POKE is required if we want to use the screen with the new bank. Lines 1Ø88 and 5Ø36 take care of this. You have to POKE the page number of the screen into location 648 so the *screen editor* will know where the screen is. A *memory page* is 256 characters, so we simply divide the starting address of screen memory by 256 to get the POKE value. We want the screen to be at the same relative position to the bank's base, or starting address. Therefore, the screen address will start at 16384 + 1Ø24, or 174Ø8. Then the page is 174Ø8/256, or 68, which is just what line 5Ø36 POKEs into memory location 648, the place in which the screen editor expects to find it.

We have to provide for moving the sprites to bank 1. To do this, in line 5Ø22, LC is changed from 12288 to 16384. This will place the sprites right at the bottom of bank 1, well out of the way of dot graphics, which retains its same relative position to the bank's starting address, namely 8192.

Line 5Ø2Ø contains the *pointer address* SØ for sprite Ø. Each sprite has such a pointer that tells the program where the sprite is stored in memory (not its position on the screen). The eight pointer locations are consecutive and are located in the last 8 characters of screen memory. Since we are locating screen memory at the same relative position but in the second bank, the pointers will start at 16384 + 2Ø4Ø, or 18424. This is the new value for SØ in line 5Ø2Ø.

The formula for the sprite pointer is

$$\text{Pointer} = \frac{\text{Sprite location} - (\text{Bank No.} * 16384)}{64}$$

The divisor 64 comes from the fact that the computer deals with sprite locations in multiples of 64, the size of each sprite. Our bank number is 1, our sprite location is 16384, giving a pointer value of Ø. This is what is entered in line 5Ø5Ø in place of the previous 192 (12288/64). The FOR. . .NEXT loop (lines 5Ø45–5Ø6Ø) takes care of all eight pointers. If the sprites were located at scattered positions (sprites can be located anywhere), we could not use a loop.

Lines 1Ø96 and 5Ø42 allow you to paint the screen with a selected color at the new screen location, 174Ø8. This provides the background color in bit mapped mode.

Notice how the use of *base locations* makes our task easier. If we had written the program using actual locations instead of offsets from a base, all affected locations in the program would have to be changed. Instead, we change only base locations, and all related locations are automatically shifted to the new memory bank.

With this as background, you can now explore bank switching on your own. You may get some weird results, but the worst you can do is hang-up the system, thus forcing you to turn off the switch and turn it on again. If you want to see some interesting drawing, try moving DRAWING BOARD into bank 2. This is where ROM resides. ROM is read only memory, which means you can read it but not write into it. The result is that DRAWING BOARD, when writing a dot, gets its information about the current character from ROM but must write the dot in RAM.

Before you can run THEATER HI, DRAWING BOARD must also be changed to draw in bank 1 rather than bank Ø. Some of the lines for the DRAWING BOARD change are the same as those you just added to THEATER HI; therefore, to save yourself trouble, try to keep them displayed on the screen during the following operations.

SAVE a copy of THEATER HI in a safe place and LOAD DRAWING BOARD into memory. Add to it the following:

```
1 REM ***** DRAW HI *****
892 IF K$="Q" THEN 1510
1120 FOR J=17408 TO 18407:POKEJ,CG:NEXT
1500 REM
1501 REM * QUIT PROGRAM GRACEFULY *
1510 POKE 56578,PEEK(56578) OR 3
1515 POKE 56576,(PEEK(56576)AND252)OR3
1520 POKE 648,4:END
5061 REM *  SET MEMORY BANK 1   *
5062 POKE 56578,PEEK(56578) OR 3
5063 POKE 56576,(PEEK(56576)AND252)OR2
5064 POKE 648,68
```

Lines 15ØØ–152Ø let you exit without having to turn off the computer, which, of course, would destroy anything you had drawn for THEATER. Lines 5Ø61–5Ø64 do the same bank switching that we discussed above.

LIST lines 8Ø, 5Ø2Ø, and 5Ø65, and make changes so they correspond to the following:

```
80 POKE 17408+RW*40+CH,CG:RETURN
5020 B=53265:BA=24576:B7=B+7:H=160:V=99
5065 FOR J=17408 TO 18407:POKE J,5:NEXT
```

These changes adjust the base values, as we did for THEATER. The variable V is reduced to 99 merely to prevent physical line overflow.

As with SPRITE THEATER, we will change the name of the DRAWING BOARD program that works in high memory. SAVE this program with the name DRAW HI. Then RUN it. Everything will work as before. Test the action of the f3 key. It should paint new colors on the background. Draw one or more objects for use with the sprites.

> **NOTE:** ***Don't press RUN/STOP.*** **Remember, you must exit with the Q key in order to retain the image you have drawn.**

When you press the Q key, you will get a mishmash of bars and lines. This is perfectly okay. *Now* you can press the key combination RUN/STOP–RESTORE to clear the screen.

LOAD a copy of THEATER HI. RUN it. When the prompt comes on the screen, LOAD the file SPRITE 1–8. After LOADing, the sprites will switch on, the figure you drew will be there, the screen will be clean, and everything you need to put on a great sprite show will be at your fingertips.

If after the computer is turned off, you run THEATER HI without first running DRAW HI, you will see serated lines in the screen's background. This is because DRAW HI has not cleared this area. Run DRAW HI before THEATER HI when you begin after the computer has been off.

TABLE 8 KEY ASSIGNMENTS FOR SPRITE THEATER

Type	Key	Operation
Direction		
	;	Move right
	H	Move left
	U	Move up
	M	Move down
	@	Move up and right
	Y	Move up and left
	/	Move down and right
	N	Move down and left
Control		
	+	Increment speed or pattern steps
	−	Decrement speed or pattern steps
	Space bar	Repeat last operation
	F	Change sprite pattern by bit
	D	Change sprite pattern by character
	S	Change sprite pattern by position
	A	Clear sprite pattern
	Z	Toggle sprite off/on
	T	Toggle sprite vertical expand/contract
	G	Toggle sprite horizontal expand/contract
Angles		
	L	Continuous move
	J	Rotate counter/clockwise
	K	Rotate clockwise
	=	Restore slope to 45 degrees

NOTE: If sprites don't move press the = key.

Type	Key	Operation
Color		
	f1	Border: cycle Ø–15
	f3	Background: cycle Ø–15
	f5	Sprite: cycle Ø–15
	X	Switch foreground and background color
Select sprite		
	1–8	Select sprite by number, counting left to right
Sound		
	O	Octave: cycle 1–8
	W	Waveform: cycle noise, sawtooth, sound off
Read sprite from disk or tape		
SHIFT/L		Initiate a read operation

10

A New World of Communication

Hello world!

The advent of low-cost computers has launched a new era in communications. Developments are occurring at such a rapid rate that we are experiencing a communications revolution of greater dimensions than that launched by the telephone. Low-cost computers in the home and office connected to highly sophisticated communication networks are giving us new approaches to work, education, and recreation. Your Commodore 64 is a window to this fascinating new world.

Your Commodore can be connected to the vast telephone network, allowing you to communicate all across the land with other owners of small computers or to access the rapidly increasing number of large computers dedicated to providing all kinds of services to people like you.

Although you will still be using the familiar telephone network, you will now be seeing and keying instead of talking and listening. And instead of the telephone with no memory or intelligence, you have a powerful computer that can remember messages, work by itself, make decisions, transfer programs and data, display or print messages, and still act as a computer when it's not serving as a communication terminal.

Here are some of the things computer owners are doing over the telephone lines:

- □ Exchanging information about computers and programs
- □ Receiving programs from information services
- □ Receiving stock market reports and prices from financial services

- □ Shopping for products
- □ Participating in games with other owners

Plugging into the new communication era is neither difficult nor expensive. Selecting from the many offerings, however, is not so easy. In this chapter we will present a survey of what's available, typical costs, and what you need in terms of hardware and software. We will also give you an overview of how one computer communicates with another.

HOW COMPUTERS COMMUNICATE

Computers have neither voice nor ears. How then do they communicate over a telephone system that was designed to transmit the human voice? This is accomplished through the use of a *modem*, a device that translates a computer's voltage pulses into the kind of signals with which the telephone system can deal. You have heard of AM and FM radio: AM stands for *amplitude modulation* and FM stands for *frequency modulation*. In radio and TV, *modulation* is a process of translating one type of signal to another type. The computer, like a radio or TV, requires a device to send messages to the world at large. The modem, derived from the words *mo*dulator and *dem*odulator, is such a device.

You can buy a Commodore modem from any Commodore dealer. The modem is literally a black box, about 6 × 4 × 1 inch that plugs directly, that is without a cable, into one of the sockets in back of your Commodore 64. Two cables are attached to the side of the modem for connecting your computer between the telephone and telephone line.

Inspect your telephone to see if it has a little plug connecting it to the telephone line. The plug has a small lever, which is easy to spot. This lever is a latch. If you press it toward the plug, the plug can be easily disconnected from the phone. You may have a similar connection at the small box on the wall where the telephone line enters the room. If you have this kind of telephone connector, you can easily connect to the Commodore modem, which is designed to accommodate this type of plug. Just disconnect your phone from the line by removing the plug. Then insert a similar plug from the modem in the phone and the phone line in the modem. It all takes less than a minute.

The Commodore modem will work with either dial or push button phones. In fact, when the modem is connected and working, you don't use the telephones to enter a number. Instead, you enter the telephone number on the keyboard of the computer. The modem then sends the signals for calling the person at that number.

If your phone does not have the removable connector described above, call your phone company to determine what options you have and their costs. Tele-

phone equipment is fairly standardized. One way or another, it will be possible for you to get the proper connection.

How does having the modem inserted between your phone and the line affect voice telephone service? There is a switch on the side of the modem that lets you choose the type of communication—voice or computer. You can have one or the other but not both at the same time. When the switch is set for the computer, you cannot call people for a conversation and they cannot call you. If somebody tries to call your number they will get a busy signal, so you are locked out of all incoming calls.

Another item to consider in planning for the use of a modem is the location of the telephone. How will its hookup to the computer affect its use for voice conversations? There is about a 6½ foot cable to connect the modem to the phone. Check the length of the cable that connects your phone to the wall outlet. With these two lengths known, you can determine possible locations for the phone and if you need longer cables. Remember, the modem does the dialing and it's not necessary to lift the handset when using the modem. Therefore, close access is not as necessary for computer communication as it is for voice.

When you talk to someone, communication seems like such a simple process that you might be surprised to learn communication between electronic devices is quite complex. When you hear someone make a mistake, not speak clearly, or say something you don't understand, you can easily interrupt and ask for clarification. All this is much more difficult with electronic equipment.

Here are just a few things with which communicating devices must deal just to send messages reliably from one point to another:

- Error detection
- Notification to the sender when an error is detected
- Notification to the receiver that a message is on its way
- Notification to the sender that the receiver is ready to receive a message
- The speed at which messages are transmitted
- The method whereby the roles of sender and receiver are reversed
- The method for coding characters into the Ø's and 1's that computers and communication circuits use

You can see that there is more to communication than hooking two pieces of equipment together with a pair of wires. In practical terms, it means that for two devices to communicate they must be *compatible*. That is, there must be agreed upon ground rules on how to handle the kinds of things listed above. These ground rules are the *communication protocol*, and the individual items that make up the protocol, such as those listed above, are the *communication parameters*.

There are many types of communication equipment and many protocols. This means that just because you have a computer and a modem it doesn't follow that you can talk to any other computer/modem combination. The other station may run at a different speed, use different type signals, expect a different mode of answer back, and on and on. Most communication equipment today, including Commodore's, allow you some selection of parameters. Otherwise, you would be very limited as to whom you could talk to.

In addition to a modem, you must have a *communication program*. The communication program RUNs in your computer just like any other program. While it is RUNning, your computer is fully occupied and you cannot RUN another program at the same time.

Communication programs vary widely in what they do and how they do it. Basically, in the *receive mode* they assemble the characters as they are received, store them in memory, and send the necessary control codes to the other station. In the *transmit mode* they perform similar functions in reverse order. Many additional functions are often added to these basic operations, such as automatic dialing, storage on disk, and the ability to set the communication parameters so you can communicate with more than one type of protocol.

Not long ago modems did nothing but translate signals from one form to another. Recently, *smart modems* have appeared that incorporate some of the functions of communication programs. For example, the Commodore modem performs automatic dialing and automatic answering. Commodore calls their modem, Automodem.

A communication program is included with the Commodore modem. This program is on tape, and you will need a tape unit attached to your computer to LOAD it. If you have a disk, check with your dealer on how to LOAD the communication program. You don't necessarily have to use this program, but you must be sure that any substitute will work with your equipment.

The Commodore modem and communication program turn your computer into a communication terminal that can talk to many other computer owners and information services. Below is a list of the capabilities of the Commodore modem–communication program combination. You will be able to talk with any station operating with the same capabilities. An explanation of unfamiliar terms follows, but a detailed understanding of these terms is not necessary in order for you to use your computer to communicate.

COMMODORE MODEM–COMMUNICATION PROGRAM CAPABILITIES

Parameter	Value
Signal rate	11Ø, 15Ø, or 3ØØ baud
Call mode	Originate or answer
Transmission mode	Half or full duplex

Parameter	Value
Parity	None, even, odd, mark or space
Word length	5, 6, 7, or 8
Stop bits	1 or 2
End of line	Linefeed or carriage return
Character code	ASCII or Commodore 64
Color change	Yes or no
End of line break	Full word or broken word

The Commodore communication program easily allows you to select from these parameter values those that match the computer with which you are communicating. Here is an explanation of what these parameters do.

Signal Rate

Signal rate determines the speed at which messages and other data are transferred between computers. Signal rate is measured in *bauds*. For example, 3ØØ bauds means that 3ØØ signals are transmitted per second. Signals may take several forms. The most common are voltage changes and frequency changes. The Commodore modem, like most low-speed modems, sends signals by switching between two frequencies.

Each character will usually have 10 possible signal changes, sometimes called *bits*. Therefore, 3ØØ baud corresponds to 3Ø characters per second. This is the fastest rate at which the Commodore modem can run.

Signal rate is important because a long distance call is required for you to connect to most information services. These charges are based on connect time, and the faster you can communicate the lower your line costs will be. 3ØØ baud is about the lowest practical transmission speed. If you need to transfer a large amount of data, you may want to switch to higher speed modems that work at 12ØØ baud. However, for recreational computing, the 3ØØ-baud Commodore modem should serve you well.

Call Mode

Before communication is established between computers, one must do the calling, that is, "dial" the other's telephone number. The called computer, in addition to being up and running, must be "listening" for a call so that it can respond to the telephone "ring." These two modes of operation are called *originate* and *answer*. The *call mode* is the only parameter in the above list not set by the program. Instead, there is a switch on the side of the modem that lets you switch between the originate and answer modes. If you are calling an informa-

tion service or a friend's computer, you would set the switch to the originate mode. If you expect a call from a friend, you would set the switch to answer.

Transmission Mode

Transmission mode refers to the way the send and receive functions are interchanged. There are two methods: *half duplex* and *full duplex*.

In half duplex mode, there is only one channel for communication so only one computer can send at a time. The receiving computer must wait for the transmitter to stop, then the roles of the computers can be reversed. If there are any errors, the transmitter will not know about them until completion of transmission, possibly a long and costly delay.

In full duplex mode there are two channels, so both computers can send and receive at the same time. The receiver can thus give the transmitter immediate knowledge that errors have been received or can interrupt transmission to make adjustments such as changing disks or checking instructions.

There are many ways of using the flexibility of full duplex mode. One common method of operation is to echo back all characters transmitted. When the sender receives the echo, the echo can be compared with the original to check for errors or lost data.

How can two computers send and receive at the same time over one telephone line? The line is divided into two parts, or *channels*, by allocating different frequencies to each computer. Just as FM stations are distinguished by frequency, two computers can use different frequencies to "talk" back and forth.

The computer that has its switch set to originate mode transmits signals by shifting between frequencies of 127Ø and 1Ø7Ø hertz (cycles per second). The computer set for answer mode shifts between 2225 and 2Ø25 hertz. This shifting of frequencies, called *frequency shift modulation*, is similar to the Ø's and 1's stored in the computer. However, from old telegraph terminology, the lower of the two frequencies is referred to as *space* and the higher of the two as *mark*.

Both the computer set for originate mode and the computer set for answer mode can serve as either transmitter and receiver. Originate and answer refer only to establishing the telephone connection, not to the sending and receiving of messages.

Parity

Parity refers to how computers check for errors. We pointed out above that checking is sometimes performed by echoing characters back in full duplex mode. Parity checks are performed by checking the character at the receiver. To make this possible, an extra bit is added to each character to make the total number of 1's (marks) either an even number (for even *parity*) or an odd num-

ber (for *odd parity*). Both even and odd parity are widely used, so the Commodore communication program lets you pick either type. Other options for this parameter are *mark parity* (highest bit always on), *space parity* (highest bit always off), and *no parity* (no extra bit added).

If your parity setting does not match that of the other computer, characters will not be received correctly, as indicated by obvious errors in screen printing.

Word Length

You know that characters in your computer are stored as combinations of eight Ø's and 1's, that is, 8 bits, or 1 byte, per character. Some communication equipment also uses 8 bits to store a character, others use 5, 6, or 7 bits. The number of bits per character is called the *word length*. The larger the word length, the greater the number of different characters the equipment can handle. Word lengths of 7 and 8 are the most common in computer communications. The older printer terminals used word lengths of 5 and 6.

Stop Bits

If characters were transmitted as a continuous series of 1's and Ø's, the equipment could easily lose track of where one character ended and the next started. Knowing where a character started would be particularly difficult at the start of a transmission. This problem is solved by adding bits to the beginning and end of each character. One bit, called the *start bit*, is added at the beginning of each character. One or two bits, called *stop bits*, may be added at the end of each character.

End of Line

Computers differ in the symbols they use to detect the *end of a line*. As you know, when you want to end a line on your Commodore, you press RETURN. This causes a line feed and a carriage return operation. Each of these operations has its own code. When receiving data some computers expect to see only the code for line feed. Others require both line feed and carriage return. You can select either of these options.

Character Code

Character code is the way characters are translated into Ø's and 1's for use in computers. There are many possible ways this can be done, but for two computers to talk to each other they must use the same character codes, just as two people must converse in a language that they both understand.

In order to facilitate commonality between equipment, standard character codes have been adopted so that manufacturers will produce equipment that can talk to each other. The most commonly used standard is the ASCII code discussed in Chapter 1. However, Commodore uses a different code, which is a modified ASCII code. You can talk to other Commodore users with this modified ASCII code, but to talk to anyone else requires translation to pure ASCII. This translation is no problem; it is performed by the communication program whenever you select the standard ASCII character code option.

Color Change

Computer communication often consists of many short interchanges between your computer and a distant computer. You send a message then receive a reply. This is repeated several times and the screen becomes full of messages. But which are yours and which are the other person's? The *color change* option lets you display your messages in one color and the other person's messages in a different color. This is the *two color option*. The alternative is to display all messages in the same color, the *one color option*.

End of Line Break

Data are printed on your screen as they are received from a distant computer. Normally, if part of a word has been printed at the end of a line, the word will be broken and the remainder printed on the next line. The result is an unsightly text that is difficult to read. This can be corrected by selecting the *full word option*. Then words are moved to the next line rather than breaking them up.

INFORMATION SERVICES

Information services, or *utilities*, store everything from stock quotations to fantasy games in big computer centers that are connected to all major cities and towns throughout the country. For an initial fee and an hourly charge, you can talk to these big computers with your small computer. It is as if you had your own big computer, because you get the benefits of all that computing power and all the accumulated knowledge that is stored in the big computer's databases.

Three of the major information utilities are the SOURCE,™ CompuServe,™ and Dow Jones News/Retrieval. There is much overlapping in what these services offer, especially the first two. As you might expect, Dow Jones News/Retrieval focuses on news reports and financial information for business. At the time of this writing, Commodore included a free hour connect time to both CompuServe and Dow Jones News/Retrieval with the purchase of the automodem. This is hardly enough to explore all the offerings but an hour of browsing

through their menus is enough for you to decide if the service is for you. Here is a rundown of the services provided, broken down into the following categories: communication, finance and business, travel, news, shopping, and games.

Communication

Electronic Mail Appropriately enough, electronic mail uses electronic mailboxes to store messages. If you want to send a message, you enter it on your keyboard and it is delivered to the mailbox of the person to whom you address it. Sometime later that person, who may be next door, across the country or in Canada, will sit down at his or her computer and ask for a display of that day's mail. After reading your message, a reply will be entered and sent to your mailbox, and you will read it the next time you sign on to the information service.

Electronic mail is used by business and professional people, as well as hobbyists and other people for special purposes.

Electronic Bulletin Boards The classified ad goes electronic! You can enter any item, serious or not. You have no way of knowing who will read it, but bulletin boards are one of the most popular uses of information services. Sell your house, oil wells, or, we hope not, your Commodore 64. Or just tell the world what you think of it. There will probably be a response the next day.

Electronic Conversations Electronic conversation enables you to use your keyboard to communicate with other computer owners. You enter a message and somebody replies—somebody you may have never met before. You talk back and forth with your keyboards and computer screens. This is a way of meeting other users and exchanging information.

Finance and Business

Financial and business reports is where you find that small company that's going to be the next Xerox. There's enough statistics to keep you occupied and away from computer games far into the next century. Stock prices, commodity prices, stock performance histories, company financial reports, all the software necessary to turn this data into something meaningful, and even graphic output, is available if you've got the equipment to display or print it.

Travel Services

Going somewhere? Be your own travel agent. Check the airline schedules on your computer screen. Then make reservations for plane, car, and hotel. Should

you take an umbrella? Check the local weather. Any revolutions planned? Check the local news. Bring your computer with you? You bet!

News Reports

Up-to-the-second news stories are available. You can search by keyword for just those items of interest to you, read syndicated columnists, and check on your favorite sports events. In addition to 90 second news breaks, Dow Jones News/Retrieval lets you access older news items going back as far as 90 days.

Electronic Shopping

Tired of pushing shopping carts through crowded aisles? Relax at your keyboard as you browse through catalogs describing TV sets, cameras, watches, computer programs, computer equipment, and more. Place your order. Discounts of 20–40 percent may be available.

Games

Here is a new dimension to electronic games. Play against the computer or against other game enthusiasts. Adventure games and games for all ages are available.

In addition to this listing, each service has special offerings. For example, the SOURCE has what they call an Electure™—an electronic lecture—that uses teleconferencing to bring interested people together to discuss issues of interest. A prominent person is selected as guest electurer. Subscribers can read the electure, ask questions, make comments, and participate in voting on the issue. The whole thing lasts about a month.

Table 9 at the end of this chapter lists the major information services and what you can expect to pay for them. You must also take into consideration the cost of using the telephone lines. Since the major information services may be a long way from your town, does that mean you will have enormous phone bills? Not necessarily. You can access an information service in one of two ways if you don't live in the immediate vicinity. Some services such as CompuServe have their own network. The CompuServe network has access points in most major cities. If you live in an access area you can connect directly to the network by dialing the local phone number for CompuServe.™ If you are outside of a local rate area, you have two options. You can call long distance or you can link up through a communication network firm such as Telenet™ or Tymnet™ in the United States or DataPack™ in Canada. These firms are in the business of providing communication facilities. They therefore have communication networks that are much more extensive than the information utilities. The chances are good that there is an access point to a communication network near where you

live. In that case, you call their number and they will see that your messages get routed to the information service. For this you pay a surcharge on top of the standard access charged by the information service.

If there is no local access to a communication network in your vicinity, you must call long distance to the nearest entry point. Communication networks are always expanding, so keep trying.

New computer buying services pop up and disappear daily. What most of these offer is discount prices and the ease of shopping from your home. Products vary from antique cars to groceries. Payment is almost always by credit card.

The services we have been describing are geared to a wide audience and are provided at relatively low mass market prices. There is another category of computer service that specializes in narrow areas of interest and charge high fees. They maintain databases in legal, medical, financial, and technical subjects of interest to professionals and specialists. If you are such a person, you are probably aware of such services in your area of interest. If not, you can investigate further, starting at your local library.

TABLE 9 INFORMATION SERVICES

Service	Typical Charges for Basic Service	
CompuServe 5000 Arlington Centre Blvd. P.O.B. 20212 Columbus, OH 43220 (800) 848-8990	8 AM–6 PM weekdays $12.50 per hour 6 PM–5 AM $6.00 per hour	
The SOURCE 1616 Anderson Rd. McLean, VA 22102 (703) 734-7500	Registration fee $100 7 AM–6 PM weekdays $20.75 per hour 6 PM–7 AM $7.75 per hour	
Dow Jones/News Retrieval P.O.B. 300 Princeton, NJ 08540 (800) 345-8500	Standard subscription One-Time Fee $75.00	
	6 AM eastern time–6 PM local time	Other times
Business and Economic News	$1.20 per min.	$0.20 per min.
Quotes	$0.90	$0.15
Financial and Investment	$1.20 per min.	$0.90 per min.
General News	$0.60 per min.	$0.30 per min.

APPENDIX A

Turning Bits On and Off

Many control locations in memory are single bits. For example, when generating computer sounds, each sound is turned on and off by writing to a single bit of memory. But every memory read and write must be a full character. Individual bits cannot be read and written. For example, if we tried to turn on a sound by turning on 1 bit and writing a 1 to that bit's memory location, the computer would actually write ØØØØØØØ1, one full character of 8 bits. The result would set the sound bit on, but it would also turn all other bits off. These other bits are used to control other aspects of the sound that we might want to have set to 1. We must find a way to set 1 bit of a character without changing any of the other bits.

The first step is to read the character from memory with a PEEK. Once the character is out of memory and in the computer, the computer does have the ability to operate on individual bits. It does this with things called OR and AND operators. The OR operator is used to turn a bit on (set it to 1). The AND operator is used to turn a bit off (set it to Ø). Here are examples of statements that turn bits on and off:

```
POKE 12288,PEEK (12288) OR 2^Ø
POKE 12288,PEEK (12288) AND 255-2^Ø
```

The first part of these statements tell the computer to read the character at location 12288 and to write a new character back in the same location. The new character is generated from the PEEKed character by the OR and AND operators. In the expression 2^Ø, the circumflex character, ^, signifies that 2 is to be

raised to a power Ø. Any character raised to a power of Ø is 1, so the first statement ORs the PEEKed number with ØØØØØØØ1.

The OR operator lines up two characters, call them A and B, and compares corresponding bits. It then creates a new character as follows: If the bit in A is a 1 OR the bit in B is a 1, then the corresponding bit in the new character is a 1.

Let's see how this might work. Assume that when the computer PEEKs into location 12288 it finds the number 254, which is stored in the computer in the form 1111111Ø. This is because the computer stores everything in terms of two voltages, which we represent on paper as 1's and Ø's. Numbers are stored in bytes, and bytes are composed of bits. The rightmost bit has the lowest value; bits increase in value by a factor of 2 for each leftward position, just as the numbers with which you are familiar increase by a factor of 1Ø for each leftward position. Since 255 is the largest number that a byte can store, it makes sense that all bits but the leftmost are set to 1 for 254.

Line up 1111111Ø with ØØØØØØØ1. Now each new bit is a 1 if either the bit from A is a 1 OR the corresponding bit from B is a 1. As the new byte we have 111111111. We have turned on the last bit without affecting any of the other bits. Now if we POKE this byte back into 12288, our objective will have been accomplished.

The AND operator works in a similar manner, but it only puts a 1 in the new character if there is a 1 in A and B. Assume this time that the character read by PEEK is 255 or, in computer language, 11111111. According to the above statement, this gets ANDed with 255 minus 2 to the Ø power (255 − 2ˆØ; 2ˆØ = 1) or 254. In computers, 254 is 1111111Ø. All bits have 1's in A and B except the last, so the new character has 1's in all bits except the last. Result: The last bit is turned off without affecting other bits.

The power of 2 is used to select the bit that is turned on or off. In the number system with which you are familiar, each successive digit is a power of 1Ø because there are 1Ø symbols, Ø–9. In the number system in the computer, there are only 2 symbols, Ø and 1. Therefore, each successive position increases by a power of 2. Here are the relationships between the bit position and its value as a power of 2:

Bit position	7	6	5	4	3	2	1	Ø
Power of 2	128	64	32	16	8	4	2	1

If you add up all the powers of 2's, you will get 255, the highest possible value. Lower values are obtained by adding only those powers of 2 that have their bit position set to 1. For example, the computer number ØØØØØ1Ø1 has only bit positions 2 and Ø turned on. We can translate that to our numbers by adding the corresponding 4 and 1 from the above chart, giving us 5. Therefore, the computer's ØØØØØ1Ø1 is equivalent to our 5.

Suppose we wanted to turn bit position 4 off or on. By using 2 to the fourth

power we get 16, which in the computer is ØØØ1ØØØØ. This is the number that would be ANDed or ORed with the current contents of the byte to create a new byte. Here are some examples:

Character from PEEK	11ØØ1ØØØ	11Ø11ØØØ	1Ø111111	11ØØØØØØ
Bit pos. to change	4	4	6	6
Bit value	16	16	64	64
Bit character	ØØØ1ØØØØ	ØØØ1ØØØØ	Ø1ØØØØØØ	Ø1ØØØØØØ
Result of OR	11Ø11ØØØ	11Ø11ØØØ	11111111	11ØØØØØØ
255-bit value	239	239	191	191
Bit character	111Ø1111	111Ø1111	1Ø111111	1Ø111111
Result of AND	11ØØ1ØØØ	11ØØ1ØØØ	1Ø111111	1ØØØØØØØ

Notice that the bit character has all 1's and Ø's interchanged when the bit value is subtracted from 255. This is called *complementing a number*. Also, the bit character is either one 1 and the rest Ø's or one Ø and the rest 1's, which is what we want in order to identify a single bit. In two of the examples, bits are turned on when they are already on, so there is no change; in two other cases bits are turned off when they are already off, again there is no change.

APPENDIX B
Programs

SPIRAL

```
1 REM *** SPIRAL ***
20 X=2:PRINT CHR$(147):REM CLEAR SCREEN
25 K=78:V=X:Y=1:H2=40-X
30 H1=1:V1=V:V2=25-X:H3=1:V3=1
34 REM
35 REM HORIZONTAL LINE
40 FOR H=H1 TO H2 STEP H3:GOSUB 400
45 NEXT H:H=H-Y
49 REM
50 REM VERTICAL LINE
55 FOR V=V1 TO V2 STEP V3:GOSUB 400
60 NEXT V:V=V-Y
64 REM
65 REM * SWAP PARAMETERS *
70 TH=H1:TV=V1:H1=H2:V1=V2:Y=-Y
75 H2=TH+X:H3=-H3:V2=TV+X:V3=-V3:X=-X
80 IF X>0 AND V2<V1 THEN 90
85 GOTO 40
90 GOSUB 520
95 GOTO 25
400 REM
401 REM ** POSITION & COLOR **
420 IF H<1 OR H>40 THEN STOP
```

```
425 IF V<1 OR V>25 THEN STOP
430 S=(H-1)+(V-1)*40+1024
435 POKE S,K:POKE S+54272,C
440 RETURN
500 REM
501 REM  ** SUB.: COLOR KEYS **
520 GET K$:IF K$="" THEN 520
545 IF K$="D" THEN K$="9"  :REM ORANGE
550 IF K$="C" THEN K$="10" :REM BROWN
555 IF K$="E" THEN K$="11" :REM LT. RED
560 IF K$="Q" THEN K$="12" :REM DK. GRY
565 IF K$="A" THEN K$="13" :REM MD. GRY
570 IF K$="Y" THEN K$="14" :REM LT. GRN
575 IF K$="U" THEN K$="15" :REM LT. BLU
580 IF K$="Z" THEN K$="16" :REM LT. GRY
595 IF K$=CHR$(133) THEN 660 :REM F1
600 IF K$=CHR$(134) THEN 675 :REM F3
648 REM
649 REM * SET CHARACTER COLOR *
650 IF VAL(K$)>16 OR VAL(K$)=0 THEN 520
652 C=VAL(K$)-1     :REM COLOR N0.
655 RETURN
658 REM
659 REM * SET BORDER COLOR *
660 C0=C0+1:IF C0>15 THEN C0=0
670 POKE 53280,C0:GOTO 520
673 REM
674 REM * SET BACKGROUND COLOR *
675 C1=C1+1:IF C1>15 THEN C1=0
680 POKE 53281,C1:GOTO 520
695 GOTO 520
```

KEY ENTRY

```
500 REM
501 REM ** SUB.: KEY ENTRY **
520 GET K$:IF K$="" THEN 520
525 K1=ASC(K$):IF K1<192 THEN 535
530 K=K1-128:RETURN   :REM GRAPHICS RT.
535 IF K1<161 THEN 545
540 K=K1-64:RETURN    :REM GRAPHICS LFT
545 IF K$="D" THEN K$="9"  :REM ORANGE
550 IF K$="C" THEN K$="10" :REM BROWN
555 IF K$="E" THEN K$="11" :REM LT. RED
560 IF K$="Q" THEN K$="12" :REM DK. GRY
565 IF K$="A" THEN K$="13" :REM MD. GRY
570 IF K$="Y" THEN K$="14" :REM LT. GRN
575 IF K$="U" THEN K$="15" :REM LT. BLU
580 IF K$="Z" THEN K$="16" :REM LT. GRY
585 IF K$="+" THEN W=W+1:GOTO 520
587 IF K$="-" THEN W=W-1:GOTO 520
590 IF W<0 THEN W=0
595 IF K$=CHR$(133) THEN 660 :REM F1
600 IF K$=CHR$(134) THEN 675 :REM F3
605 IF K$=CHR$(135) THEN 700 :REM F5
610 IF K$=CHR$(136) THEN 710 :REM F7
648 REM
649 REM * SET CHARACTER COLOR *
650 IF VAL(K$)>16 OR VAL(K$)=0 THEN 520
652 C=VAL(K$)-1     :REM COLOR NO.
655 RETURN
658 REM
659 REM * SET BORDER COLOR *
660 C0=C0+1:IF C0>15 THEN C0=0
670 POKE 53280,C0:GOTO 520
673 REM
674 REM * SET BACKGROUND COLOR *
675 C1=C1+1:IF C1>15 THEN C1=0
680 POKE 53281,C1:GOTO 520
695 GOTO 520
698 REM
699 REM * SELECT CORNERS OR CENTER *
700 F1=-F1:GOTO 520
708 REM
709 REM * SET SPACING *
710 S1=S1+1:IF S1>3 THEN S1=1
715 GOTO 520
```

LIGHT SHOW

```
1 REM ***** LIGHT SHOW *****
20 PRINT CHR$(147) CHR$(5)
21 POKE 53272,21
22 PRINT "                LIGHT SHOW"
25 POKE 53280,3:POKE 53281,2
30 PRINT:PRINT
35 PRINT SPC(9) "+=INCR. -=DECR. DIMND.
40 PRINT SPC(9) "F1=BORDR, F3=BACKGRND"
45 PRINT SPC(9) "F5=SWITCH CORNERS/CNTR
50 PRINT SPC(9) "F7=SPACE 0,1 OR 2"
55 PRINT:PRINT:PRINT
60 PRINT SPC(9)"PRESS ANY KEY TO START"
62 PRINT SPC(9)"THEN PRESS A COLOR KEY"
65 GET A$:IF A$="" THEN 65
70 PRINT CHR$(147)          :REM  CLEAR
80 K=160:S1=1:W=6:F1=-1
95 REM
100 REM ***** MAIN LINE *****
101 REM
105 GOSUB 520
110 IF F1=1 THEN 180 :REM GO TO DIAMOND
115 H1=02:H3=+S1:HB=12:HE=2:HS=-1:V=2
120 GOSUB 230
125 H1=39:H3=-S1:HB=28:HE=38:HS=+1:V=2
130 GOSUB 230
135 H1=02:H3=+S1:HB=02:HE=12:HS=+1:V=14
140 GOSUB 230
145 H1=39:H3=-S1:HB=38:HE=28:HS=-1:V=14
150 GOSUB 230
170 GOTO 105
175 REM
180 CE=20:LB=5:LE=16:LS=1:V=2
185 GOSUB 280
190 CE=20:LB=15:LE=05:LS=-1
195 GOSUB 280
210 GOTO 105
215 REM
216 REM ***** SUBROUTINES *****
220 REM
225 REM ** DRAW TRIANGLE **
230 FOR H2=HB TO HE STEP HS
235 FOR H=H1 TO H2 STEP H3
240 GOSUB 420
```

```
245 NEXT H:V=V+1:NEXT H2
250 RETURN
270 REM
275 REM ** DRAW DIAMOND **
280 FOR HL=LB TO LE STEP LS
285 FOR H=CE-HL TO CE-HL+W STEP S1
290 GOSUB 420
295 NEXT H
300 FOR H=CE+HL-W TO CE+HL STEP S1
305 GOSUB 420
310 NEXT H
315 V=V+1:NEXT HL
320 RETURN
400 REM
401 REM ** POSITION & COLOR **
420 IF H<1 OR H>40 THEN H=20:W=1
425 IF V<1 OR V>25 THEN STOP
430 S=(H-1)+(V-1)*40+1024
435 POKE S,K:POKE S+54272,C
440 RETURN
500 REM
501 REM ** SUB.: KEY ENTRY **
520 IF ML=1 THEN 920
522 GET K$:IF K$="" THEN 520
525 K1=ASC(K$):IF K1<192 THEN 535
530 K=K1-128:RETURN    :REM GRAPHICS RT.
535 IF K1<161 THEN 545
540 K=K1-64:RETURN     :REM GRAPHICS LFT
545 IF K$="D" THEN K$="9"  :REM ORANGE
550 IF K$="C" THEN K$="10" :REM BROWN
555 IF K$="E" THEN K$="11" :REM LT. RED
560 IF K$="Q" THEN K$="12" :REM DK. GRY
565 IF K$="A" THEN K$="13" :REM MD. GRY
570 IF K$="Y" THEN K$="14" :REM LT. GRN
575 IF K$="U" THEN K$="15" :REM LT. BLU
580 IF K$="Z" THEN K$="16" :REM LT. GRY
585 IF K$="+" THEN W=W+1:GOTO 520
587 IF K$="-" THEN W=W-1:GOTO 520
590 IF W<0 THEN W=0
592 IF K$<>" " THEN 595
593 ML=1-ML:K$="+":GOTO 920
595 IF K$=CHR$(133) THEN 660 :REM F1
600 IF K$=CHR$(134) THEN 675 :REM F3
605 IF K$=CHR$(135) THEN 700 :REM F5
610 IF K$=CHR$(136) THEN 710 :REM F7
648 REM
```

```
649 REM * SET CHARACTER COLOR *
650 IF VAL(K$)>16 OR VAL(K$)=0 THEN 520
652 C=VAL(K$)-1      :REM COLOR NO.
655 RETURN
658 REM
659 REM * SET BORDER COLOR *
660 C0=C0+1:IF C0>15 THEN C0=0
670 POKE 53280,C0:GOTO 520
673 REM
674 REM * SET BACKGROUND COLOR *
675 C1=C1+1:IF C1>15 THEN C1=0
680 POKE 53281,C1:GOTO 520
695 GOTO 520
698 REM
699 REM * SELECT CORNERS OR CENTER *
700 F1=-F1:GOTO 520
708 REM
709 REM * SET SPACING *
710 S1=S1+1:IF S1>3 THEN S1=1
715 GOTO 520
900 REM
901 REM *** AUTOMATIC MODE ***
920 GET K$
925 IF K$=" " THEN ML=1-ML:GOTO 520
930 REM
935 REM * SET TIME DELAY *
940 IF K$="+" THEN TM=TM+900
945 IF K$="-" THEN TM=TM-900
950 IF TM<0 THEN TM=0
955 FOR J=0 TO TM:NEXT
960 REM
965 REM * GET RANDOM OPERATION *
970 J=INT(RND(0)*1000) :REM RANDOM NO.
972 IF J>255 THEN K$="+"
973 IF J>379 THEN K$="-"
975 IF J>503 THEN K$=CHR$(133)
977 IF J>627 THEN K$=CHR$(134)
979 IF J>751 THEN K$=CHR$(135)
985 IF J>875 THEN K$=CHR$(136)
990 IF J<256 THEN K$=CHR$(J)
995 GOTO 525
```

MUSICAL KEYS

```
1 REM  **** MUSICAL KEYS ****
2 REM  HUNT AND PECK KEY ENTRY
10 GOTO 5020  :REM BY-PASS SUBROUTINES
11 REM
12 REM ** SUB: TURN NOTE ON/OFF **
13 REM * NOTE OFF *
20 POKE M+4,WF
25 FOR Z=1 TO TP:NEXT
30 REM
35 REM * NOTE ON *
40 NT=INT(NT):IF NT>64814 THEN NT=1072
45 HF=INT(NT/256):LF=INT(NT-256*HF)
50 POKE M,LF:POKE M+1,HF
55 POKE M+4,WF+1
60 FOR Z=1 TO TN:NEXT
65 RETURN
800 REM
801 REM ******************************
802 REM *** SUB. SOUND KEY ENTRY ***
820 GET K$:IF K$="" THEN 820
830 IF K$="A" THEN NT=3608
832 IF K$="B" THEN NT=4050
834 IF K$="C" THEN NT=4291
836 IF K$="D" THEN NT=4817
838 IF K$="E" THEN NT=5407
840 IF K$="F" THEN NT=5728
850 IF K$="G" THEN NT=6430
860 RETURN
5000 REM
5001 REM **** START PROGRAM ****
5010 REM
5011 REM * INITIALIZE SOUND *
5020 M=54272:AM=15:N8=1:PRINT CHR$(147)
5030 WF=32
5040 FOR J=M TO M+24:POKE J,0:NEXT
5050 POKE M+5,9:POKE M+6,0:POKE M+24,15
5100 REM
5101 REM *** MAIN LINE ***
5120 GOSUB 820
5130 GOSUB 20
5140 GOTO 5120
```

MUSICAL KEYBOARD

```
1 REM **** MUSICAL KEYBOARD ****
2 REM  SOUND AND MUSIC SYNTHESIZER
10 GOTO 5020
11 REM
12 REM ** SUB: TURN NOTE ON/OFF **
13 REM * NOTE OFF *
20 POKE M+4,WF
25 FOR Z=1 TO TP:NEXT
30 REM
35 REM * NOTE ON *
36 IF ML=1 THEN 45
37 IF NT>4050 THEN NT=NT*N8:GOTO 40
38 NT=NT/N8
40 NT=INT(NT):IF NT>64814 THEN NT=1072
45 HF=INT(NT/256):LF=INT(NT-256*HF)
50 POKE M,LF:POKE M+1,HF
55 POKE M+4,WF+1
60 FOR Z=1 TO TN:NEXT
65 RETURN
800 REM
801 REM ******************************
802 REM *** SUB. SOUND KEY ENTRY ***
820 IF ML=1 THEN 1320
821 GET K$:IF K$="" THEN 821
825 A=ASC(K$)
830 IF A>186 THEN 844
832 IF A=061 THEN NT=7217:RETURN
834 IF A=091 THEN NT=5728:RETURN
836 IF A=093 THEN NT=6430:RETURN
838 IF A=131 THEN NT=2145:RETURN
840 IF A=141 THEN NT=8101:RETURN
842 IF A=186 THEN NT=6069:RETURN
844 IF A>200 THEN 858
846 IF A=192 THEN NT=6812:RETURN
848 IF A=193 THEN NT=2408:RETURN
850 IF A=196 THEN NT=2864:RETURN
852 IF A=198 THEN NT=3215:RETURN
854 IF A=199 THEN NT=3608:RETURN
856 IF A=200 THEN NT=4050:RETURN
858 IF A>209 THEN 872
860 IF A=201 THEN NT=4547:RETURN
862 IF A=202 THEN NT=4291:RETURN
864 IF A=203 THEN NT=4817:RETURN
```

```
866 IF A=204 THEN NT=5407:RETURN
868 IF A=207 THEN NT=5103:RETURN
870 IF A=209 THEN NT=2273:RETURN
872 IF A=210 THEN NT=3034:RETURN
874 IF A=211 THEN NT=2703:RETURN
876 IF A=212 THEN NT=3406:RETURN
878 IF A=215 THEN NT=2551:RETURN
880 IF A=217 THEN NT=3823:RETURN
882 IF A=222 THEN NT=7647:RETURN
890 IF A=160 THEN N8=2/N8:GOTO 820
900 IF A>136 AND A<141 THEN 1210
902 IF A=145 THEN 2120
904 IF A=60 OR A=62 OR A=63 THEN 2220
905 IF A=205 THEN 2360
906 IF A=157 THEN 2320
1090 REM
1091 REM * TURN NOTE OFF *
1092 IF A<>38 AND A<>39 THEN 1098
1094 POKE M+4,WF
1096 GOTO 820
1098 REM * DUMMY BRANCH POINT *
1110 REM
1111 REM * WAVEFORM SELECTION *
1114 IF A=214 OR A=195 THEN 2010
1116 IF A=216 OR A=218 THEN 2010
1119 REM
1120 REM * NOTE SELECTION *
1123 IF A=219 OR A=221 OR A=48 THEN1710
1124 REM
1125 IF A=33 OR A=34 THEN 1520
1128 REM
1129 REM * SWITCH COMPOSE/PLAY *
1130 IF A<>95 THEN 1191
1135 ML=1:A(0,0)=1:Q=0:GOTO 820
1190 REM
1191 GOTO 820       :REM INVALID KEY
1200 REM
1201 REM * ADSR FROM FUNCTION KEYS *
1210 IF A=137 THEN AT=FNE(AT)
1215 IF A=138 THEN DE=FNE(DE)
1220 IF A=139 THEN SU=FNE(SU)
1225 IF A=140 THEN RE=FNE(RE)
1230 AD=AT*16+DE:SR=SU*16+RE
1235 POKE M+5,AD:POKE M+6,SR
1240 PRINT CHR$(19)
1245 FOR J=1 TO 5:PRINT:NEXT
```

```
1247 PRINT "                              "
1248 PRINT "                              "
1249 PRINT CHR$(145);:PRINT CHR$(145);
1250 PRINT "ATTACK= " AT " DECAY=  ";DE
1255 PRINT "SUSTAIN=" SU " RELEASE=";RE
1260 GOTO 820
1300 REM
1301 REM ** PLAY COMPOSITION **
1320 GET K$:IF K$<>"" THEN 1360
1325 TD=A(Q,0)+TI-7:NT=A(Q,1):Q=Q+1
1330 IF NT=0 THEN ML=0:Q=Q-1:GOTO 820
1335 IF TD-TI>150 THEN TD=TI+150
1340 IF TI<TD THEN 1340
1345 RETURN
1359 REM
1360 IF K$=CHR$(95) THEN ML=0
1370 GOTO 820
1500 REM
1501 REM *** DISK/TAPE TRANSFER ***
1520 PRINT CHR$(147) "PRESS SHIFT/LOCK"
1525 INPUT "FILE NAME";F$
1530 IF A=34 THEN 1620   :REM LOAD ?
1535 REM
1536 REM * SAVE *
1540 OPEN 2,8,2,"@0:"+F$+",S,W"
1545 FOR J=0 TO 999:FOR K=0 TO 1
1550 PRINT#2,A(J,K)       :REM WRITE
1555 IF A(J,K)=0 THEN 1565
1560 NEXT K:NEXT J
1565 PRINT#2,CHR$(13)
1570 CLOSE 2:GOTO 5082
1600 REM
1601 REM * LOAD *
1620 OPEN 2,8,2,"@0:"+F$+",S,R"
1625 FOR J=0 TO 999:FOR K=0 TO 1
1630 INPUT#2,A(J,K)        :REM READ
1635 IF A(J,K)=0 THEN 1645
1640 NEXT K:NEXT J
1645 CLOSE 2:GOTO 5082
1700 REM
1701 REM * CHANGE NOTE NO. *
1710 IF A=219 THEN Q=Q+1
1715 IF A=221 THEN Q=Q-1
1720 IF A=48 THEN Q=0
1722 A(Q,0)=40
```

```
1725 PRINT CHR$(19) "                    "
1730 PRINT CHR$(19) "NOTE " Q:GOTO 820
2000 REM
2001 REM * WAVEFORM COMPOSITION *
2010 IF A<>214 THEN 2022
2015 WT=0:WS=0:WP=0:WN=128-WN:T4=1-T4
2017 T1=0:T2=0:T3=0
2020 GOTO 2040
2022 WN=0:T4=0
2025 IF A=218 THEN WT=16-WT:T1=1-T1
2030 IF A=216 THEN WS=32-WS:T2=1-T2
2035 IF A=195 THEN WP=64-WP:T3=1-T3
2040 WF=WT+WS+WP+WN:PRINT CHR$(19)
2045 FOR J=1 TO 8:PRINT:NEXT
2050 PRINT "                                   "
2055 PRINT "                                   "
2060 PRINT CHR$(145) CHR$(145);
2065 PRINT "TIMBRE: LOW  MED. HI NOISE"
2070 PRINT SPC(8) T1"  "T2" "T3"  "T4
2075 GOTO 820
2100 REM
2101 REM * SUB.: SET PULSE WIDTH *
2120 PW=PW+PX:IF PW>4095 THEN PW=100
2125 HP=INT(PW/256):LP=INT(PW-256*HP)
2130 POKE M+2,LP:POKE M+3,HP
2135 PRINT CHR$(19)
2140 FOR J=1 TO 20:PRINT:NEXT
2145 PRINT SPC(17) "    "
2150 PRINT CHR$(145);
2155 PRINT "TIMBRE IS SET AT ";PW
2160 RETURN
2200 REM
2201 REM * FILTERS ON/OFF *
2220 IF A=60 THEN FL=16-FL:T5=1-T5
2230 IF A=62 THEN FM=32-FM:T6=1-T6
2235 IF A=63 THEN FH=64-FH:T7=1-T7
2240 POKE M+24,FL+FM+FH+AM
2242 PRINT CHR$(19)
2245 FOR J=1 TO 16:PRINT:NEXT
2250 PRINT "                                   "
2255 PRINT "                                   "
2260 PRINT CHR$(145) CHR$(145);
2265 PRINT "TONE:   LOW  MED. HI"
2270 PRINT SPC(8) T5"  "T6" "T7
2275 GOTO 820
```

```
2300 REM
2301 REM * SUB: SET FREQUENCY CUTOFF *
2320 CH=CH+15:IF CH>255 THEN CH=0
2325 POKE M+22,CH:PRINT CHR$(19)
2330 FOR J=1 TO 22:PRINT:NEXT
2332 PRINT SPC(18) "    "
2333 PRINT CHR$(145);
2335 PRINT "TONE IS SET AT   ";CH
2340 RETURN
2358 REM
2359 REM * TURN FILTER ON/OFF *
2360 FV=1-FV:POKE M+23,FV
2365 PRINT CHR$(19)
2370 FOR J=1 TO 13:PRINT:NEXT
2375 PRINT "TONE SWITCH IS " FV
2380 GOTO 820
5000 REM
5001 REM **** START PROGRAM ****
5010 REM
5011 REM * INITIALIZE SOUND *
5020 M=54272:AM=15:NB=1:PRINT CHR$(147)
5030 WF=32:PX=100:DE=9
5032 DIM A(999,2)
5035 DEF FN E(X)=X+1-(X+1)*INT(X/15)
5040 FOR J=M TO M+24:POKE J,0:NEXT
5050 POKE M+5,9:POKE M+6,0:POKE M+24,15
5078 REM
5079 REM * START SCREEN AND TITLE *
5080 PRINT:PRINT:PRINT:PRINT
5082 PRINT " *** PRESS SHIFT LOCK ***"
5084 PRINT "  THEN THE J KEY TO START"
5086 GOSUB 820
5088 PRINT CHR$(147)
5090 PRINT SPC(8)"* MUSICAL KEYBOARD *"
5100 REM
5101 REM *** MAIN LINE ***
5120 GOSUB 820
5130 GOSUB 20
5132 IF ML=1 THEN 5120
5135 A(Q,0)=TI-T:T=TI:A(Q,1)=NT:Q=Q+1
5140 GOTO 5120
```

DRAWING BOARD

```
1 REM ***** DRAWING BOARD *****
10 GOTO 5020   :REM BY PASS SUBROUTINES
11 REM
12 REM ** LOCATE A BIT U/L **
20 Z=V*.833
25 IF H<=000 THEN H=000:MH=-MH
30 IF H>=319 THEN H=319:MH=-MH
35 IF Z<=000 THEN Z=000:MV=-MV
40 IF Z>=199 THEN Z=199:MV=-MV
45 CH=INT(H/8)
50 RW=INT(Z/8)
55 LN=Z AND 7
60 BY=BA+RW*320+CH*8+LN
65 BT=7-(H AND 7)
70 IF FE=1 THEN 85
75 POKE BY,PEEK(BY) OR 2^BT
80 POKE 1024+RW*40+CH,CG:RETURN
85 POKE BY,PEEK(BY) AND 255-2^BT:RETURN
800 REM
801 REM ******************************
802 REM *** SUB.: GRAPHICS KEYS ***
820 IF ML=0 THEN 825 :REM SET FOR MOLE?
822 GET K$:IF K$="" THEN RETURN
824 GOTO 826
825 GET K$:IF K$="" THEN 825
826 A=ASC(K$)           :REM ASCII VAL.
832 REM
833 REM * DIRECTION KEYS *
836 IF K$=";" THEN 1020 :REM RT.
838 IF K$="H" THEN 1025 :REM LEFT
840 IF K$="U" THEN 1030 :REM UP
842 IF K$="M" THEN 1035 :REM DN.
844 IF K$="@" THEN 1040 :REM UP./RT.
846 IF K$="N" THEN 1045 :REM DN./LEFT
848 IF K$="Y" THEN 1050 :REM UP/LEFT
850 IF K$="/" THEN 1055 :REM DN./RT.
852 REM
853 REM * CONTROL KEYS *
856 IF K$="+" THEN 1420
858 IF K$="-" THEN 1425
860 IF K$=" " THEN RETURN :REM REPEAT
869 REM
870 REM * MOLE,ANGLES,CURVES,CIRCLES *
```

```
872 IF K$="L" THEN ML=1-ML:GOTO 820
874 IF K$="J" THEN 1220  :REM ROT. CCW
876 IF K$="K" THEN 1230  :REM ROT. CW
878 IF K$="=" THEN 1260  :REM RESTORE
890 IF K$="C" THEN 1320  :REM CIRCLE
970 REM
971 REM * FUNCTION KEYS *
972 REM  COLOR & TURN DRAW ON & OFF
974 IF A=133 THEN 1070      :REM F1
976 IF A=134 THEN 1110      :REM F3
978 IF A=135 THEN 1094      :REM F5
980 IF A=136 THEN DE=1-DE:GOTO 820
982 REM
983 REM * DOT COLOR *
984 IF VAL(K$)<1 OR VAL(K$)>8 THEN 995
986 C=VAL(K$)-1
990 CG=16*C+C1:MH=+0:MV=+0:RETURN
995 GOTO 820       :REM INVALID KEY
1000 REM
1001 REM ** KEY PROCESSING **
1020 MH=+1:MV=+0:RETURN
1025 MH=-1:MV=+0:RETURN
1030 MH=+0:MV=-1:RETURN
1035 MH=+0:MV=+1:RETURN
1040 MH=+1:MV=-1:RETURN
1045 MH=-1:MV=+1:RETURN
1050 MH=-1:MV=-1:RETURN
1055 MH=+1:MV=+1:RETURN
1068 REM
1069 REM * BORDER *
1070 C0=C0+1:IF C0>15 THEN C0=0
1072 POKE 53280,C0:GOTO 820
1090 REM
1091 REM * CHARACTER BACKGROUND *
1094 C1=C1+1:IF C1>7 THEN C1=0
1098 CG=16*C+C1:MH=+0:MV=+0:RETURN
1100 REM
1101 REM * CLEAR AND CYCLE BACKGROUND *
1110 C1=C1+1:IF C1>7 THEN C1=0
1115 CG=16*C+C1
1120 FOR J=1024 TO 2023:POKE J,CG:NEXT
1125 GOTO 820
1200 REM
1201 REM * SET ANGLE *
1220 AN=AN+10:IF AN>360 THEN AN=0
```

```
1225 GOTO 1235
1230 AN=AN-10:IF AN<0 THEN AN=360
1235 DH=COS(AN*AR):MH=+SGN(DH)
1240 DV=SIN(AN*AR):MV=-SGN(DV)
1245 DH=ABS(DH):DV=ABS(DV)
1250 RETURN
1255 REM
1260 DH=1:DV=1:AN=0:GOTO 820
1300 REM
1301 REM * DRAW CIRCLE *
1320 FE=DE                  :REM DELETE?
1325 SH=H:SV=V              :REM SAVE PARMS
1330 RD=RA*6:X=RD:Y=0
1335 FOR Y=0 TO RD/SQR(2)
1340 X=SQR(RD^2-Y^2)
1345 FOR J=1 TO 2
1350 H=SH+X:V=SV+Y:GOSUB 20
1355 H=SH-X:V=SV+Y:GOSUB 20
1360 H=SH+X:V=SV-Y:GOSUB 20
1365 H=SH-X:V=SV-Y:GOSUB 20
1370 T=X:X=Y:Y=T             :REM SWAP X,Y
1375 NEXTJ:NEXT Y
1380 H=SH:V=SV:GOTO 820
1400 REM
1401 REM * INCR/DECR MOVEMENTS *
1419 REM * PLUS *
1420 RA=RA+1:GOTO 820
1424 REM * MINUS *
1425 RA=RA-1:IF RA<1 THEN RA=1
1430 GOTO 820
5000 REM
5001 REM **** START OF PROGRAM ****
5010 REM
5011 REM ** INITIALIZE GRAPHICS **
5020 B=53265:BA=8192:B7=B+7:H=160:V=100
5025 C=0:C1=5:CG=5:DH=1:DV=1:RA=1
5030 AR=3.14159265/180 :REM RADIAN/DEG
5035 PRINT CHR$(147):J$="          "
5040 PRINT J$ "CLEARING DRAWING BOARD"
5045 PRINT J$ "PLEASE WAIT 30 SECONDS"
5050 FOR J=BA TO BA+7999:POKE J,0:NEXT
5055 POKE B7,PEEK(B7) OR 8 :REM BA=8192
5060 POKE B,PEEK (B) OR 32 :REM GRPHCS
5065 FOR J=1024 TO 2023:POKE J,5:NEXT
5070 GOSUB 20                :REM CURSOR
```

```
5100 REM
5101 REM **** MAIN LINE ****
5120 GOSUB 820
5125 IF DE=0 THEN 5140
5130 FE=1:GOSUB 20          :REM DELETE
5135 FE=0
5140 H=H+MH*INT(RA*DH)
5145 V=V+MV*INT(RA*DV)
5150 GOSUB 20               :REM WRITE
5155 GOTO 5120
```

SPRITE ART

```
1 REM ***** SPRITE ART *****
10 GOTO 5020    :REM BY PASS SUBROUTINES
11 REM
12 REM ** LOCATE A SPRITE **
20 IF H<=024 THEN H=024:MH=-MH
30 IF H>=HS  THEN H=HS :MH=-MH
35 IF V<=050 THEN V=050:MV=-MV
40 IF V>=VS  THEN V=VS :MV=-MV
45 HL=H-256*HH
50 REM
55 REM * PROCESS HH BIT SET/RESET *
60 IF HH=0 AND H<256 THEN 105
65 IF HH=1 AND H>255 THEN 105
70 IF HH=1 THEN 90
75 HH=1:HL=0
80 POKE U+16,PEEK(U+16) OR 2^SP
85 GOTO 105
90 HH=0:HL=255
95 POKE U+16,PEEK(U+16) AND 255-2^SP
100 REM
105 POKE U+1+2*SP,V:POKE U+2*SP,HL
110 RETURN
800 REM
801 REM ****************************
802 REM *** SUB.: GRAPHICS KEYS ***
820 IF ML=0 THEN 825 :REM SET FOR MOLE?
822 GET K$:IF K$="" THEN RETURN
824 GOTO 826
825 GET K$:IF K$="" THEN 825
826 A=ASC(K$)               :REM ASCII VAL.
832 REM
```

```
833 REM * DIRECTION KEYS *
836 IF K$=";" THEN 1020 :REM RT.
838 IF K$="H" THEN 1025 :REM LEFT
840 IF K$="U" THEN 1030 :REM UP
842 IF K$="M" THEN 1035 :REM DN.
844 IF K$="@" THEN 1040 :REM UP./RT.
846 IF K$="N" THEN 1045 :REM DN./LEFT
848 IF K$="Y" THEN 1050 :REM UP/LEFT
850 IF K$="/" THEN 1055 :REM DN./RT.
852 REM
853 REM * CONTROL KEYS *
856 IF K$="+" THEN 1420
858 IF K$="-" THEN 1425
860 IF K$=" " THEN RETURN :REM REPEAT
861 IF K$="F" THEN P1=P1+RA:GOTO 1620
862 IF K$="D" THEN P2=P2+RA:GOTO 1620
863 IF K$="S" THEN P3=P3+RA:GOTO 1620
864 IF K$="A" THEN              1610
865 IF K$="Z" THEN 1650
866 IF K$="T" THEN 1710    :REM EXPAND
868 IF K$="G" THEN 1750
869 REM
872 IF K$="L" THEN ML=1-ML:GOTO 820
874 IF K$="J" THEN 1220  :REM ROT. CCW
876 IF K$="K" THEN 1230  :REM ROT. CW
878 IF K$="=" THEN 1260  :REM RESTORE
970 REM
971 REM * FUNCTION KEYS *
972 REM  COLOR & TURN DRAW ON & OFF
974 IF A=133 THEN 1070      :REM F1
976 IF A=134 THEN 1094      :REM F3
978 IF A=135 THEN 1110      :REM F5
980 IF A=136 THEN DE=1-DE:GOTO 820
982 REM
983 REM * SPRITE SELECTION *
984 IF VAL(K$)<1 OR VAL(K$)>8 THEN 995
986 SP=VAL(K$)-1:J=PEEK(U+21) :REM TURN
988 POKE U+21,J OR 2^SP
990 V=PEEK(U+1+2*SP):HL=PEEK(U+2*SP)
992 HH=PEEK(U+16) AND 2^SP:H=HL+256*HH
995 GOTO 820       :REM INVALID KEY
1000 REM
1001 REM ** KEY PROCESSING **
1020 MH=+1:MV=+0:RETURN
1025 MH=-1:MV=+0:RETURN
```

```
1030 MH=+0:MV=-1:RETURN
1035 MH=+0:MV=+1:RETURN
1040 MH=+1:MV=-1:RETURN
1045 MH=-1:MV=+1:RETURN
1050 MH=-1:MV=-1:RETURN
1055 MH=+1:MV=+1:RETURN
1068 REM
1069 REM * BORDER *
1070 C0=C0+1:IF C0>15 THEN C0=0
1072 POKE 53280,C0:GOTO 820
1090 REM
1091 REM * BACKGROUND *
1094 C1=C1+1:IF C1>15 THEN C1=0
1098 POKE 53281,C1:GOTO 820
1100 REM
1101 REM * SPRITE COLOR *
1105 C=PEEK(U+39+SP) AND 15
1110 C=C+1:IF C>15 THEN C=0
1115 POKE U+39+SP,C
1125 GOTO 820
1200 REM
1201 REM * SET ANGLE *
1220 AN=AN+10:IF AN>360 THEN AN=0
1225 GOTO 1235
1230 AN=AN-10:IF AN<0 THEN AN=360
1235 DH=COS(AN*AR):MH=+SGN(DH)
1240 DV=SIN(AN*AR):MV=-SGN(DV)
1245 DH=ABS(DH):DV=ABS(DV)
1250 RETURN
1255 REM
1260 DH=1:DV=1:AN=0:GOTO 820
1400 REM
1401 REM * INCR/DECR MOVEMENTS *
1419 REM * PLUS *
1420 RA=RA+1:GOTO 820
1424 REM * MINUS *
1425 RA=RA-1:IF RA<1 THEN RA=1
1430 GOTO 820
1600 REM
1601 REM ** CHANGE PATTERNS **
1610 FOR J=0 TO 62        :REM CLEAR
1615 POKE LC+64*SP+J,0:NEXT:GOTO 820
1619 REM
1620 IF P1>7 THEN P1=0    :REM CHANGE
1622 IF P3>9 THEN P3=1
```

```
1623 IF 2^P1+P2>255 THEN P2=0
1625 FOR J=0 TO 62 STEP P3
1630 POKE LC+64*SP+J,2^P1+P2
1635 NEXT:GOTO 820
1648 REM
1649 REM * TURN SPRITE OFF & ON *
1650 J=PEEK(U+21)
1655 IF (J AND 2^SP)=0 THEN 1665
1660 POKE U+21,J AND 255-2^SP:GOTO 820
1665 POKE U+21,J OR    2^SP:GOTO 820
1708 REM
1709 REM * EXPAND VERTICALY *
1710 V(SP)=1-V(SP):IF V(SP)=0 THEN 1725
1715 POKE U+23,PEEK(U+23) OR 2^SP
1720 RETURN
1725 POKE U+23,PEEK(U+23) AND 255-2^SP
1730 RETURN
1748 REM
1749 REM * EXPAND HORIZONTALY *
1750 H(SP)=1-H(SP):IF H(SP)=0 THEN 1765
1755 POKE U+29,PEEK(U+29)  OR 2^SP
1760 RETURN
1765 POKE U+29,PEEK(U+29) AND 255-2^SP
1770 RETURN
5000 REM
5001 REM **** START OF PROGRAM ****
5010 REM
5011 REM ** INITIALIZE GRAPHICS **
5020 S0=2040:U=53248  :REM BASE LOC.
5022 H=30:V=55:C=5:LC=12288:RA=1
5023 P1=1:P3=1:DH=1:DV=1
5030 AR=3.14159265/180 :REM RADIAN/DEG
5035 PRINT CHR$(147):J$="            "
5045 FOR J=0 TO 7    :REM POINTERS
5050 POKE U+39+J,C+J:POKE S0+J,192+J
5055 POKE U+1+2*J,V:POKE U+2*J,H+J*30
5060 NEXTJ
5065 POKE U+21,255   :REM SPRITES ON
5068 REM
5069 REM * SET ALL DOTS ON *
5070 FOR J=0 TO 7:FOR K=0 TO 62
5072 POKE LC+J*64+K,255:NEXT K:NEXT J
5100 REM
5101 REM **** MAIN LINE ****
5120 GOSUB 820
```

```
5125 HS=320-H(SP)*24:VS=229-V(SP)*21
5140 H=H+MH*INT(RA*DH)
5145 V=V+MV*INT(RA*DV)
5150 GOSUB 20            :REM WRITE
5155 GOTO 5120
```

SPRITE STUDIO

```
1 REM ***** SPRITE STUDIO *****
10 GOTO 5020   :REM BY PASS SUBROUTINES
11 REM
12 REM ** ONE DOT ON/OFF **
20 IF H<00 THEN H=23:V=V-1
30 IF H>23 THEN H=00:V=V+1
35 IF V>20 THEN V=20
40 IF V<00 THEN V=00
50 CH=INT(H/8)
55 LB=(LC+64*SP)+(V*3+CH)
60 BT=7-(H AND 7)
65 REM
66 REM * IF FE=0, WRITE NEW CURSOR POS.
67 REM IF TB=1 DOT IS A 1;IF 0 THEN 0
70 IF FE=0 THEN 85
75 IF TB=1 THEN 100
80 IF TB=0 THEN 105
85 TB=SGN(PEEK(LB) AND 2^BT)
90 IF TB=1 THEN 105
95 IF TB=0 THEN 100
100 POKE LB,PEEK(LB) OR 2^BT:RETURN
105 POKE LB,PEEK(LB) AND 255-2^BT
110 RETURN
300 REM
301 REM ** AUTO-SYMETRIC PROCESSING **
302 REM SYMETRIES: HORIZONTAL,VERTICAL,
303 REM         DIAGONAL, HORIZ. & VER.
320 IF XS<>1 THEN 330
325 H=24-H:GOSUB 20:H=24-H
330 IF YS<>1 THEN 345
340 V=20-V:GOSUB 20:V=20-V
345 IF DS<>1 THEN 360
350 T=22-H:H=22-V:V=T:GOSUB 20
355 T=22-H:H=22-V:V=T
360 IF AS<>1 THEN 375
```

```
365 H=24-H:V=20-V:GOSUB 20
370 H=24-H:V=20-V
375 RETURN
800 REM
801 REM ******************************
802 REM *** SUB.: GRAPHICS KEYS ***
820 IF ML=0 THEN 825 :REM SET FOR MOLE?
822 GET K$:IF K$="" THEN RETURN
824 GOTO 826
825 GET K$:IF K$="" THEN 825
826 A=ASC(K$)              :REM ASCII VAL.
832 REM
833 REM * DIRECTION KEYS *
836 IF K$=";" THEN 1020 :REM RT.
838 IF K$="H" THEN 1025 :REM LEFT
840 IF K$="U" THEN 1030 :REM UP
842 IF K$="M" THEN 1035 :REM DN.
844 IF K$="@" THEN 1040 :REM UP./RT.
846 IF K$="N" THEN 1045 :REM DN./LEFT
848 IF K$="Y" THEN 1050 :REM UP/LEFT
850 IF K$="/" THEN 1055 :REM DN./RT.
852 REM
853 REM * CONTROL KEYS *
860 IF K$=" " THEN RETURN :REM REPEAT
861 IF K$="F" THEN XS=1-XS:GOTO 820
862 IF K$="D" THEN YS=1-YS:GOTO 820
863 IF K$="A" THEN DS=1-DS:GOTO 820
864 IF K$="S" THEN AS=1-AS:GOTO 820
865 IF K$="Z" THEN 1650
866 IF K$="T" THEN 1710   :REM EXPAND
868 IF K$="G" THEN 1750
869 REM
872 IF K$="L" THEN ML=1-ML:GOTO 820
874 IF K$="J" THEN DE=1:TB=0:RETURN
876 IF K$="K" THEN DE=1:TB=1:RETURN
880 IF K$="X" THEN 1810  :REM REVRS C'S
950 REM
951 REM * DISK SAVE OR LOAD *
955 IF A=211 OR A=204 THEN 1910
970 REM
971 REM * FUNCTION KEYS *
972 REM  COLOR & TURN DRAW ON & OFF
974 IF A=133 THEN 1070     :REM F1
976 IF A=134 THEN 1094     :REM F2
978 IF A=135 THEN 1110     :REM F3
```

```
980 IF A=136 THEN DE=1-DE:GOTO 820
982 REM
983 REM * SPRITE SELECTION *
984 IF VAL(K$)<1 OR VAL(K$)>8 THEN 995
986 SP=VAL(K$)-1:J=PEEK(U+21) :REM TURN
988 POKE U+21,J OR 2^SP:TB=1
990 V=10:H=12:J=LC+64*SP :REM SET CRSR.
992 POKE J+31,PEEK(J+31) AND 247
995 GOTO 820       :REM INVALID KEY
1000 REM
1001 REM ** KEY PROCESSING **
1020 MH=+1:MV=+0:RETURN
1025 MH=-1:MV=+0:RETURN
1030 MH=+0:MV=-1:RETURN
1035 MH=+0:MV=+1:RETURN
1040 MH=+1:MV=-1:RETURN
1045 MH=-1:MV=+1:RETURN
1050 MH=-1:MV=-1:RETURN
1055 MH=+1:MV=+1:RETURN
1068 REM
1069 REM * BORDER *
1070 C0=C0+1:IF C0>15 THEN C0=0
1072 POKE 53280,C0:GOTO 820
1090 REM
1091 REM * BACKGROUND *
1094 C1=C1+1:IF C1>15 THEN C1=0
1098 POKE 53281,C1:GOTO 820
1100 REM
1101 REM * SPRITE COLOR *
1105 C=PEEK(U+39+SP) AND 15
1110 C=C+1:IF C>15 THEN C=0
1115 POKE U+39+SP,C
1125 GOTO 820
1648 REM
1649 REM * TURN SPRITE OFF & ON *
1650 J=PEEK(U+21)
1655 IF (J AND 2^SP)=0 THEN 1665
1660 POKE U+21,J AND 255-2^SP:GOTO 820
1665 POKE U+21,J OR      2^SP:GOTO 820
1708 REM
1709 REM * EXPAND VERTICALY *
1710 V(SP)=1-V(SP):IF V(SP)=1 THEN 1725
1715 POKE U+23,PEEK(U+23) OR 2^SP
1720 GOTO 820
1725 POKE U+23,PEEK(U+23) AND 255-2^SP
```

```
1730 GOTO 820
1748 REM
1749 REM * EXPAND HORIZONTALY *
1750 H(SP)=1-H(SP):IF H(SP)=1 THEN 1765
1755 POKE U+29,PEEK(U+29)  OR 2^SP
1760 GOTO 820
1765 POKE U+29,PEEK(U+29) AND 255-2^SP
1770 GOTO 820
1800 REM
1801 REM * SWTCH BACKGRND. & FORGRND. *
1810 FOR J=0 TO 62
1825 K=PEEK(LC+64*SP+J)
1830 POKE LC+64*SP+J,255-K
1835 NEXT J:TB=1-TB:GOTO  820
1900 REM
1901 REM * MENU FOR SAVE/LOAD *
1910 PRINT CHR$(147)    :REM CLEAR/HOME
1912 PRINT "YOU CAN SAVE OR LOAD 1-8"
1914 PRINT "SPRITES AT A TIME."
1916 INPUT "START SPRITE NO.";SS
1918 IF SS<1 OR SS>8 THEN 1916
1920 INPUT "END SPRITE NO.";ES
1922 IF ES<1 OR ES>8 THEN 1920
1924 IF ES<SS THEN 1920
1926 POKE U+21,0:REM SPRITES MST BE OFF
1928 PRINT "ENTER Q TO QUIT"
1930 INPUT "FILE NAME OR Q";F$
1932 PRINT CHR$(147)
1934 IF F$<>"Q" THEN 1938
1936 POKE U+21,255:GOTO 820
1938 IF A=204 THEN 2010 :REM LOAD?
1940 REM
1941 REM * SAVE SPRITE *
1945 OPEN 2,8,2,"@0:"+F$+",S,W"
1950 FOR SP=SS-1 TO ES-1
1955 FOR J=0 TO 62
1960 PRINT CHR$(147)"SPRITE IS    "SP+1
1965 PRINT "CHARACTER IS "J+1
1970 S$=STR$(PEEK(LC+64*SP+J))
1975 PRINT#2,S$     :REM WRITE CHAR.
1980 NEXT J:NEXT SP
1985 PRINT#2,CHR$(13)
1990 CLOSE 2:SP=SP-1:POKE U+21,255
1995 GOTO 820
2000 REM
```

```
2001 REM * LOAD SPRITE *
2010 OPEN 2,8,2,"@0:"+F$+",S,R"
2015 FOR SP=SS-1 TO ES-1
2020 FOR J=0 TO 62
2025 PRINT CHR$(147)"SPRITE IS    "SP+1
2030 PRINT "CHARACTER IS "J+1
2035 INPUT#2,S$:SV=VAL(S$)
2040 POKE LC+64*SP+J,SV
2045 NEXT J:NEXT SP
2050 CLOSE 2:SP=SP-1:POKE U+21,255
2055 GOTO 820
5000 REM
5001 REM **** START OF PROGRAM ****
5010 REM
5011 REM ** INITIALIZE SPRITES **
5020 S0=2040:U=53248  :REM BASE LOC.
5022 H=48:V=125:C=1:LC=12288:HL=H
5035 PRINT CHR$(147):J$="         "
5037 PRINT J$ " INITIALIZING SPRITES"
5040 PRINT J$ "PLEASE WAIT 10 SECONDS"
5042 REM
5045 FOR J=0 TO 7     :REM POINTERS
5050 POKE U+39+J,C:POKE S0+J,192+J
5055 POKE U+1+2*J,V:POKE U+2*J,HL
5060 H=H+73:IF H<278 THEN 5067
5065 V=V+64:H=48:GOTO 5068
5067 IF H>255 THEN HL=H-256:GOTO 5070
5068 HL=H
5070 FOR K=0 TO 62
5072 POKE LC+J*64+K,255:NEXT K:NEXT J
5074 POKE U+16,136
5079 REM
5080 REM ** ENTRY POINT FOR RESTART **
5082 PRINT CHR$(147) SPC(8);
5084 PRINT "**** SPRITE STUDIO ****"
5086 U=53248:LC=12288:C=1
5088 POKE U+23,255:POKE U+29,255
5090 POKE U+21,255:POKE LC+31,247
5095 TB=1:H=12:V=10  :REM NEW H AND V
5100 REM
5101 REM **** MAIN LINE ****
5102 REM IF FE=1 AND DE=0, ZAP CURSOR
5103 REM IF FE=1 AND DE=1, DRAW A DOT
5104 REM IF FE=0, WRITE NEW CURSR. POS.
5120 GOSUB 820
```

```
5125 FE=1:GOSUB 20  :REM ZAP/DRAW
5130 IF DE=1 THEN DE=0:GOTO 5170
5135 FE=0
5140 H=H+MH
5145 V=V+MV
5150 GOSUB 20          :REM WRITE CURSOR
5155 GOTO 5120
5170 GOSUB 320    :REM AUTO-SYMETRIC
5175 GOTO 5120
```

SPRITE THEATER

```
1 REM ***** SPRITE THEATER *****
10 GOTO 5020    :REM BY PASS SUBROUTINES
11 REM
12 REM ** LOCATE A SPRITE **
20 IF H<=024 THEN H=024:MH=-MH
30 IF H>=HS  THEN H=HS :MH=-MH
35 IF V<=050 THEN V=050:MV=-MV
40 IF V>=VS  THEN V=VS :MV=-MV
45 HL=H-256*HH
50 REM
55 REM * PROCESS HH BIT SET/RESET *
60 IF HH=0 AND H<256 THEN 105
65 IF HH=1 AND H>255 THEN 105
70 IF HH=1 THEN 90
75 HH=1:HL=0
80 POKE U+16,PEEK(U+16) OR 2^SP
85 GOTO 105
90 HH=0:HL=255
95 POKE U+16,PEEK(U+16) AND 255-2^SP
100 REM
105 POKE U+1+2*SP,V:POKE U+2*SP,HL
110 RETURN
400 REM
401 REM ** COLISION DETECTION **
402 REM SPRITE HITS SPRITE,HS
420 IF WF=0 THEN 495 :REM SOUND OFF ?
425 SH=PEEK(U+30):J=SH AND 2^SP
430 IF J=0 THEN BC=0:GOTO 495
432 IF BC=1 THEN 495
433 BC=1:MV=-MV
435 SH=SH-2^SP :REM REMOVE SEL. SP.
```

```
440 REM
445 REM * COMPUTE NOTE BASED ON SPRITE*
450 SJ=0:FOR J=0 TO 7
455 SJ=SJ+SGN(SH AND 2^J)*J:NEXT J
460 IF SJ>255 THEN SJ=255
470 POKE M+1,(1+SJ)*N8  :REM HI VAL.
475 REM
480 REM * TRN ON, DELAY, TRN OFF, DELAY
485 POKE M+4,WF  :FOR J=1 TO 250:NEXT
490 POKE M+4,WF-1:FOR J=1 TO 002:NEXT
495 RETURN
800 REM
801 REM *****************************
802 REM *** SUB.: GRAPHICS KEYS ***
820 IF ML=0 THEN 825 :REM SET FOR MOLE?
822 GET K$:IF K$="" THEN RETURN
824 GOTO 826
825 GET K$:IF K$="" THEN 825
826 A=ASC(K$)               :REM ASCII VAL.
832 REM
833 REM * DIRECTION KEYS *
836 IF K$=";" THEN 1020 :REM RT.
838 IF K$="H" THEN 1025 :REM LEFT
840 IF K$="U" THEN 1030 :REM UP
842 IF K$="M" THEN 1035 :REM DN.
844 IF K$="@" THEN 1040 :REM UP./RT.
846 IF K$="N" THEN 1045 :REM DN./LEFT
848 IF K$="Y" THEN 1050 :REM UP/LEFT
850 IF K$="/" THEN 1055 :REM DN./RT.
852 REM
853 REM * CONTROL KEYS *
856 IF K$="+" THEN 1420
858 IF K$="-" THEN 1425
860 IF K$=" " THEN RETURN :REM REPEAT
861 IF K$="F" THEN P1=P1+RA:GOTO 1620
862 IF K$="D" THEN P2=P2+RA:GOTO 1620
863 IF K$="S" THEN P3=P3+RA:GOTO 1620
864 IF K$="A" THEN               1610
865 IF K$="Z" THEN 1650
866 IF K$="T" THEN 1710   :REM EXPAND
868 IF K$="G" THEN 1750
869 REM
872 IF K$="L" THEN ML=1-ML:GOTO 820
874 IF K$="J" THEN 1220  :REM ROT. CCW
876 IF K$="K" THEN 1230  :REM ROT. CW
```

```
878 IF K$="=" THEN 1260  :REM RESTORE
880 IF K$="X" THEN 1810  :REM REVRS C'S
882 IF K$="W" THEN 3020
884 IF K$="O" THEN 3060
950 REM
951 REM * DISK SAVE OR LOAD *
955 IF A=211 OR A=204 THEN 1910
970 REM
971 REM * FUNCTION KEYS *
972 REM  COLOR & TURN DRAW ON & OFF
974 IF A=133 THEN 1070      :REM F1
976 IF A=134 THEN 1094      :REM F3
978 IF A=135 THEN 1110      :REM F5
980 IF A=136 THEN DE=1-DE:GOTO 820
982 REM
983 REM * SPRITE SELECTION *
984 IF VAL(K$)<1 OR VAL(K$)>8 THEN 995
986 SP=VAL(K$)-1:J=PEEK(U+21) :REM TURN
988 POKE U+21,J OR 2^SP
990 V=PEEK(U+1+2*SP):HL=PEEK(U+2*SP)
992 HH=PEEK(U+16) AND 2^SP:H=HL+256*HH
995 GOTO 820        :REM INVALID KEY
1000 REM
1001 REM ** KEY PROCESSING **
1020 MH=+1:MV=+0:RETURN
1025 MH=-1:MV=+0:RETURN
1030 MH=+0:MV=-1:RETURN
1035 MH=+0:MV=+1:RETURN
1040 MH=+1:MV=-1:RETURN
1045 MH=-1:MV=+1:RETURN
1050 MH=-1:MV=-1:RETURN
1055 MH=+1:MV=+1:RETURN
1068 REM
1069 REM * BORDER *
1070 C0=C0+1:IF C0>15 THEN C0=0
1072 POKE 53280,C0:GOTO 820
1090 REM
1091 REM * BACKGROUND *
1094 C1=C1+1:IF C1>15 THEN C1=0
1098 POKE 53281,C1:GOTO 820
1100 REM
1101 REM * SPRITE COLOR *
1105 C=PEEK(U+39+SP) AND 15
1110 C=C+1:IF C>15 THEN C=0
1115 POKE U+39+SP,C
```

```
1125 GOTO 820
1200 REM
1201 REM * SET ANGLE *
1220 AN=AN+10:IF AN>360 THEN AN=0
1225 GOTO 1235
1230 AN=AN-10:IF AN<0 THEN AN=360
1235 DH=COS(AN*AR):MH=+SGN(DH)
1240 DV=SIN(AN*AR):MV=-SGN(DV)
1245 DH=ABS(DH):DV=ABS(DV)
1250 RETURN
1255 REM
1260 DH=1:DV=1:AN=0:GOTO 820
1400 REM
1401 REM * INCR/DECR MOVEMENTS *
1419 REM * PLUS *
1420 RA=RA+1:GOTO 820
1424 REM * MINUS *
1425 RA=RA-1:IF RA<1 THEN RA=1
1430 GOTO 820
1600 REM
1601 REM ** CHANGE PATTERNS **
1610 FOR J=0 TO 62        :REM CLEAR
1615 POKE LC+64*SP+J,0:NEXT:GOTO 820
1619 REM
1620 IF P1>7 THEN P1=0   :REM CHANGE
1622 IF P3>9 THEN P3=1
1623 IF 2^P1+P2>255 THEN P2=0
1625 FOR J=0 TO 62 STEP P3
1630 POKE LC+64*SP+J,2^P1+P2
1635 NEXT:GOTO 820
1648 REM
1649 REM * TURN SPRITE OFF & ON *
1650 J=PEEK(U+21)
1655 IF (J AND 2^SP)=0 THEN 1665
1660 POKE U+21,J AND 255-2^SP:GOTO 820
1665 POKE U+21,J OR      2^SP:GOTO 820
1708 REM
1709 REM * EXPAND VERTICALY *
1710 V(SP)=1-V(SP):IF V(SP)=0 THEN 1725
1715 POKE U+23,PEEK(U+23) OR 2^SP
1720 RETURN
1725 POKE U+23,PEEK(U+23) AND 255-2^SP
1730 RETURN
1748 REM
1749 REM * EXPAND HORIZONTALY *
```

```
1750 H(SP)=1-H(SP):IF H(SP)=0 THEN 1765
1755 POKE U+29,PEEK(U+29)  OR 2^SP
1760 RETURN
1765 POKE U+29,PEEK(U+29) AND 255-2^SP
1770 RETURN
1800 REM
1801 REM * SWTCH BACKGRND. & FORGRND. *
1810 FOR J=0 TO 62
1825 K=PEEK(LC+64*SP+J)
1830 POKE LC+64*SP+J,255-K
1835 NEXT J:TB=1-TB:GOTO  820
1900 REM
1901 REM * MENU FOR SAVE/LOAD *
1910 PRINT CHR$(147)    :REM CLEAR/HOME
1912 PRINT "YOU CAN SAVE OR LOAD 1-8"
1914 PRINT "SPRITES AT A TIME."
1916 INPUT "START SPRITE NO.";SS
1918 IF SS<1 OR SS>8 THEN 1916
1920 INPUT "END SPRITE NO.";ES
1922 IF ES<1 OR ES>8 THEN 1920
1924 IF ES<SS THEN 1920
1926 POKE U+21,0:REM SPRITES MST BE OFF
1928 PRINT "ENTER Q TO QUIT"
1930 INPUT "FILE NAME OR Q";F$
1932 PRINT CHR$(147)
1934 IF F$<>"Q" THEN 1938
1936 POKE U+21,255:GOTO 820
1938 IF A=204 THEN 2010 :REM LOAD?
2000 REM
2001 REM * LOAD SPRITE *
2010 OPEN 2,8,2,"@0:"+F$+",S,R"
2015 FOR SP=SS-1 TO ES-1
2020 FOR J=0 TO 62
2025 PRINT CHR$(147)"SPRITE IS    "SP+1
2030 PRINT "CHARACTER IS "J+1
2035 INPUT#2,S$:SV=VAL(S$)
2040 POKE LC+64*SP+J,SV
2045 NEXT J:NEXT SP
2050 CLOSE 2:SP=SP-1:POKE U+21,255
2052 PRINT CHR$(147)     :REM CLEAR
2055 GOTO 820
3000 REM
3001 REM * WAVEFORM CONTROL *
3020 SU=SU+1:IF SU>3 THEN SU=1
3025 IF SU=1 THEN WF=33
```

```
3030 IF SU=2 THEN WF=0
3035 IF SU=3 THEN WF=129
3040 RETURN
3050 REM
3055 REM * OCTAVE CONTROL *
3060 N8=N8+1:IF N8>7 THEN N8=1
3065 RETURN
5000 REM
5001 REM **** START OF PROGRAM ****
5010 REM
5011 REM ** INITIALIZE GRAPHICS **
5020 S0=2040:U=53248  :REM BASE LOC.
5022 H=30:V=225:C=5:LC=12288:RA=1
5023 P1=1:P3=1:DH=1:DV=1
5030 AR=3.14159265/180 :REM RADIAN/DEG
5035 PRINT CHR$(147):J$="         "
5045 FOR J=0 TO 7    :REM POINTERS
5050 POKE U+39+J,C+J:POKE S0+J,192+J
5055 POKE U+1+2*J,V:POKE U+2*J,H+J*30
5060 NEXTJ
5065 POKE U+21,255
5070 POKE U+30,0
5080 REM
5081 REM ** INITIALIZE SOUND **
5084 M=54272:FOR J=M TO M+24
5086 POKE J,0:NEXT  :REM INIT. CHIP
5088 POKE M+5,060:POKE M+6,060
5090 POKE M+24,15
5095 POKE M+1,017:POKE M,037
5098 HF=16:N8=1:WF=129
5100 REM
5101 REM **** MAIN LINE ****
5120 GOSUB 820
5125 HS=320-H(SP)*24:VS=229-V(SP)*21
5140 H=H+MH*INT(RA*DH)
5145 V=V+MV*INT(RA*DV)
5150 GOSUB 20               :REM WRITE
5152 GOSUB 420
5155 GOTO 5120
```

Computer Books from PLUME

(0452)

☐ **MASTERING SIGHT AND SOUND ON THE COMMODORE® 64™ by Kent Porter.** This reader-friendly book tells you everything you need to know to get the most out of your Commodore 64 home computer including: an overview of the computer's full range of powers; the full spectrum of visual potential available; the secrets of synthesizing sounds for special effects or as musical compositions. Complete with computer codes, program illustrations, and a host of other uniquely helpful features. (254906—$9.95)

☐ **BEGINNING WITH BASIC: AN INTRODUCTION TO COMPUTER PROGRAMMING by Kent Porter.** Now, at last, the new computer owner has a book that speaks in down-to-earth everyday language to explain clearly—and step-by-step—how to master BASIC, Beginner's All-Purpose Symbolic Instructional Code, and how to use it to program your computer to do exactly what you want it to do. (254914—$10.95)

☐ **DATABASE PRIMER: AN EASY-TO-UNDERSTAND GUIDE TO DATABASE MANAGEMENT SYSTEMS by Rose Deakin.** The future of information control is in database management systems—tools that help you organize and manipulate information or data. This essential guide tells you how a database works, what it can do for you, and what you should know when you go to buy one. (254922—$9.95)†

☐ **THE COMPUTER PHONE BOOK™ by Mike Cane.** The indispensable guide to personal computer networking. A complete annotated listing of names and numbers so you can go online with over 400 systems across the country. Includes information on: free software; electronic mail; computer games; consumer catalogs; medical data; stock market reports; dating services; and much, much more. (254469—$9.95)

All prices higher in Canada.

†Not available in Canada.

To order, use the convenient coupon on the next page.

Titles of Related Interest from PLUME and MERIDIAN

(0452)

☐ **THE COMPUTER FREELANCER'S HANDBOOK: Moonlighting With Your Home Computer by Ardy Friedberg.** This practical guide will show you how you can use your personal computer for extra income. Step-by-step advice, a wealth of real-life success stories, and inspiring ideas offer all the information you'll need for choosing the right home-based business, figuring prices, attracting customers, and growing as much and as fast as you want. (255627—$10.95)

☐ **ALMOST FREE COMPUTER STUFF FOR KIDS by Linda Gail Christie and Gary Bullard.** Hundreds of companies across the country offer a tremendous array of products for computer fun and educational challenge at startlingly low prices—or even no cost—if you know where to write. This book tells you all the things you can get and provides the send-away-for coupons you need to enjoy special discounts on everything from software to T-shirts. (255619—$9.95)

☐ **WEBSTER'S NEW WORLD DICTIONARY OF THE AMERICAN LANGUAGE, 100,000 Entry Edition.** Includes 100,000 vocabulary entries as well as more than 600 illustrations. (006198—$8.50)

☐ **WEBSTER'S NEW WORLD THESAURUS by Charlton Laird.** A master key to the resources and complexities of twentieth-century American English, containing 30,000 major entries with synonyms listed by frequency of use. (006279—$8.95)

All prices higher in Canada.

Buy them at your local bookstore or use this convenient coupon for ordering.

NEW AMERICAN LIBRARY
P.O. Box 999, Bergenfield, New Jersey 07621

Please send me the PLUME and MERIDIAN BOOKS I have checked above. I am enclosing $__________(please add $1.50 to this order to cover postage and handling). Send check or money order—no cash or C.O.D.'s. Prices and numbers are subject to change without notice.

Name____________________________

Address__________________________

City____________State________Zip Code________

Allow 4-6 weeks for delivery
This offer subject to withdrawal without notice.

COMPUTERS • Z5571 • $12.95
CANADA • $16.25

RECIPES FOR COMPUTER CREATIVITY

Your Commodore 64 is as good as the programs you punch into it —and probably better than you ever imagined. For this amazingly powerful personal computer is capable of an extraordinary variety of functions to entertain you, enrich you, and expand your computer creativity.

Now one book plugs both beginners and advanced users alike into the many marvels of the Commodore 64. Here are the simplest ways to produce graphics, music and animation that delight the eye, the ear, and the intellect. Here, too, is how to let your Commodore 64 reach out to other computers and computer users all over the land for a fruitful exchange of ideas and information, and for the plain fun of getting to know one another.

You've bought your Commodore 64. Now's the time to find out what a really great bargain it is!

COOKBOOK OF CREATIVE PROGRAMS FOR THE COMMODORE® 64

PROJECTS FOR MUSIC, ANIMATION AND TELECOMMUNICATIONS

ISBN 0-452-25571-6