

AN INTRODUCTION TO THE
commodore

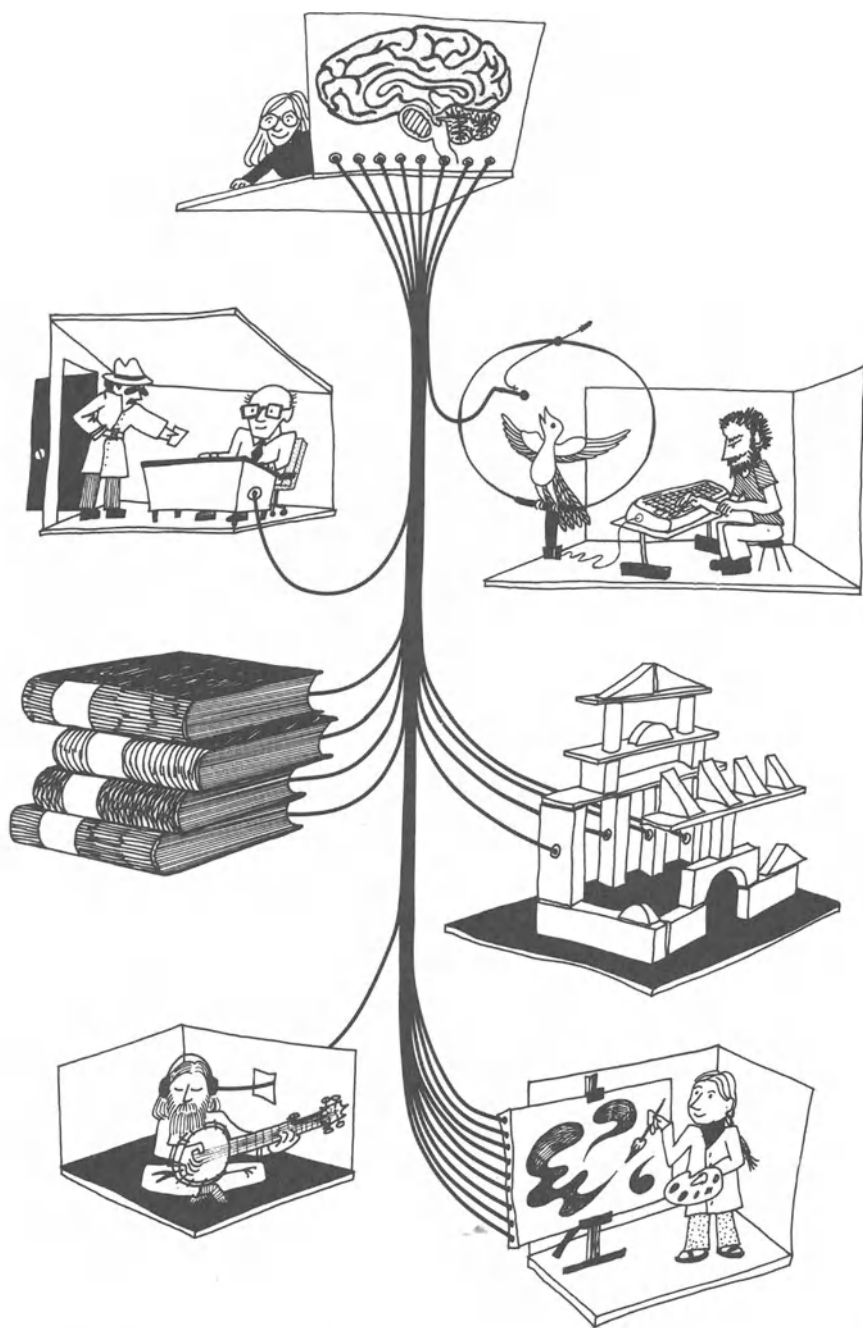


NEVIN B. SCRIMSHAW

JAMES VOGEL

An Introduction to the Commodore 64

Adventures in Programming



Scrantz_5.29.83

An Introduction to the Commodore 64

Adventures in Programming

Nevin B. Scrimshaw
and
James Vogel

Springer Science+Business Media, LLC

Library of Congress Cataloging in Publication Data

Scrimshaw, Nevin, 1950-
An introduction to the Commodore 64.

Bibliography: p.
Includes index.

1. Commodore 64 (Computer) – Programming. I. Vogel,
James, 1952- . II. Title.
QA76.8.C64S37 1983 001.64 '2 83-15650

ISBN 978-1-4899-6789-3 ISBN 978-1-4899-6787-9 (eBook)
DOI 10.1007/978-1-4899-6787-9

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission of the copyright owner.

A B C D E F G H I J

© Springer Science+Business Media New York 1983
Originally published by Birkhäuser Boston, Inc., in 1983

Contents

| | |
|--|-----------|
| Preface | |
| 1 First Things First | 1 |
| Tips on the quirks and strong points of your Commodore 64. A journey through the keyboard is included. | |
| 2 BASIC Number Crunching | 8 |
| Learning to keep track of all those numbers! | |
| 3 A BASIC Sampler | 14 |
| How to talk in a way that the computer understands. A closer look at what BASIC is and what it has to offer. | |
| 4 Program Editing | 19 |
| Using the first measure of Beethoven's Fifth Symphony to explain the ins and outs of editing and entering programs—a chapter on debugging in disguise. | |
| 5 The Case of the Cursory Cursor | 26 |
| A chance to practice your newfound editing skills with a game that exercises your cursor control. | |
| 6 Loops in Loops in Loops... | 29 |
| The use of nested loops is examined. Loop technology is used to explore probability via simulations. | |
| 7 Graphics | 35 |
| Exploring the fourth dimension—a fish tank for randomly swimming square fish made out of sprites. | |
| 8 Program Design | 42 |
| An in-depth charge at subroutines. | |
| 9 Number Theory I | 48 |
| Put the computer to work doing what it does best. | |
| 10 Sound and Music | 52 |
| How to write sound programs for music and special sound effects. Includes Nick's favorite. | |
| 11 Real-Time Graphics | 71 |
| The first steps toward a program that will use sprites to fly a spaceship around your screen. An exposé of the sprite switchboard. | |
| 12 Microsurgery | 77 |
| An introduction to the use of string variables. The odd hexmas tree is examined as an application. | |
| 13 Number Theory II | 82 |
| Exploring mathematical oddities. How long does it take a drunken graphics bug to totter off the screen? | |

| | | |
|-----------|--|-----|
| 14 | Launching a Sprite | 89 |
| | Build your fleet, sail in your home computer waters. Will you fall off the edge? Complete with fog horn. | |
| 15 | Tricks Of The SID | 98 |
| | Variations of a simple music program provide an amazing range of effects. | |
| 16 | Out of This World Graphics. | 105 |
| | The rocket takes off amidst a thundering roar and disappears into hyperspace. | |
| | Appendices and Charts | 111 |
| | Index | 123 |

Preface

Your Commodore 64 is a powerful microcomputer whose memory, color graphics, and sound effects far exceed those of other computers in its class. At first, learning to control these features may seem difficult and complex, but don't be intimidated; there's nothing mysterious about your machine—it is, after all, only a machine. To communicate with it you simply need to use its language: Commodore 64 BASIC (Beginner's All-purpose Symbolic Instruction Code), a version of the most popular language used on today's microcomputers. This book will introduce you to Commodore 64 BASIC and to the fun you can have with your computer.

Learning a new language takes time, effort, and practice—especially practice. So when you first set up your Commodore 64, take time to read the first few chapters of the *User's Guide*, which this book supplements but does not replace. Then you'll be ready to practice your new language using the programs and skills developed in this book. In addition, there is a rather thick book put out by Commodore called the *Programmer's Reference Guide*. It contains enough information about your 64 to occupy a long New England winter.

The first three chapters introduce some fundamental principles of microcomputers, review the Commodore 64 and its quirks, and get you into the basics of BASIC. The next two chapters use music and a game to explore the 64's editing features and cursor controls. Chapter 6 demonstrates some numerical acrobatics, and Chapters 7 and 11 introduce graphics and the notion of binary arithmetic. In Chapter 8 we examine the planning behind writing long programs. Chapters 9 and 13 introduce some number theory in disguise, and Chapters 10 and 15 sample the full range of music-making with your Commodore. In Chapters 14 and

16 we pull the numbers, sound, and graphics together to create a screen full of sound and light. Our goal is to present some critical programming concepts while giving you a collection of programs to practice and play with.

One note: the programs in this book use the Commodore's full range of sound and color. If you don't have a color TV, you can still run the programs, but the results will be less spectacular. If you do have a color TV, you might want to turn off the color while you are keying or editing in programs to ease the eye strain sometimes associated with using video terminals. If you wish to use the programs as models for future efforts of your own, you will need an external storage device, either a disk drive or DATASSETTE™ recorder. Refer to Chapter 2 in the *User's Guide* for more information.

With all the talk of "computer illiteracy" it is easy to lose track of what you really want computers to do for you—expand your sense of life. This can't be done for you by some computer expert. You, at some point, must do it for yourself. When you get to the song called "Chariot" in Chapter 11, shut your eyes, lean back and listen: you'll hear one reward of having spent time learning about your computer.

Acknowledgements

The able assistance of the following people is deeply appreciated:

David Epstein
Virginia Michie
Cort Shurtleff
Roy Piascik

Special thanks to the students of Northeastern University.

N. Scrantz Lersch did all of the illustrations for this book. She is a freelance illustrator living in Worcester, Massachusetts with her husband and two children. She also works as a photographer in the Anatomy Department at the University of Massachusetts Medical School.

This book was produced on an IBM PC and telecommunicated and typeset by Arts & Letters, Brookline, Massachusetts.

An Introduction to the Commodore 64

Adventures in Programming

1 First Things First

Like all computers, your Commodore 64 processes information: it receives information (or input), does something with it, and sends it out again as output. A single chip of silicon with thousands of electronic components—in this case called the 6510 microprocessor—endows your microcomputer with its computing power; this tiny chip is the brain of the system.

The 64 has two other chips that are almost computers in themselves. The VIC II chip handles graphics and the SID chip is a complete sound synthesizer with three separate voices.

But for all its power, your Commodore 64 is something of a brute: you must tell it what to do and how to do it. You must feed in not only the information you want processed (data), but the instructions for processing it (programs). Collectively, programs are called software, and without them your computer would be like a stereo system without records or tapes—useless. A computer program is simply a list of instructions written in a programming language (e.g., BASIC, FORTRAN, or COBOL). The microprocessor “reads” the instructions and carries them out. Although computers process hundreds or thousands of instructions in less than a second, the microprocessor actually performs one instruction—one program step—at a time. Meanwhile, your data and other program steps are electronically stored on more tiny silicon chips that make up the computer’s internal memory.

THE KEYBOARD

Your primary means of communicating with the Commodore 64 is the keyboard, which is similar in many ways to that of a standard

typewriter. But you'll notice that the Commodore keyboard has several special keys; these keys make it much more versatile than a typewriter's.

Modes

Set up your 64 as directed in the *User's Guide* and flip up the rocker switch on the keyboard's right side. When you first turn the computer on, you're in Standard, or upper case/graphic, Mode. That blinking square under READY is the cursor; it marks the spot where what you type next will appear on the screen.

Look for the two cursor control keys on the lower right of the keyboard. In Standard Mode, pressing these keys moves the cursor down or to the right. Pressing the [SHIFT] key and a [CRSR] key at the same time moves the cursor up or to the left. In this book, pressing two keys together will be indicated with a colon, e.g., [SHIFT]:[RCSR]. Our notation for the right cursor is [RCSR]; the left cursor is [LCRSR], otherwise we shall simply use [CRSR] to ambiguously refer to either cursor. Go ahead and experiment to get a feel for using the cursor controls.

Perhaps the most important key on the keyboard is the [RETURN] key. Like its counterpart on a typewriter, [RETURN] brings you (the cursor actually) back to the left margin, one line down; if the cursor is at the bottom of your screen, pressing [RETURN] scrolls up the display one line. Try it!

But [RETURN] on a computer has a more important function: it signals the computer that you have finished typing instructions and enters those instructions into memory. Failing to press [RETURN] at the end of any command is like not talking to the computer at all; pressing [RETURN] means that you expect the computer to respond.

Now, starting with the letter A, type in all the characters in the third row from the top of your keyboard (but don't push [RETURN]). You'll see that in standard mode the Commodore prints on the screen either the character on the top face of the key or the lower character when there are two; all the letters are upper-case.

Now push [RETURN]. Your screen should look like this:

```
ASDFGHJKL:; =  
?SYNTAX ERROR  
READY.
```

By hitting [RETURN], you entered an instruction that makes no sense in the BASIC language your computer understands; the way you phrased

your instruction—your syntax—was faulty, and your Commodore responded with an ERROR message. (For a list of ERROR messages and what they mean, see Appendix L, page 150, in your *User's Guide*).

Now, from left to right, press all the keys in the top row. What happened? When you hit the [CLR/HOME] key the cursor shot back up to the left corner of your screen. That position is called home; the computer followed the instruction printed on the lower half of the [CLR/HOME] key, sending the cursor home. Use the left [LCRSR] key to bring the cursor back one line below the line you just typed.

On a typewriter [SHIFT]ing gives you a completely different set of characters. On the Commodore [SHIFT]ing does the same thing, but what characters or functions you get depends on what mode you're in. In Standard Mode holding down the [SHIFT] key while pressing any other key will print the graphics symbol on the right front face of that key. For keys with two characters on the top (the number keys, for instance), [SHIFT]ing gives you the top character or function. Pushing a [SHIFT]ed [CLR/HOME] key thus not only takes the cursor home, but clears the screen as well.

Type in some letters and symbols and find the [COMMODORE] key on the keyboard's lower left corner. Press this and the [SHIFT] key at the same time.

Welcome to the lower case! In this second mode the Commodore keyboard behaves essentially like a typewriter: the [SHIFT] key produces upper-case letters and symbols, but no graphics. Experiment for a while. To get back to Standard Mode, push both [COMMODORE] and [SHIFT] again.

Quote Mode

A third mode is Quote Mode, and you get in and out of it from either Standard or Typewriter Mode by typing quotation marks: one set gets you in, a second set gets you out. Quote Mode can be disconcerting at first because keys like [CLR/HOME], [INST/DEL], and the [CRSR] keys don't do what they did in the other modes; instead they print graphic symbols. [DEL] is the only function that works normally. For example, if you hold down the [SHIFT] key and type "[CLR/HOME]" you should see a deep blue heart on a light blue square. I guess your Commodore thinks that home is where the heart is.

Quote Mode is used with the BASIC command PRINT to display something in a precise format. Try this: first clear the screen by typing [SHIFT]:[CLR/HOME].

Now type PRINT and quotation marks followed by [SHIFT]: [CLR/HOME] and A LITTLE SONG. The line should look like this so far:

PRINT " A LITTLE SONG

Now press the left [LCRSR] key three times and type AND DANCE" so that the line looks (except on your screen each "Q" is in inverse video) like this:

PRINT " A LITTLE SONGQQQAND DANCE"

To make the 64 respond to your command, press [RETURN]. With any luck you should have:

A LITTLE SONG

AND DANCE

In Quote Mode, pressing the [CRSR] or [CLR/HOME] keys elicits a kind of delayed response from the computer. You don't want the cursor moved or the screen cleared now; rather, you want those actions incorporated into your display. The graphic symbols are placeholders representing what you want done. The computer won't print them when it "reads" them—one at a time—as part of your instruction; it performs the indicated action instead.

Your Commodore 64 has still another mode: Reverse Field Mode. The heart that appears when you type [SHIFT]:[CLR/HOME] is in a reverse field. To make everything appear on a reverse field, hold down the [CTRL] key (second row left) while pressing [9]. Type in some letters and graphic symbols and watch the effect. What happens when you hold down the space bar? To get back to a normal field, type [CTRL]:[0].

SPECIAL KEYS

[RUN/STOP] and [RESTORE]

By now, you've probably filled the screen with symbols and are wondering how to get back to normal. Hold down [RUN/STOP] (third row left) and press [RESTORE] (second row right). This combination returns you to the original blue screen with light blue border and lettering; nothing stored in memory is altered or lost. When you start experimenting with complex programs, this combination will be a lifesaver.

[RUN/STOP] is also used with BASIC commands and to stop the

execution of most BASIC programs. For example, typing LIST followed by [RETURN] displays any program stored in memory. If you decide you don't want a LISTing after all, hitting [RUN/STOP] will break it off. [RUN/STOP] also stops a tape loading sequence.

[CTRL] and [COMMODORE]

Whatever mode you're in, two keys, [CTRL] and [COMMODORE], work something like the [SHIFT] key, giving extra characters or performing special functions when pressed with other keys. Using [COMMODORE], [CTRL], or [SHIFT] with other keys while in Quote Mode allows you to PRINT an amazing array of patterns in almost any color. We'll see such combinations in action in later chapters.

On the Commodore 64, [CTRL] is used primarily with the number keys [1]-[8] to set colors (see pages 11-12 in the *User's Guide*); when used with numbers [9] and [0], it turns the reverse field on and off. It can also produce some special effects when used in Quote Mode. Try typing:

PRINT “

Now press simultaneously the [CONTROL] and the [5] keys. (Our notation for pressing two keys together is to separate the brackets with a colon e.g. [CONTROL]:[5].)

You should see a reverse field graphic symbol between your quotation marks. Now hit [RETURN]. The text color changes. To get back



ROMulus and RAMus

to the light blue start-up screen push [RUN/STOP]:[RESTORE].

You can also press [CTRL] to slow a BASIC program LISTing to scroll at a readable pace.

Used with letters in Standard Mode, the [COMMODORE] key produces the left front graphic symbol. Used with the color keys, it generates eight new colors. And when combined with the [SHIFT] key, it takes you from standard upper case/graphic mode to upper/lower case mode.

The [COMMODORE] key can also speed up loading time of tape cassettes. Normally, several seconds elapse between the time you get a FOUND statement and a LOADING statement (see pages 19-20 in the *User's Guide*). You can load without waiting by pressing the [COMMODORE] key after the FOUND statement flashes.

[INST/DEL]

In Standard Mode this key allows you to correct typing mistakes: pushing it moves the cursor back one space, deleting the previous character. (Remember, [DEL] always gets rid of the character to the left of the cursor.) A [SHIFT]ed [INST/DEL] allows you to insert one letter, symbol, or space. To insert more than one character, you must type [SHIFT]:[INST/DEL] before typing in each new character.

In Quote Mode, [DEL] still deletes, but [INST] prints a reverse field graphic symbol. Pressing [COMMODORE]:[INST/DEL] in Quote Mode will print the graphic for [DEL].

Space Bar

Hitting the space bar on a typewriter puts empty space between words. On the Commodore the space bar also puts a space between words, but the space is not empty: it holds a perfectly distinct character—just like A or Z—that simply looks like empty space.

This means that you can't use the space bar to advance quickly to the end of a typed line, as you can on an electric typewriter. If you try it, you'll replace each character with a space and wipe out the line. (This feature can actually come in handy when you start typing long programs.) If you want to leave your text intact, you must use the cursor keys to move around.

One more note: in Quote Mode, where you put spaces affects how your PRINTed display will look.

[SHIFT/LOCK]

Like its counterpart on a typewriter, this key locks on the [SHIFT]

functions. But a word of warning: the down and right cursor functions won't work. To unlock the [SHIFT] just push the [SHIFT/LOCK] key a second time. (This feature of pushing a key twice to get back to the original state is called a *toggle*.)

[<], [>] and [↑]

[<] means “less than,” [>] means “greater than,” and [↑] means exponentiate. The shifted [↑] key, or [π] allows you to use an approximate value of PI (3.14159264 — there's more but no one's ever reached the last digit). PI is a number that occupies a special place in nature.

Function Keys

Our look at the keyboard ends with the four FUNCTION keys located on the extreme right. Each key has two functions: one un[SHIFT]ed and one [SHIFT]ed. In future chapters, we will program these keys to perform in a number of different ways.

2 BASIC Number Crunching

Computers, like elephants, never forget. The computer keeps track of vast arrays of information. But unlike elephants, computers are very fast. They're fast because the microprocessor is a real whiz at getting information in and out of memory.

You can think of computer memory as a wall full of cubbyholes or post office boxes, each labeled with an address and able to contain a finite amount of information. That information is in the form of numbers having only two digits, a 1 and a 0. Such numbers are called binary numbers, and the digits are called bits. Inside the computer's circuits, each series of 1's and 0's translates into a series of electronic on-off switches. When you type in a program, your data and program steps are stored in 8-digit, or 8-bit, units called bytes (such as 101100010). The 64 in "Commodore 64" refers to your machine's total memory capacity — just over 64,000 (64K) bytes of information. Like an Indonesian goddess with dozens of arms, the microprocessor reaches into the appropriate memory cubbyhole, pulls out an instruction, executes it, and returns it to the same cubbyhole. Your instruction remains at that memory address until you need it again, clear it out, or turn your computer off.

You can think of a series of adjacent addresses controlling a particular function as forming a block. Both you and the microprocessor have access to certain blocks — those that are part of Random Access Memory, or RAM. The computer uses these blocks to build shapes, play music, or crunch numbers according to your wishes. RAM, then, is literally a place where you can play with your computer.

Commodore 64 BASIC provides the interface between you and the binary numbers that your computer understands. Developed by two Dartmouth professors in 1964, BASIC allows you to interact with your

machine using English verbs. (For a glossary of Commodore 64 BASIC vocabulary and syntax, refer to Appendix C, pages 112-119 of the *User's Guide*.)

From the keyboard, you can command your Commodore in two ways: by issuing a single-line instruction and getting an immediate response, or by typing in a formal program consisting of a numbered list of instructions. In this chapter we'll get acquainted with the immediate form of BASIC; Chapter 3 will introduce formal programming. But first...

BASIC SURVIVAL KIT

When you first turn on your computer or press [RUN/STOP]:[RESTORE], the word **READY** appears on the screen. The word is called a prompt and means just what it says: the Commodore is **READY** to accept input. **READY**, however, does not necessarily mean that the computer's memory is clear. If you've been practicing all day, even with single-line commands or graphic symbols, a lot of numbers could be floating around in the memory cubbyholes. To clear them out completely, type **NEW** and press [RETURN] below the **READY** prompt.

Because **NEW** clears out all the old information, use it carefully! If you want to save your programs, follow the instructions on pages 21-22 of the *User's Guide* before typing **NEW**. If not, pressing [RUN/STOP]:[RESTORE] and typing **NEW** followed by [RETURN] will give you a completely fresh start.

If you don't need a fresh start but just want to clear the screen, press [SHIFT]:[CLR/HOME].

One of the most delightful features of this microcomputer is the screen editor, which allows you to move, add, or delete text anywhere on the screen. (We'll exercise this feature fully in Chapter 4.) When typing in a line, use the [INST/DEL] key to backspace and delete mistakes; until you hit [RETURN] the computer cannot read your typos. You can correct past mistakes or change what you've typed simply by moving the cursor over the error and typing the correction. Hitting [RETURN] enters the corrected version onto memory.

A word about spaces: Commodore 64 BASIC doesn't care whether you put spaces after commands. We use extra spaces in the listings in this book just to make the program easier for you to read. **PRINT "HELLO"** plus [RETURN] will give the same result as **PRINT"HELLO"** plus [RETURN].

BASIC *does* care about punctuation and about spaces within quotation marks. **PRINT 1,3333333,,5,7** is not the same as **PRINT 1;3;5;7**,

and PRINT "COMMODORE 64" is not the same as "COMMODORE64". As you can see, the PRINT command prints whatever is inside the quotation marks exactly as is.

Here is a list of symbols that have special meanings in BASIC; we'll explore their functions further as we use them:

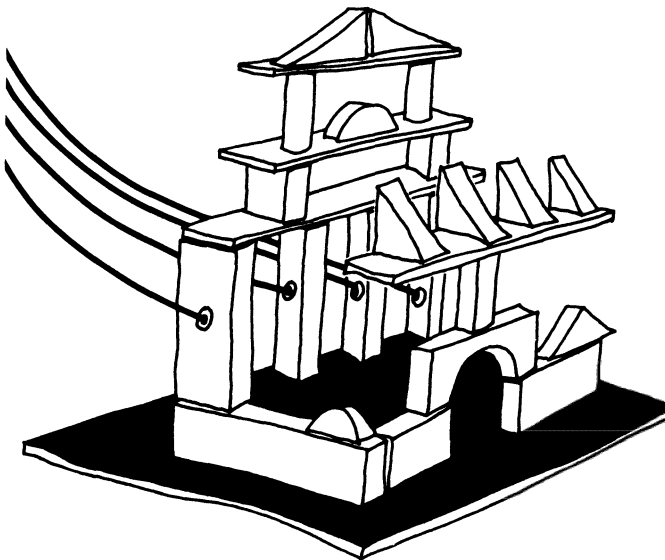
- \$ identifies a string variable
- % identifies an integer variable
- ? is an abbreviation for the PRINT statement
- : separates different BASIC statements used on the same line
- , separates variables in DATA lists; puts space in a PRINTed display
- ; indicates no space or carriage return after a PRINT statement
- . functions as a decimal point

MANIPULATING MEMORY

Certain post office boxes, or addresses, within the Commodore's memory are set aside for special purposes. Two sets of addresses turn the graphics and sound features on and off. Another 1000-byte block beginning at address number 55296 controls the colors you see on the screen.

You can control the information at different memory addresses directly from the keyboard with the BASIC command POKE. Type:

```
POKE 53281,0
```



Now press [RETURN]. The screen turns black! Bring the cursor up over the 0 and type 5; the screen is now bright green. 53281 is the memory address for screen color; the number 0 is the code for black, and 5 is the code for green. You'll find a complete list of color codes on page 61 of the *User's Guide*. Experiment with other color codes.

To get back to normal, type color code 6 or press [RUN/STOP]: [RESTORE].

Two other useful immediate BASIC commands are LIST, which displays any program stored in memory, and RUN, which executes that program. You'll use these commands frequently.

MANIPULATING NUMBERS

In its simplest form, your Commodore 64 is a calculator. Using the PRINT command without quotation marks and the +, -, *, /, and = keys tells the computer to add, subtract, multiply (*) or divide (/) and puts the results on the screen. If you type:

```
PRINT "256/8 = "; 256/8
```

and hit [RETURN], you should see 256/8 = 32 printed below the line you just entered. The semicolon indicates that you want the result of 256/8. (For more arithmetic, see pages 22-29 in the *User's Guide*.)

You can enter the above calculation in another way. Clear the computer's memory by typing NEW on an empty line and pressing [RETURN]. Clear the screen with [SHIFT]:[CLR/HOME]. Now type (being sure to hit [RETURN] after each line):

```
A = 256
```

```
B = 8
```

```
PRINT A/B
```

If you enter these lines accurately, your Commodore responds with READY after the first two and with 32 after the last one.

What you've just done is to define two variables named A and B, assigning them the numerical values of 256 and 8, respectively. A variable is a name for a quantity that varies. The temperature, your bank balance, or your weight all are variables. Because the 64 never forgets (unless you tell it to or the power fails), it responds to your PRINT statement by performing the requested calculation using the variable values you entered.

Variables are among the most useful creatures in programming because

a simple name like A or B can represent a great deal of information. Once you define your variables, you need only keep track of their names, and manipulating short names is easier than manipulating long expressions. For example, you could redefine A and B and add a new variable like this:

```
A = 256/8
```

```
B = 2*4
```

```
C$ = "A DIVIDED BY B = "
```

```
PRINT C$;A/B
```

Did you type in each line followed by [RETURN]? Then you should have gotten:

```
A DIVIDED BY B = 4
```

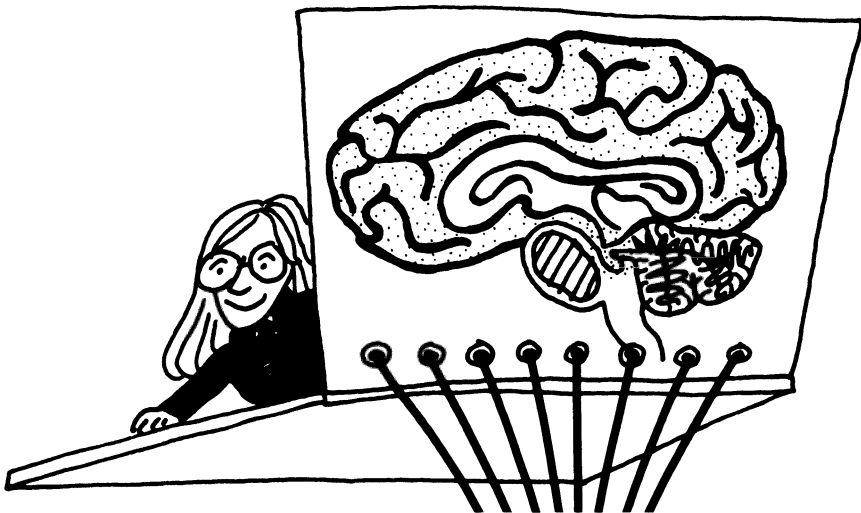
as your last line.

The \$ in C\$ means that what follows—the variable's value, or definition—will contain a string of text; such variables are called string variables (which we'll explore these further in Chapter 12). A % after a variable name (e.g., X%) defines an integer variable. If you type:

```
X% = 1.67
```

```
PRINT X%
```

the Commodore will PRINT only the 1, truncating what comes after the decimal point to the whole number (integer) alone.



You should keep in mind a few rules when naming variables. First, a variable name can have one or two characters. The first must be a letter of the alphabet, and the last indicates the type of variable. As we just saw a \$ indicates a string variable and a % indicates that the variable contains a whole number. The computer will accept longer names, but will read only the first two characters. Thus, COUNT and COUNTESS both reduce to CO, and the computer reads them as the same variable.

Second, variable names cannot be or contain any BASIC keyword. VALUE won't work, for example, because VAL is a BASIC keyword (in this case a function.) For a list of BASIC keywords, see Appendix D, pages 130-131 in the *User's Guide*.

Variables can be updated: if you change their values, as you did for variables A and B in the example on page 0, the new values replace the old ones. In ordinary algebra, the statement $X = X + 3$ cannot possibly be true. But to the computer, that statement makes perfect sense: it is an instruction to update the value of X. The computer simply adds 3 to the current value of X and gives the new sum the same name. It can continue to add 3 until you tell it to stop.

3 A BASIC Sampler

Learning how to program means learning how to give systematic instructions. No matter what language you use, you need to break your programming task into a logical sequence of steps, each expressed by a single statement in that language. If you are organized in your approach, you will be able to expand your programs and find errors (or **debug**) easily. This chapter will introduce BASIC programming techniques that will not only be used later in this book, but can also be applied to other computer languages.

In the last chapter you entered program steps one at a time, and the computer responded after each. If instead you number the steps and enter them into memory by hitting [RETURN] after each one, the computer waits for the command RUN before performing your instructions. As long as you keep typing numbered lines in BASIC, the Commodore 64 automatically adds them into memory, creating a single program. When you type RUN on a blank line followed by [RETURN], the computer performs all the numbered instructions in sequence. Keeping track of line numbers is therefore a must. The LIST command can help you with this. When you type LIST followed by [RETURN], all numbered program lines are displayed in numerical order.

If, while reading this book for instance, you type in several different programs in a short time, you should SAVE each one or clear the computer's memory by typing NEW before entering the next program. If you don't, the Commodore will (conveniently) write over the lines with the same line numbers, but old lines with different numbers will remain in the new program. Although you can LIST and RUN lines selectively (and thus effectively store more than one set of instructions in memory; see Chapter 4 and pages 115 and 116 in the *User's Guide*), long pro-

grams will behave strangely. Remember: The Commodore 64 does *everything* you tell it to, even when you've forgotten what you told it and when.

A last word of warning: While experimenting you may, unwittingly, issue a command that **crashes** the computer—the cursor freezes or disappears, nothing you type appears on the screen, and pressing [RUN/STOP]:[RESTORE] has no effect. If this happens, turn the machine off, wait a few seconds, then turn it on again. The good news—you haven't damaged anything; the bad news—you've lost anything entered into memory but not yet SAVED. So, before you experiment too radically, SAVE your programs!

A QUICK FORAY

For a clear slate, type [RUN/STOP]:[RESTORE] and NEW followed by [RETURN]. Now type in the following program (don't forget to press [RETURN] after each line):

```
10 LET X = 10
20 Z = X^2
30 PRINT "TEN SQUARED IS ";Z
```

On the next line type RUN and hit [RETURN]. The following two lines should appear on your screen:

```
100
TEN SQUARED IS 100
```

This program follows the patterns used in Chapter 2, but because the lines are numbered, the computer waits for the RUN command to perform the calculations.

Remember how, in Chapter 1, you created

A LITTLE SONG

AND DANCE

using PRINT and Quote Mode? Here's a fancier example of controlling displays (PRINTed output) using the TAB function to insert spaces. The 64 has no TAB key, but the BASIC TAB(x) function does the same thing; X can be any integer up to 255. (You can get the heart symbol in

Line 10 by pressing [SHIFT]:[CLR/HOME]—you are in Quote Mode.)

```
10 PRINT "♥" : REM *** CLEAR SCREEN ***
20 PRINT "NOW HERE" TAB (25) "NOW THERE"
30 PRINT : PRINT : PRINT
40 PRINT "A BIG" TAB (250) "JUMP"
```

The REM statement in Line 10 stands for REMark and can be followed by any text. It's a note to yourself or to anyone reading your program and can keep long programs organized and easy to understand. The 64 ignores your REMarks when it RUNs a program.

In Line 20 the TAB function PRINTs "NOW THERE" 25 spaces from the left margin. As on a typewriter, TAB is a handy feature for preparing charts and tables.

The three PRINT commands in Line 30 guarantee that A BIG will print three lines below NOW HERE and against the left margin. Go ahead and experiment with different TAB(numbers). Remember that no change is official until you press [RETURN].

The next program illustrates some different ways of assigning values to variables. Type:

```
10 LET A = 3
20 B = 5
30 INPUT "ENTER A NUMBER BETWEEN 1 AND 10";COP
40 LET EX = (A*B)/COP
50 PRINT " EX = ";EX
```



Did you remember to press [RETURN] after every line? Good; RUN your program. The computer asks you to enter a number. Go ahead, enter a number as directed and hit [RETURN].

INPUT is a convenient way of assigning variable values and illustrates what people mean when they describe BASIC as an interactive language. When the computer reaches an INPUT statement, it stops, prints out the phrase inside the quotation marks and waits for you to type in a value. It then takes that value and automatically assigns it to the variable named after the semicolon—in this case, COP. INPUT statements are particularly helpful when you want the computer to run the same long procedure with many different variable values.

Line 10 declares that until further notice A has the value of 3.

Line 20 is similar but shows that LET is optional.

Line 30 stops the program to ask for your INPUT.

Line 40 demonstrates a powerful way to use variables: the variable EX is defined in terms of the current value of other variables.

Line 50 PRINTs the value of EX.

What happens if you add:

```
60 GOTO 10
```

to your program? When you get tired of INPUTing numbers, hit [RUN/STOP]:[RESTORE]. You have just broken an infinite loop.

The following program introduces more BASIC elements.

```
10 A = 3
```

```
20 A$ = "HOW DID I DO THAT?"
```

```
30 INPUT "TYPE IN A NUMBER AND HIT RETURN";  
NUMBER
```

```
40 READ F,G
```

```
50 DATA 1,2,2,3
```

```
60 PRINT A : PRINT "F = ";F : PRINT A$
```

Lines 10-30 define your variables.

Lines 40 and 50 demonstrate a way to enter more than one variable at a time. The READ and DATA statements are a matched pair. When the computer sees the READ statement, it goes to the DATA statement and reads in the first value, assigning it to the variable F; it then reads the second value, assigning it to the variable G. The other two numbers in the DATA statement simply unREAD in memory until called up by

another READ statement or other appropriate instruction (more on this in later chapters). A **pointer** inside the computer keeps track of which DATA values have already been READ (see pages 93-94 in the *User's Guide*).

Line 60 PRINTs the current values for A, F, G, and A\$. The colon separates the four different PRINT statements.

But whatever happened to that NUMBER you put in? Like the two extra numbers in the DATA statement, it too, is still in memory. Type:

PRINT NUMBER

hit [RETURN], and see.

The last program in this chapter demonstrates a simple FOR/NEXT loop:

```
10  FOR I = 1 TO 10
20  PRINT I
30  LET SUM = SUM + I
40  NEXT I
50  PRINT "THE SUM IS: "
60  PRINT SUM
```

The FOR statement in Line 10 starts a sequence that says: FOR those values of I from 1 to 10, follow the directions in Lines 20 and 30. Then (Line 40) get the NEXT value of I and do the same thing.

Line 30 defines SUM as SUM (all variables start at 0) + I.

Line 40 sends the computer back to the NEXT value of I to repeat the sequence.

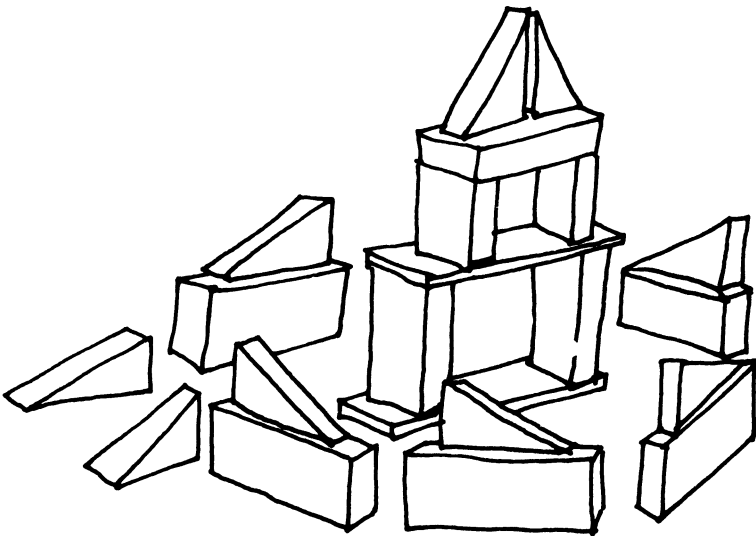
Lines 50 and 60 PRINT the final result: the cumulative sum of the numbers 1 through 10.

4 Program Editing

The screen editor inside your Commodore 64 makes correcting mistakes or changing your programs incredibly easy. You can use the editing features as you create programs, or you can go back and polish SAVED work later. Best of all, you can use editing to test different versions of a program and to build up variations on a successful theme.

Here's a familiar theme to fine-tune your editing skills. Through editing, you can turn this program for a song with three voices into other songs, so when you know it works, SAVE it on disk or DATASSETTE.

Some last reminders: be sure to hit [RETURN] after typing each numbered command. (If a command runs over 40 spaces, the computer



will automatically carry it onto the next line.) The REMark statements are optional. If you choose not to use them, make sure you also delete the : that precedes them. And finally, spaces are irrelevant to the Commodore except inside quotation marks, but they do make your programs easier to read.

```
5  REM *** THE FIFTH ***
10  PRINT "♥" : REM *** CLEAR SCREEN
15  FOR M = 54272 TO 54296 : POKE M,0 : NEXT
20  POKE 54296,15
30  POKE 54277,88 : POKE 54278,89 : REM ADSR1
40  POKE 54284,88 : POKE 54285,89 : REM ADSR2
50  POKE 54291,88 : POKE 54292,89 : REM ADSR3
60  READ A,B,C,D,E,F,G
70  IF A < 0 THEN GOTO 900
80  POKE 54273,A : POKE 54272,B : REM HF1, LF1
90  POKE 54280,C : POKE 54279,D : REM HF2, LF2
100 POKE 54287,E : POKE 54286,F : REM HF3, LF3
110 POKE 54276,33 : POKE 54283,33 : POKE 54290,33
120 FOR Z = 1 TO G : NEXT
130 POKE 54276,32 : POKE 54283,32 : POKE 54290,32
140 GOTO 60
200 DATA 15,210,2,233,3,244,150
210 DATA 15,210,2,233,3,244,150
230 DATA 15,210,2,233,3,244,150
240 DATA 12,143,6,71,3,35,1050
250 DATA -1,-1,-1,-1,-1,-1,-1
900 POKE 54296,0
910 END
```

Type RUN on a new line and hit [RETURN]. Any luck? Sorry about the pun...

If your program didn't run, let's figure out why. Even if it did, use

this section to familiarize yourself with editing—a vital skill for debugging programs.

Perhaps you made a syntax error, say in Line 40, and you got this message:

```
? SYNTAX ERROR IN 40
```

```
READY
```

To find the error, type LIST 40 on a new line. You should now see your Line 40. Check it against ours. Are all your commas really commas, or is one a period? How about your zeros? Are they really zeros or are they the letter O? Are your POKE numbers the same?

Here is a sample Line 40 for you to change. If you have a similar error in a different line, follow these directions.

```
40 POKE 54284,88 ; PAKE 52485.89 : REM ADSR2
```

Bring your cursor up to this Line 40 or to your LISTed line. Then bring it over the semicolon after 88. When the cursor is flashing on the semicolon, press a colon on your keyboard. Now move the cursor over the A in PAKE and type in an O. Hit [RETURN] to enter your corrections. If you don't, the screen may look right but the version inside the computer won't have changed.

Go to a blank line, type RUN, and hit [RETURN].

Hmmm...still a syntax error in Line 40. Once again, type LIST 40 on a new line and hit [RETURN]. Look closely: the period in POKE 54285.89 should really be a comma. Bring your cursor up and over, type in a comma, and hit [RETURN]. RUN the program again. Now does it work?

Yes? Good! No? Keep trying. Are all your syntax errors corrected? If not, go back and list each line that has an error and try to find the problem.

Does your program run but give you only one brief sound? That could happen for many reasons. Try this: On a new line, type LIST -100 and [RETURN].

LINE LISTING

Now you should see displayed on your screen all the lines from 0 through 100. Typing LIST plus a minus sign before the line number LISTs all the lines up to and including that line—here, the first section of your program.

Before going on, let's go over other ways of LISTing. On a new line, type:

LIST 100-

This gives you a display from Line 100 to the end of the program. You can also:

LIST 60-200

[RETURN]

which will display the lines from 60 through 200. If the LIST moves (scrolls) by too fast to read, hold down the CTRL key.

NON-SYNTAX ERRORS

Let's return to Lines 0-100. Is your READ Statement in Line 60 correct? Let's say you mistyped a letter, and your line looked like this:

60 READ A,B,C,D,E,F,V

Because the format, or syntax, of this command is correct, the computer will not find a syntax error, and will run the program. The program will work fine until it reaches V, then it will END, and the computer won't give you any hint of what went wrong. If this happens, you'll need to do more sophisticated debugging than you're using in this chapter. (Hint: look at Line 120.)

For now, however, let's concentrate on the editing. Bring the cursor up to the V and change it to a G. Hit [RETURN], and try to RUN your program.

Once you get your program running properly, SAVE it by typing on a new line:

SAVE "THE FIFTH",8

for the disk drive, or just

SAVE "THE FIFTH"

for DATASSETTE storage.

By the way, if you're curious about all those POKE numbers, you might want to skip ahead to Chapter 10, "SID's Song." Meanwhile, let's take the Fifth and amend it.

ADDING, ERASING AND RENUMBERING

You've seen how easy it is to use LIST to help make changes in existing program lines. Adding and erasing lines is even easier. To add a line, just type in a numbered line after the READY prompt followed by [RETURN]. To erase, type the line number on an empty line immediately followed by [RETURN]. When the program LISTs again, additions will have been incorporated, and unwanted lines will have disappeared.

Often when you add lines to a program, you have to renumber some of the original lines, either for clarity or to conform to the rules of BASIC. Say we want to add four DATA lines to "The Fifth," numbered 250, 260, 270, and 280. But we already have a Line 250 that we need at the end of the DATA statements for the program to run correctly. To preserve its information, let's renumber it 300.

Here's how: type LIST 250 and hit [RETURN]. Now bring the cursor up and change the 250 to 300. The line should read:

```
300 DATA -1, -1, -1, -1, -1, -1, -1
```

Hit [RETURN] to enter the change.

On a new line, type LIST and [RETURN]: Your program will now have a new line numbered 300. But you'll also notice that Line 250 is still there; it will stay the same until you enter a new Line 250. Do this by typing:

```
250 DATA 14,24,7,12,3,134,150
```

and pressing [RETURN] at the end of the line. To check your change, LIST the program again. Now add the following, pressing [RETURN] at the end of each line:

```
260 DATA 14,24,7,12,3,134,150
```

```
270 DATA 14,24,7,12,3,134,150
```

```
280 DATA 11,218,5,237,2,246,1000
```

```
RUN
```

Let's put a silence between the two phrases. Add:

```
245 DATA 0,0,0,0,0,0,750
```

Press [RETURN], now LIST. As you see, all the new lines have been added in numerical order even though they were not typed in that order.

The computer arranges the lines automatically as one of its editing functions.

RUN your program to see it if works.

INSERTING CHANGES

Often you will want to add information to existing lines but find there isn't enough room. Time to use the INSeRT function.

On a new line, type LIST - 40 to display Lines 5 through 40 on your screen. Let's change the 88 in Line 30 to read 137. In this instance, we have to put three numbers into a place currently occupied by two. To do this, bring your cursor up to Line 30. Then bring it over so that the cursor flashes on the first 8 of the 88. Press [SHIFT]:[INST/DEL].

If you did this properly, the whole text starting with the first , moved one step to the right, leaving the cursor where the 8 used to be.

Now type 137. Hit [RETURN] to enter the change.

Now change the 89 to a 128: Bring the cursor over the 8 of 89; press [SHIFT]:[INST/DEL], and type 128. Press [RETURN].

Note that once you command the computer to make room for an INSeRT, you must fill the space even if you only hit the space bar. If you attempt to use DELeTe to correct what you're inserting, you will print the reversed field graphic symbol for DELeTe.

Now for a surprise. On a new line, type LIST - 40 again. You should now have a blank line between Lines 30 and 40. By INSeRTing your new numbers, you carried Line 30 out a full 40 characters. (Remember, spaces count as characters.) The cursor therefore moved automatically down to the next line. Watch out! The 64 reads that blank as part of Line 30 so don't try to put another numbered command there.

And another thing: the computer can only handle two full lines, or 80 characters in one instruction. Therefore, when entering any BASIC statement or DATA lines, you *must not* exceed two lines. If you do, anything running over the second line will not be entered into memory.

Remember, too, that if you use alternate symbols for BASIC words (such as ? in place of PRINT), the entire word will appear when you LIST. Therefore, information that might have fit on two lines with an abbreviation will not necessarily fit on two lines after it is LISTed, and your Commodore works with the LISTed version.

EDITING IN QUOTE MODE

In Chapter 1, you saw that typing a set of quotation marks always puts the computer into Quote Mode. Everything to the right of the quotation mark will be in Quote Mode. Typing the closing quotation marks gets you back to Standard Mode.

Remember, the cursor controls stop functioning when in Quote Mode, printing their reversed field graphic symbols. Therefore, if you make a mistake and wish to edit, you must use the DELeTe key, which is the only editing key that works normally in Quote Mode. Of course, you can also finish the line and enter it with the mistake; then you can go back and use the cursor normally—it's awkward, but it works.

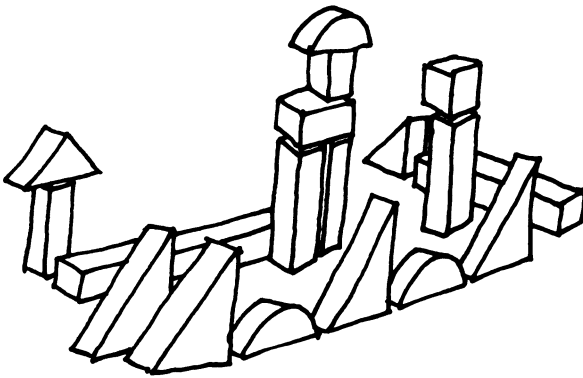
Let's make a final change by retyping Line 5:

```
5  REM *** THE FIFTH AMENDED ***
```

Type RUN, say AMEN, and press [RETURN].

5 The Case of The Cursory Cursor

Here is simple game that will make using the cursor controls second nature to you. The program will place a letter and the cursor at two random points on the screen. Use the cursor controls to bring the cursor over the letter. The amount of time you took will then appear briefly at the top of the screen, and the program will loop back to let you play again. Try to get your average time below six seconds. To stop the program press [RUN/STOP]:[RESTORE]; or, if all else fails, turn off your computer. Make sure to SAVE your program. We will return to and expand this program in Chapter 8.



CURSORY CURSOR

```
2  REM *** CURSORY CURSOR ***
4  REM *** n.b.scrimshaw
12 POKE 53280,11 : POKE 53281,0
16 GOTO 150
19 REM *** PICK POINT ***
20 LET CP = INT(RND(1)*1000)
35 RETURN
49 REM *** SET LETTER ***
50 GOSUB 20
60 LET A = INT(RND(1)*26) + 1
70 LET W = CP
80 POKE CP + 1024,A
90 POKE CP + 55296,3
95 RETURN
99 REM *** GET CURSOR ***
100 GOSUB 20
110 POKE CP + 1024,83
120 POKE CP + 55296,1
125 RETURN
149 REM ***** MAIN LOGIC UNIT**
150 PRINT " [♥] " : REM *** CLEAR SCREEN
160 GOSUB 50
170 GOSUB 100
180 LET SC = TI
190 GET A$ : IF A$ = "" THEN 190
200 LET CD = ASC(A$)
204 POKE CP + 55296,9
210 IF CD = 29 THEN CP = CP + 1
```

```
220 IF CD = 157 THEN CP = CP - 1
230 IF CD = 145 THEN CP = CP - 40
240 IF CD = 17 THEN CP = CP + 40
250 IF CP < 0 OR CP > 1000 THEN CP = 1
260 IF CP = W THEN 300
270 GOSUB 110
280 GOTO 190
299 REM *** KEEP SCORE ***
300 PRINT "YOU TOOK ";(TI - SC)/60;" SECONDS"
310 AT = AT + (TI - SC) : TRIES = TRIES + 1
320 PRINT "YOUR AVERAGE IS ";(AT/60)/TRIES;" SECONDS"
330 FOR I = 1 TO 1500 : NEXT I
333 GOTO 150
```

“One night I was asleep and dreamed that
I was awake, then I woke and realized
that it was just a dream. . . .”

6 Loops in Loops in Loops

Loops, which we first saw in Chapter 3 and have been using ever since, are powerful tools that deserve a closer look. Loops can be nested and, when teamed with IF/THEN statements can make your program fold into itself like an Escher design.

First, let's use loops to predict the result of tossing ten coins. Say you want to know the probability of getting exactly eight heads. To calculate this directly, you need to know that there are 1024 (2^{10}) different possible outcomes, and 45 of them have exactly eight heads. Dividing these two numbers gives you the probability: $45/1024 = .0439$.

Many problems of probability are far harder than this, and it is sometimes difficult just to figure out how to approach them. One tool for estimating probabilities is **simulation**, that is, using mathematics and the computer to mimic the problematic situation. Simulation works because of the law of averages: even though individual trials of some chance phenomenon vary in outcome, the average of many trials varies only marginally from a single number, the probability. The greater the number of trials, the more accurate the probability estimate. But once you reach a certain number of trials, the accuracy gained from still more trials is minimal. (That's why polling companies are able to poll relatively few people and still predict how national elections will turn out.)

LOOP TECH

Before approaching the coin problem we need to take a look at loops.

We have been using FOR/NEXT loops, but now we must use a new creature: the nested loop. Enter this short program (to form the inverse Q in Line 20, press the down cursor [LCRSR] while in Quote Mode):

```
10  FOR H = 1 TO 5
20  PRINT "PASS# ";H;" Q ";
30  NEXT H
```

Typing H after the NEXT in Line 30 is optional but helps keep your bookkeeping straight.

When this program RUNs, the computer follows the instructions in Line 20 five times. Notice that inside each pass of the loop, the current value of H (called the counter variable) is printed onto the screen. This will help you keep track of what is going on. If we are in loop number five, the computer prints:

```
PASS# 5
```

When the computer reaches the down cursor symbol, the cursor obediently shifts down one line; the semicolon at the end of Line 20 holds it in the same column. With Line 20 completed, the computer moves to Line 30, and follows that instruction by going on to the next value of H. The old value is incremented by one so that 6 is the value of H after PASS# 5. To check this, type PRINT H.

The word STEP can be added to the basic FOR/NEXT loop to adjust what the computer counts by, or the increment size (see page 119 of the *User's Guide*). Using the screen editor, change Line 10 so that it looks like this:

```
10  FOR H = 1 TO 21 STEP 2
```



**Double Breasted
Nested Loop**

RUN your program.

STEP has made the variable H count to 21 by twos. Since there are 10 increments of two, there are 10 passes through the loop.

STEPS can also be negative. Clear out the old program with NEW and try this one:

```
10 FOR I = 20 TO 1 STEP -1 : PRINT "DOWN";I : NEXT I
```

Here we have compressed the whole loop into a single line using the colon to separate statements. One-line loops tend to make a program more readable.

RUN this program and trace the value of the loop counter as each line is printed.

A CLASSIC TRIANGLE PROGRAM

Let's do one more program with loops before we flip coins. Clear the computer with the NEW command and type:

```
10 FOR P = 1 TO 10
20 FOR H = 1 TO 20
30 PRINT"*";
40 NEXT H
50 PRINT
60 NEXT P
```

We are using the semicolon in Line 30 to control print format. When the inner loop, begun on Line 20, has printed out a line of 20 asterisks, it exits the loop and encounters the PRINT on Line 50. Since there is nothing to print, the 64 prints nothing. But, this time there is no semicolon. So the cursor gets bumped to the beginning of the next line. The next pass through the inner loop thus has a fresh line to work with. RUN the program and you will see a block of asterisks appear on the screen. If you're not sure of how Line 50 works, simply erase it by typing 50 and [RETURN] on an empty line and reRUN the program.

Now, to see a classic application of loop-variable technology, LIST your program and amend Line 20 to read:

```
20 FOR H = 1 TO P
```

Don't forget to hit [RETURN] after you make your change. Before RUNning the amended program, try to predict on paper what this single change will accomplish.

Now we are ready to use the computer to answer questions that have obscure precise answers. . .

THE RND FUNCTION: HEADS OR TAILS?

The RND (or RaNDom number) function of the Commodore 64 produces random numbers that can effectively simulate throwing dice, or shuffling cards, or flipping coins. (See your *User's Guide* for more details.)

Flipping coins results in either a head or a tail. In this simulation, we will encode the outcome with 1 for heads or 0 for tails. The RND function will help us by simulating a random selection of a 0 or a 1. The program will correctly interpret the result and keep the overall tally.

With the RND function and the power of loops we can build a program that will calculate the approximate probability of getting eight heads when flipping ten coins. The program uses a FOR/NEXT loop to perform 100 trials, each simulating flipping 10 coins. Thus, we want to see how many times out of a hundred there are exactly eight heads. Every time a trial comes out with eight heads we'll increment a counter variable (ET) by one. This will give us the information we need to estimate the probability. Clear the computer with the NEW command and let's get started.

First we tell the computer to do something 100 times:

```
9  REM *** DO 100 TIMES ***  
10 FOR TR = 1 TO 100
```

Now we describe what that something is:

```
20 REM *** FLIP COIN 10 TIMES ***  
30 FOR FL = 1 TO 10  
40 H = INT(RND(0)*2)  
50 ET = ET + H  
60 NEXT FL  
70 IF ET = 8 THEN COP = COP + 1
```

Line 40 does the work of randomly choosing a zero or a one and then assigns that value as the new value of H. ET is our counter; it counts

the total number of heads. If RND has selected a one then $H = 1$ and ET gets incremented by H. COP counts how many times ET is equal to eight. After the inner loop is finished, the program reaches Line 70 and asks: is ET equal to eight? IF ET equals eight THEN the counter COP gets bumped up by one.

Now we need to close off the outer loop we started back on Line 10 and calculate the estimated probability. Line 80 closes our loop:

```
80  NEXT TR
```

The probability of a given event may be expressed as a percent, as in "50 percent chance of rain." It may also be written as a decimal function of 1: 0.3 means 30 percent probable; 0.5 means 50 percent probable and 1.0 means 100 percent probable. Since the probability of getting eight heads from flipping 10 coins equals the number of trials in 100 that get exactly eight heads divided by the total number of trials, we add:

```
90  P = COP/(TR - 1)
```

```
100 PRINT "THE ESTIMATED PROBABILITY IS ";P
```

RUN the program (it takes several seconds). Oh dear! The value you get looks suspicious; maybe there's a bug.

To find bugs in situations like this we can use **tracers** to trace the value of key variables as the program runs. We ask the computer to print out the current value of the variables at key points in the computation. This slows the calculations but can often pinpoint problems. Insert the following line:

```
75  PRINT "ET = ";ET; " COP = ";COP
```

When you RUN the program now you should see immediately that COP stops changing and that ET keeps climbing. ET has not come home after the first pass. We intended that ET would be the count of how many times there were exactly eight heads in ten tosses. It worked fine the first time through the loop, but on the second pass, ET still had the count from the last pass. ET must be **re-initialized**, or reset to 0, to make our program work. Add the following line (which erases our tracer):

```
75  ET = 0
```

This line sets the counter ET back to 0 so it is ready for duty again.

The program now works correctly, and we should be able to adapt

it to answer similar questions. For example: what is the probability of getting at most eight heads? Change Line 70 to:

```
70 IF ET < 8 THEN COP = COP + 1
```

RUNning your program should give you a reasonable estimate. Remember to change it back to the original before we put the whole thing inside yet another loop.

A TRIPLE-NESTED LOOP

The last program illustrates a nested loop. The inside loop flips 10 coins. The outside loop performs the inside loop 100 times. Now to get an even more accurate estimate we shall take the average of five runs of 100 flips. Hmmm...how to devise a program that will flip 10 coins 100 times 5 times?

Add to our previous program the following line:

```
5 FOR Z = 1 TO 5
```

Line 5 starts our big loop. Now we insert a line that will add together the five estimated probabilities:

```
110 PR = PR + P
```

and close off the Z loop:

```
120 NEXT Z
```

and print out the computed average:

```
130 PRINT " THE AVERAGE ESTIMATE IS "; PR/(Z - 1)
```

Ooops! Another bug; haul out the tracers if need be. But, as you may see from the LISTed program, we have made a similar error as before—COP has to come home after each pass of the Z loop, so we add:

```
115 COP = 0
```

Before you RUN this program it might be nice to add the following line to clear the screen at the beginning. Remember that the reversed field heart inside the quotes comes from pressing [SHIFT]:[CLR/HOME] in Quote Mode.

```
2 PRINT"♥" : REM CLEAR SCREEN
```

7 Graphics

The Commodore 64 supports some very sophisticated tools for graphics. In addition to the symbols accessible through the keyboard, you have many other ways to design bright pictures and animate your programs. Using the POKE command to control screen color and write symbols on the screen is just the beginning.

The Commodore's impressive array of high resolution and multicolor graphics include eight independent TV screens called **sprites**. These features come from what is essentially a second computer inside the 64—the VIC II. A built-in memory switchboard with 47 reserved addresses, or **registers**, controls the VIC II chip. The POKE command activates and adjusts the VIC II's features by placing numbers into these 47 control addresses.

THE MAGIC OF 53248

Remember (from Chapter 2) that the computer's memory is a series of addresses, each holding one byte, or eight bits. Each bit carries a value of either 0 or 1. When we POKE in a number, say 243, the computer converts it into a code of 0s and 1s. It is this code that actually resides in any given memory address.

The VIC II switchboard starts at address #53248. You turn the 64's graphics features on and off by POKEing the right values into addresses 53248-53294. The following program will show you what spectacular results your POKEing can have. Before you begin, fill the screen with assorted symbols. Then, at the bottom, type:

```
10 POKE 53272,7
```

Proofread the line (spaces don't matter); now, without clearing the screen, RUN the program. Press various keys on the keyboard and watch the show. When you get tired of all that action press [RUN/STOP]: [RESTORE]. This convenient key combination will get you back to normal without erasing the program. Be careful to get POKE numbers right!

You have just changed the organizational structure of memory so that the computer lost track of which addresses contained which information—it lost track of the keyboard characters. This ability to move video memory is an advanced feature of your 64 (which will not be used in this book).

The VIC II chip is so versatile that this book can't hope to cover all of its features; we will concentrate on using sprites. Mastering the sprite switchboard will get you a long way toward understanding how to use the VIC II chip. Let's begin with a program that will move all eight sprites around on the screen.

SPRITE CHECKLIST

To play with sprites, we must turn them on and off, set the shape, assign colors, and define screen positions. This may seem like a lot of work, but we can actually do it all with a program that's only eight lines long!

So here's what we have to do:

1. Turn on sprites.
2. Set sprite color.
3. Define sprite shape.
4. Set sprite coordinates.
5. Set memory pointers.

And here's the program that does the job:

```
999  REM *** SQUAREFISH TANK ***
1000  FOR M = 2040 TO 2042 : POKE M, 13 : NEXT
1010  FOR W = 832 TO 832 + 62 : POKE W,255 : NEXT
1020  V = 53248 : POKE V + 21,255
1030  FOR J = 39 TO 46 : POKE V + J,J - 38 : NEXT
1040  FOR H = 0 TO 14 STEP 2 : GOSUB 1200
1050  POKE V + H,X : POKE V + H + 1,Y : NEXT : GOTO 1000
```

```
1200 X = INT(RND(0)*250+3) : Y = INT(RND(0)*140+50) :  
      RETURN
```

To clear the screen and turn the background black (to better show off the colors of our sprites), add:

```
10 REM *** INITIALIZATION ***  
20 PRINT "♥" : REM *** CLEAR SCREEN  
30 POKE 53281,0
```

Go ahead and RUN this program.

WHAT HAPPENED

Those little dancing squares are sprites. Stop the program by hitting [RUN/STOP] and then press [SHIFT]:[CLR/HOME] to clear the screen. The sprites are still right there because they are controlled independently by the VIC II chip. LIST your program and you will find both the sprites and your program filling the screen. To get rid of the sprites, press [RUN/STOP]:[RESTORE].

The only reason that this short program can control all eight sprites is the use of FOR/NEXT loops. In addition, we have included a convenient subroutine (a sort of subprogram) to keep choosing new locations for the sprites as the program runs. Subroutines, by the way, are handy because you can use them over and over in a variety of programs.

OUR EIGHT-LINE WONDER

Let's look closely at a line-by-line description of the job done by the commands in our program.

Line 1000 includes two statements (separated by a colon) that define a loop telling Sprites 0, 1, and 2 to take their image pattern from memory sector 13.

Line 1010 defines a loop that inserts into memory the code for a solid block sprite shape.

Line 1020 sets V equal to 53248 which is the first address of VIC II's switchboard. The second statement on this line, after the colon, switches on all eight sprites.

Line 1030. Here we use a little trick to change the color of the sprites. V + 39 contains the code for the color of Sprite 0. V + 40 contains the

color code for Sprite 1 and so on. We don't want to POKE in the same color code for each sprite so we tie the value that is POKEd to how far we worked through the loop. If you think about it, you will realize that color codes one through eight are POKEd into the color control addresses for Sprites 0-7. We used a loop, *and* the value that gets POKEd in changes in each pass of the loop.

Lines 1040 and 1050. It takes two lines to define this loop. In each pass through the loop, a random position in X,Y coordinates is chosen for the sprite active at that moment. This loop uses a subroutine: the last statement on Line 1040, GOSUB 1070, temporarily sends the computation careening through Line 1070 where it picks up new, randomly chosen values for X and Y. Line 1050 POKEs the values just retrieved by the subroutine into the appropriate address numbers. At this point the active sprite takes a hop on the screen. The last command on this line, GOTO 1000, is easy to understand. It tells the computer to go back to the instruction on Line 1000 and start the cycle all over again.

Line 1200 is the subroutine. It uses the RND function to select appropriate random values for X and Y. You could use a user-defined function here if you prefer. For longer formulas this can be a real time and space saver. Look at the description of the DEF FN command on page 118 of the *User's Guide* for the details.

You also might want to inspect the sprite switchboard in the Appendix.

Almost all programs involving sprites start with declaring the variable V to have the value 53248, the first address in the VIC II switchboard. If we want to fiddle with the other 46 addresses, or control registers, we can simply POKE in an increment of V such as POKE V + 21 rather than POKE 53269. Besides making the address numbers shorter and easier to remember, POKE V + a number reduces the chance of typos that could lead to a panicky [RUN/STOP]:[RESTORE].

PROGRAM PROJECT

Note that Sprites 0, 1, and 3 are solid blocks, while the rest are flickering odd patterns. If you look at Line 1000, you will notice we only POKEd three of the eight shape registers. Add a couple of lines to this program and try POKEing other values into the last five shape registers.

COUNTING WITH TWO FINGERS

Before we go on, we need to understand a little more about the computer's counting system. Believe it or not, there's more than one way

to count to 10. People have 10 fingers, so through the ages we've developed a number system with 10 digits ("digit" even means finger): 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Computers, in contrast, have only two fingers—off and on—so they count with only two digits: 0 and 1. Our counting system, called the **decimal** system, is thus based on the number 10; the computer's system, called the **binary** system, is based on the number two. Everything you feed into your Commodore 64 is translated electronically into strings of 0s and 1s.

Specific memory addresses in the Commodore contain a byte of binary code: eight ones or zeroes (eight binary digits, or bits) in a row. That series of bits has a numeric value that is interpreted by the computer as a command. But because the computer's counting system is based on the number two, 10110010 doesn't carry the same numeric value to the computer as it does to us. We would read 10,110,010 as "ten million one hundred ten thousand ten"; the computer reads it as 178. How can this be? This is where other ways of counting come in.

In our number system we can write:

0

1

2

3

4

5

6

7

8

9

10

11

etc.

Counting to 10 simply requires adding one to the rightmost place until we exhaust the available digits; then we started with 0 in the rightmost place again and add 1 in the next place to the left, called the tens place.

Because it only has two digits, the computer can only write:

0
 1
 10
 11
 100
 etc.

With only two digits, how can the computer write “two” in the binary system? The same way we write 10 in the decimal system: by going to 0 in the rightmost column or place and adding one in the next column to the left, the twos place.

The third place to the left in the decimal system is for hundreds—it takes us from 10 to 99 in the tens column to exhaust our digits. The third place to the left in the binary system is the fours place—it only takes to four to exhaust binary digits in the twos column.

Do you begin to see a pattern? If we label the 10 the **base** of the decimal system and two the base of the binary system, we can label each column as follows:

| Decimal | | | | Binary | | | |
|-------------------|-------------------------------------|-------------|------|-------------------|-------------------------------------|----------------------|------|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | | | 1s | | | | 1s |
| | | base | = 10 | | | base (written 10 but | = 2) |
| | base × base = b ² | = 100 | | | base × base = b ² | = 4 | |
| | base × base × base = b ³ | = 1,000 | | | base × base × base = b ³ | = 8 | |
| | base ⁴ | = 10,000 | | | base ⁴ | = 16 | |
| | base ⁵ | = 100,000 | | | base ⁵ | = 32 | |
| | base ⁶ | = 1,000,000 | | | base ⁶ | = 64 | |
| base ⁷ | = 11,000,000 | | | base ⁷ | = 128 | | |

In both cases, adding the values in each place will give us the total decimal value of each string of eight ones above: in base 10, $1 + 10 + 100 + 1000 + 10,000 + 100,000 + 1,000,000 + 11,000,000 =$ eleven million one hundred eleven thousand one hundred eleven; in base 2, $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$. Thus, 255 is the highest value possible for one eight-bit string.

We can calculate the decimal equivalent for any binary number—

any string of eight bits—by adding the values in each place:

| | | | | | | | | |
|---------------------|--------------------|------------------|------------------|------------------|----------------|----------------|----------------|---------------------------|
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | |
| | | | | | | | | $0 \times 1 = 0$ |
| | | | | | | 1 | $\times 2 = 2$ | |
| | | | | | 0 | $\times 4 = 0$ | | |
| | | | | 0 | $\times 8 = 0$ | | | |
| | | | 1 | $\times 16 = 16$ | | | | |
| | | 1 | $\times 32 = 32$ | | | | | |
| | 0 | $\times 64 = 64$ | | | | | | |
| 0 | $\times 128 = 128$ | | | | | | | |
| Total decimal value | | | | | | | | <u>$= 178$</u> |

Grasping the concept of counting to 255 with two fingers will help us figure out how to control the sprite switchboard. We will learn more about sprites in Chapter 12.

8 Program Design

In Chapter 5 you practiced cursor control with a program called *Cursory Cursor*. In this chapter, we take a closer look at how this program is put together. When you understand the underlying logic, you can add your own custom features to create all kinds of games.

Cursory Cursor has six parts:

1. Initialization.
2. Pick a point subroutine.
3. Set letter subroutine.
4. Get cursor subroutine.
5. Main logic unit.
6. Keep score.

The main logic unit ties together all six parts of the program, so we'll analyze it before describing the subroutines. Don't try to follow each step; the point is to get a feel for what lies behind the BASIC instructions. You will probably return to this chapter several times as you begin to learn more and wish to use some of the ideas in other programs.

THE MAIN LOGIC UNIT

The first step specifies what the program has to do. In English:

Line 150 clears the screen.

Line 160 randomly chooses a letter of the alphabet and places it at a random place on the screen.

Line 170 places the cursor at a random place on the screen.

Line 180 checks jiffy clock for starting time.

Line 190 pauses until the user hits a key.
Lines 200 stores the ASC code that corresponds to the key pressed in the variable CD.
Line 204 changes the heart at the cursor position to red.
Line 210. IF right cursor key was pressed THEN move right.
Line 220. IF left cursor key THEN move left.
Line 230. IF down cursor key THEN move down.
Line 240. IF up cursor key THEN move up.
Line 250. IF cursor is now off the screen THEN send it to upper right corner.
Line 260. IF last move hits the target, GOTO the Keep Score section at Line 300.
Line 270 places a white heart at the new cursor position.
Line 280 says GOTO Line 190 and wait for another key.

The subroutines are called up in Lines 160, 170, and 270. In BASIC:

```
149 REM *** MAIN LOGIC ***
150 PRINT "♥" : REM *** CLEAR SCREEN
160 GOSUB 50
170 GOSUB 100
180 LET SC = TI
190 GET A$ : IF A$ = "" THEN 190
200 LET CD = ASC(A$)
204 POKE CP+55296,9
210 IF CD = 29 THEN CP = CP + 1
220 IF CD = 157 THEN CP = CP - 1
230 IF CD = 145 THEN CP = CP - 40
240 IF CD = 17 THEN CP = CP + 40
250 IF CP < 0 OR CP > 1000 THEN CP = 1
260 IF CP = W THEN 300
270 GOSUB 110
280 GOTO 190
```

See? There really is some English behind all that code. Let's take a closer look at the subroutines.

SUBROUTINES

Subroutines (defined in Chapter 7, pages 5-6) aren't really needed, in this (relatively) short program, but because the program is designed to grow, they will prove useful. Using subroutines is partly a matter of style, and, like all programmers, you will develop your own style.

The GOSUB command we have used is essentially the GOTO command with a special feature attached. Like GOTO, GOSUB makes the computer branch off to the indicated program line (e.g., GOSUB 19 means go to the subroutine beginning at Line 19). The special feature involves the RETURN command at the end of the subroutine. The computer automatically returns to the *next line after* the GOSUB statement that sent it to the subroutine in the first place. Thus, you can use GOSUB to call the same subroutine from different parts of a long program, and the computer will always come back to the right line to continue the program.

The first subroutine we need is:

```
19  REM *** PICK A POINT ***
20  CP = INT(RND(0)*1000)
35  RETURN
```

This subroutine chooses a random number and assigns it to the variable CP. CP takes on one of 1000 binary code values, each code value representing one of the 1000 locations on the 64's screen. The RETURN on Line 35 then sends the program to the next line after the one that called the subroutine. In essence this subroutine chooses for us a location on the screen when we need one.

Remember, though, all subroutines called by a GOSUB call *must* end with a RETURN statement.

WHAT CAN GO WRONG

Enter the following program:

```
1  REM *** JUST A TEST ***
2  GOSUB 11
3  REM *** NO RETURN FROM THIS ONE! ***
11 PRINT "HI THERE"
12 GOTO 1
```

RUN this demonstration and 64K or no 64K, you run out of memory!
Now change Line 12 to:

```
12 RETURN
```

Now RUN it. Hmmm...RETURN without GOSUB error?

The problem is that after the first successful subroutine call, the program returns to Line 3, then Line 11 and—aha—hits that RETURN *without having been sent by a GOSUB call*. The moral of the story is that subroutines have to be blocked off from the rest of the program.

A NESTED SUBROUTINE

The next subroutine is a bit longer:

```
49 REM *** SET LETTER ***
50 GOSUB 20
60 A = INT(RND(0)*26)+1
70 W = CP
80 POKE CP+1024,83
90 POKE CP+55296,1
95 RETURN
```

The Set Letter subroutine first calls the Pick Point subroutine to get a location on the screen (these are the nested subroutines). It then puts a randomly chosen letter at that location. Line 60 gives us a **screen display code** that corresponds to a letter in the alphabet. (See Appendix E, page 132 of the *User's Guide* for more screen display codes.)

The next subroutine really serves as two, depending on which line the computation enters it. If you enter at Line 100 the cursor will be placed randomly. If you enter at Line 110 the computer will update the cursor position according to which cursor key you have just pressed while the game is on. Line 120 sets the color to white.

```
99 REM *** GET CURSOR ***
100 GOSUB 20
110 POKE CP+1024,83
120 POKE CP+55296,1
125 RETURN
```

The final section of the program is not reached by a GOSUB; here GOTO works fine: we don't want to RETURN after the score is tallied; we want to go back and improve our cursor control response time. The delay loop in Line 310 allows you enough time to read your score.

```
299 REM *** KEEP SCORE ***
300 PRINT "YOU TOOK";(TI-SC)/60;"SECONDS"
310 FOR I = 1 TO 1500 : NEXT
333 GOTO 150
```

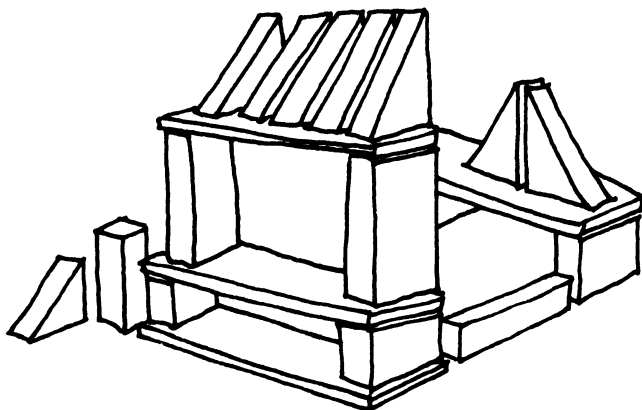
ALPHA-CURSORY CURSOR

That's that, but let's add a module to expand the game. Now after you reach the letter on the screen, the clock doesn't stop until you press the letter on the keyboard. We first change Line 260 to:

```
260 IF CP = W THEN 500
```

and add:

```
500 REM *** ALPHA MODULE ***
510 Q = PEEK (W+1024)
520 Z$ = CHR$(64+Q)
530 GET A$ : IF A$ = "" THEN 530
540 IF A$ = Z$ GOTO 300
550 GOTO 530
560 REM
```



CENTIPEDE

If you find the time, here's a program that shows how you might trade in the heart symbol for a centipede. Right now the program is an isolated block, but you can access these new subroutines by adjusting the main logic unit.

```
5  REM *** CENTIPEDE ***
10 INPUT A
20 PRINT "♥" : REM *** CLEAR SCREEN
30 FOR P = 1 TO 20
40 FL = INT (((P/2) - INT (P/2))*2) + .5)
50 SN = 1 - 2 * FL
60 FOR J = 0 TO 39
70 CH = (40*P) + (39*FL) + (SN*J)
80 POKE 1024 + CH,81
90 POKE 55296 + CH,1
100 IF J >= A THEN POKE 1024 + (40*P) + ((J - A)*SN) +
    (39*FL),32
110 IF J < A THEN POKE 1024 + (40*(P - 1)) + ((A - J)*SN) +
    (39*FL) - SN,32
120 FOR T = 1 TO 10 : NEXT
130 NEXT : NEXT
```

Tired of playing catatonic to the computer's dogmatic?
Try...

9 Number Theory I

In this chapter we put the computer to work doing the kind of tasks it does best: repeated performance of simple procedures. Ever wonder what the sum of the first 1000 odd numbers was? Probably not. But the ideas behind the solution to such a problem are central to many interesting programs. Let's go ahead and see what we can do with the first 1000 odd numbers.

One of the simplest programs for adding up those numbers is:

```
10  FOR Q = 1 TO 1999 STEP2
20  LET S = S+Q
30  NEXT Q
40  PRINT "THE SUM IS ";S
```

When you RUN this program you will get 1,000,000. This answer seems suspicious. Did we really get the first 1000 odd numbers? How can we be sure that there weren't 999 by mistake? To check this we can put tracers into the program as we did in the coin flip program in Chapter 6. Add the following lines:

```
25  COP = COP + 1
27  PRINT S,Q,COP
35  PRINT "THE COUNTER IS AT ";COP
```

The variable COP enters the FOR/NEXT loop with a value of 0 (another example of initialization). In the first pass through the loop, COP gets incremented to one: that's what we want because this is pass #1.

Line 27 gives us an interim status report on our variables. This will

slow our computation because the computer has to stop and print numbers on the screen. But seeing our values helps make the program more transparent. (Reminder: the two commas separating the three variables tell the computer to separate what it is printing.)

When you RUN the program now, you will see three lines of numbers scrolling up your screen. The righthand column is the number of loops so far; the center contains the current odd number; and the left column contains our running sum. After a while the program halts and the bottom of our three columns look like this.

```
996004 1995  998
998001 1997  999
1000000 1999 1000
THE COUNTER IS AT 1000
THE SUM IS 1000000
```

Now we can have more confidence in our answer because the counter shows that we have indeed added 1000 numbers. Let's check further to see if there are any hidden logical errors in our program. A 1000-pass loop is somewhat unmanageable, so let's make the program smaller. Change Line 10 to:

```
10  FOR Q = 1 TO 9 STEP 2
```

All the results now fit on the screen and should look like this:

```
1  1  1
4  3  2
9  5  3
16 7  4
25 9  5
THE COUNTER IS AT 5
THE SUM IS 25
```

The third column indicates that we have passed through the loop five times. The second is the list of numbers that we want to add: it does indeed contain the first five odd numbers. And there's no arguing that the sum of these numbers is 25. The first column gives a running subtotal, and ends where it should with 25. The case is now pretty strong that our answer of 1,000,000 for the first 1000 odd numbers is correct.

Let's use this program one more time to add up the first ~~500~~ odd numbers. (Note that this is not the same as the sum of odd numbers from 1-~~500~~.) Change Line 10 again so it looks like:

```
10  FOR Q = 1 TO 999 STEP 2
```

When you RUN the program, time it with a watch or clock. We will then delete Line 27 and see how much time that PRINT statement is using. What's the answer?

```
THE COUNTER IS AT 500
```

```
THE SUM IS 250000
```

Hmmm...adding up half as many numbers gives one-quarter the sum. The computation took about ~~40~~ seconds. Erase Line 27 by typing 27 on any empty line and hitting [RETURN]. Now get out your timepiece again: pretty impressive improvement!

EVEN STEVEN

Let's try something else. What is the sum of the first ~~1000~~ even numbers? A very simple modification of our program can handle this. For this run, we'll let the computer keep time with its internal clock.

But we must decide: is zero or two the first even number? For the moment we'll say that two is the first. Since this program takes so long, we'll work with the first ~~500~~ even numbers:

```
5   SC = TI
10  FOR Q = 2 TO 1000
20  S = S + Q
25  COP = COP + 1
27  PRINT S,Q,COP
30  NEXT Q
35  LET SC = (TI - SC)/60
40  PRINT "THE COUNTER IS AT ";COP
50  PRINT "THE SUM IS ";S
60  PRINT "COMPUTATION TIME: ";SC;" SECONDS"
```

The variable TI is a reserved variable in BASIC that always contains the current value of the clock. This variable increments at 1/~~60~~-second

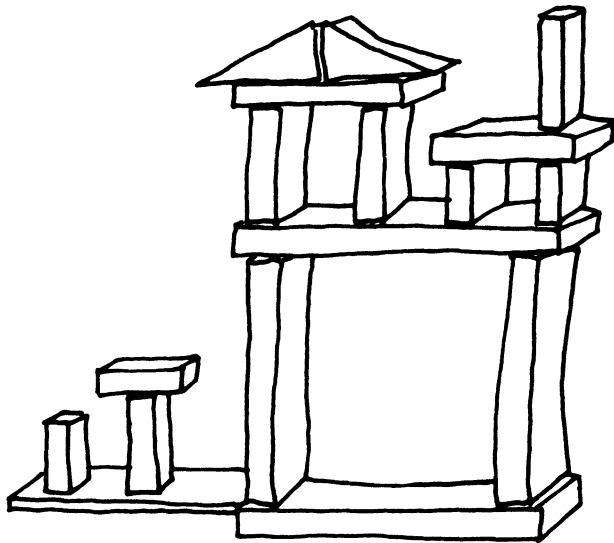
intervals. PRINTing the current value of TI tells you how long your computer has been turned on. Find an empty line and type:

PRINT TI/3600

The number appearing on the next line is the number of minutes your computer has been on.

RUN your program. My computer took 36.533333 seconds (or so) to add the first 500 even numbers. What happens when we erase Line 27? This time my computer took 3.71666673 seconds: a tenfold improvement.

Amend Line 10 to find the sum of $2 + 4 + 6 + 8 + 10 + 12 + 14 + \dots + 1000$.



A single note well played is a thing of beauty...

10 Sound and Music

Part I

AN INTRODUCTION TO SOUND CONTROLS

Among its many features, your Commodore 64 boasts a synthesizer on a chip. The chip is called SID, or Sound Interface Device, and with it, you can create sound and music.

The SID chip has three oscillators, which produce a full eight-octave range of sounds. These oscillators can also produce four separate and distinct waveforms that you can use to vary the “color” of your voices. You can also adjust the **Attack**, **Decay**, **Sustain**, and **Release** (ADSR) of each note by programming what is called the SID’s Envelope Generator.

For the impatient: try the final program in this chapter – a three voice rendition of “Swing Low, Sweet Chariot.”

SOUND SAMPLER

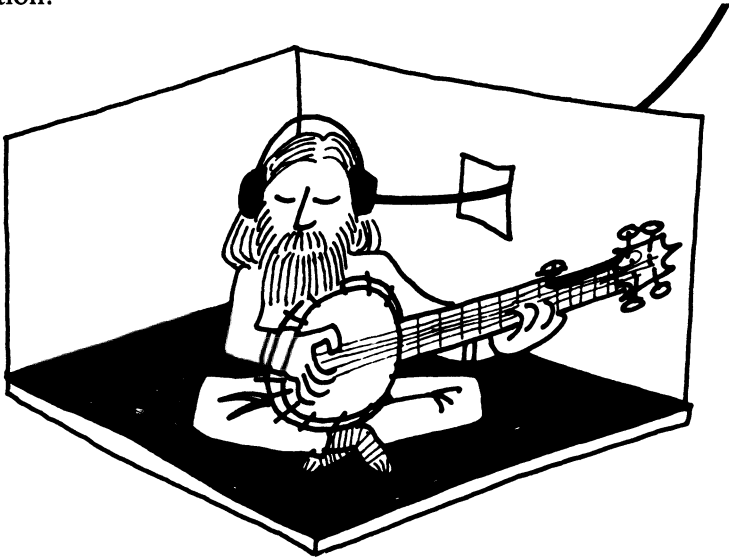
Let’s begin with a program that allows you to create a musical note by pressing any letter or number on the keyboard. Type in the program, RUN it, then press any letter or number on your keyboard and see what happens.

One important note before you begin: to exit this program, press the [RUN/STOP] key. Otherwise, you will continue to get notes every time you touch a key.

```
5  REM *** SID's Single Solo Sound Sampler ***
10  FOR M = 54272 TO 54296 : POKE M,O : NEXT
15  GET A$ : IF A$ = "" THEN 15
20  POKE 54296,15
30  POKE 54277,9 : POKE 54278,64
40  POKE 54273,28 : POKE 54272,49
50  POKE 54276,17
60  FOR DR = 1 TO 350 : NEXT
70  POKE 54276,16
75  GOTO 15
```

BETWEEN THE LINES

The following explanation of the “Solo Sound Sampler” will introduce you step-by-step to programming sound. Feel free at any time to enter your own values in the program to see how they affect the sound. Don’t forget to press [RETURN] when you have entered or edited your new value. By the way, a good time-saving trick is to type RUN at the top of the screen. Then you can simply press [CLR/HOME], bringing the cursor HOME after entering your new value, and [RETURN] to RUN your variation.



Clear the Chip

Line 10. As we have seen, memory always holds a lot of leftover information so it's a good idea to start each sound program by clearing the SID chip memory. SID is organized much like the VIC II in that a section of RAM addresses (54272-54296 inclusive), serves as a SID switchboard. The statement:

```
10 FOR M = 54272 to 54296 : POKE M,0 : NEXT
```

POKEs a zero into each SID switchboard address register, thereby clearing the switchboard.

The GET Statement

Line 15. We used this command in *Cursory Cursor*. The line:

```
15 GET A$ : IF A$ = "" THEN 15
```

instructs the computer to GET the last key pressed, if any; if none, keep trying.

Setting Volume Controls

Line 20 sets the volume at 15. You have a choice of 0-15 volume settings; 15 is the loudest and most frequently used. Numbers larger than 15 influence certain filters that can be used in more advanced programming. For now, however, stick with 15. Also, note that one volume setting governs all three voices.

Setting ADSR Controls

Line 30 sets the ADSR, or **Attack, Decay, Sustain, Release**. POKE 54277,9 controls the note's attack and decay; POKE 54278,64 controls its sustain and release. These two settings affect the coloration of your sound. They control the loudness of the sound at different points during its production, forming what is called an **envelope**—hence the name “Envelope Generator.”

The Attack is the beginning of a sound; it controls how rapidly the sound rises to peak volume. The Decay controls the time it takes for the volume to come down to the middle level, which will be defined and then sustained by your sustain setting. Finally, the sound will decay down to nothing; the time this takes is the release time. Drums, for example, have an explosive attack and a rapid decay with little sustain. Bowed violins, on the other hand, can have a slow, steady attack, sustain over a long period, and then fade away with a slow release.

You will notice that one number controls both the Attack and Decay, while another number controls the Sustain and Release. That's because they are additive.

Attack/Decay Control Numbers

Each voice has a POKE number for its Attack/Decay control setting: Voice 1, POKE 54277; Voice 2, POKE 54284; Voice 3, POKE 54291. Following the POKE number is a comma, and then a number between 0 and 255 that encodes the combined Attack and Decay. The control is built around the numbers 1, 2, 4, 8, 16, 32, 64, and 128, the decimal equivalents of the bits in the voices' address register. The first four numbers control the Decay; the last four numbers, the Attack.

You can add any, all, or none of your Attack numbers together for your attack range, i.e., 0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240; 240 is the maximum Attack setting. You can also add any, all or none of your Decay numbers; thus, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 are all possible Decay settings. To encode your final setting, add the Attack number to the Decay and enter the sum; the computer will do the rest.

Here are some examples: the command POKE 54277,73 means set the Attack at 64 and the Decay at 9 for Voice 1. POKE 54284,63 means set Attack at 48, Decay at 15 for Voice 2. In our sample program, we used POKE 54277,9, which sets the Attack at 0 and the Decay at 9 for Voice 1.

Sustain/Release Control Numbers

The Sustain/Release setting functions much the same as the Attack/Decay. The higher numbers—128, 64, 32, 16—control the Sustain, and the lower numbers—8, 4, 2, 1—control the Release. The same rules hold for entering your setting. Thus, POKE 54278,137 means set Sustain at 128 and release at 9 for Voice 1.

Creating An Envelope

Because the Attack/Decay settings interact with those for Sustain/Release, finding the right combinations can be tricky. Why not try a few in your program? Try to be systematic. For example, try the maximum Attack of 240; then try 128, then 16, then 0. Surprised? As you see, the maximum Attack does not necessarily mean maximum sound. Now try the Decay settings 15, 9, 4, 0. After that, try a few combinations such as 240 and 15, or 128 and 9. As you can hear, they are all different.

Now, set your Attack/Decay back to 64 and test your Sustain/Release setting. Try 240, 128, 64, 16, 15, 9, 4, 0; then try combination settings such as 255, 68 or 20. As you can hear, the Sustain number has a great deal of influence over how loud and long the middle part of your note sounds. If you use 128, the note tends to stay loud much longer than when you use 16, even though the overall duration of the note is the same.

Now for the big surprise: start combining your Attack/Decay and Sustain/Release settings. Once again, try to be systematic. Try 240 A/D with 240 S/R, then keep the 240 A/D and try 128 S/R, etc. For example: set A/D at 9, S/R at 0, 16, 128, and 240. Next, set your S/R at 0 and try A/D at 9, 89, and 137.

Setting Pitch Controls

Now that you have a handle on the ADSR, let's change the pitch, or frequency of vibration. This is done in the next line of SID's Single Solo Sound Sampler.

Line 40 has two POKE statements. The first, POKE 54273, sets the **high bit frequency** number of Voice 1, while the second, POKE 54272, sets the **low bit frequency** for Voice 1. Each pitch requires a high frequency number and a low frequency number, which are listed in the Table of Note Values in the Appendix. Right now, the frequency is set for A-440, which is the A in the 4th octave. If you wanted to play the A in the 6th octave, you would look up A-6 in your Table of Note Values and see that the high setting is 112 and the low setting is 199. Try entering these new values in Line 40.

The SID has an eight-octave range. Try a low note such as D-2. Go to the Appendix again: the high number is 4, the low is 180. Enter these new values. It's that simple.

If you wanted to program the pitch for Voice 2 or 3, you would follow the same procedure, using different POKE numbers. For Voice 2, the high bit frequency number is preceded by POKE 54280, and the low number is preceded by POKE 54279. For Voice 3, you would use POKE 54287 for the high and POKE 54286 for the low.

SETTING WAVEFORM CONTROLS

Now let's really change the sound by altering the values in Line 50.

Triangle Waveform

Line 50 sets the waveform. The SID has four waveforms. The first is called the "triangle" because of its shape. You already know how it

sounds—mellow and organ-like—because it's the one we've been using up until now. You turn on the triangle waveform by POKEing 54276 with the number 17. You turn it off by POKEing 54276,16. Waveform settings are the same for all voices. In other words, a triangle waveform is always 17—only the POKE number changes. So POKE 54283,17 turns on the triangle waveform for Voice 2 and POKE 54290,17 sets the triangle for Voice 3.

White Noise Waveform

Let's dramatically change the sound by using the "white noise" waveform. To do this, just go to SID's Sampler, and change the 17 in Line 50 to 129. Also change Line 70 so that the 16 is now 128. Surprised? You would use this waveform for many videogame sound effects. You can make explosions, crash noises, rattles, shots, and hisses. Try different ADSR settings along with this white noise waveform. Now try pressing a key many times in quick succession. Don't be afraid to experiment; who knows what you may discover?

Sawtooth Waveform

Let's return to a more musical electronic sound with the "sawtooth" waveform. Just go to your program and change the 129 on Line 50 to 33. Also change Line 70 so that the 128 reads 32. As you can hear, the sawtooth waveform has a kind of buzzing quality. Try it with a few different ADSR settings: A/D 9, S/R 0; A/D 25, S/R 9. They both have a "harpsichord" sound. Now try A/D 137, S/R 128.

Pulse Waveform

The final waveform available to you is the "pulse." It is a bit more difficult to use and requires two extra POKE statements: you must POKE in a high pulse number and a low pulse number. Listen to what happens if you forget by changing Line 50 from POKE 54276,33 to POKE 54276,65. Also edit Line 70 so that the 32 now reads 64. When you RUN this version, you should get a low, buzzing and thunking. Now add this line to your program:

```
45 POKE 54275,8 : POKE 54274,255
```

See what a difference it makes?

Try experimenting with high pulse and low pulse numbers. The one rule you must follow is that the high pulse number must be between 0 and 15 inclusive; the low pulse can be between 0 and 255. The same rule applies when using Voices 2 and 3. Once again, only the POKE

number will change; the settings are identical. For example, POKE 54283,65 to turn on the pulse waveform for Voice 2; then POKE 54282,8 for the high pulse number, and POKE 54281,255 for the low pulse number. For Voice 3, POKE 54290,65 to turn on the pulse; POKE 54282,8 for the high pulse number and POKE 54281,255 for the low pulse number.

Now go back to the original triangle waveform. (Don't forget to remove Line 45.)

One final note: always set your ADSR *before* turning on the waveform control. That is essential because the ADSR cycle is triggered by the waveform gate that opens when you turn on a note.

Controlling Duration

We have seen how a note is built electronically by setting its volume, envelope, and waveform. Our final control in this program is the *duration*—how long the note lasts.

Line 60 is a simple time loop. It tells the computer how long to hold the note before going on to the next statement. If you want the note to last for a shorter period, just decrease the time loop. Go to your program and change Line 60, for example, from FOR DR = 1 TO 350 : NEXT to FOR DR = 1 TO 50 : NEXT. If you want to make the note last longer, try FOR DR = 1 TO 750 : NEXT.

Ending Notes By Closing The Gate

Line 70 turns off the note by closing the waveform gate. To do this, we POKE in the waveform control for Voice 1, setting it at a number one less than the starting waveform—e.g., 16 to turn off 17, 32 to turn off 33, 64 to turn off 65, and 128 to turn off 129. The closing of the waveform gate triggers the release cycles of the ADSR.

Line 75 sends the program back to Line 15. You may notice a residual noise after each note. This can be avoided by going back to Line 10 instead of Line 15. This will clear the chip. Unfortunately, it will also cause a clicking noise. The choice is yours.

Multiple Voice Samplers

Now that you have control of the sound, try changing SID's Single Solo Sound Sampler into SID's Double Duo Sound Sampler by adding:

```
22 POKE 54284,9 : POKE 54285,0
```

```
32 POKE 54280,18 : POKE 54279,209
```

42 POKE 54283,33

72 POKE 54283,32

Or how about adding these lines for SID's Triple Trio Sound Sampler:

23 POKE 54291,125 : POKE 54292,64

33 POKE 54287,50 : POKE 54286,60

43 POKE 54290,17

73 POKE 54290,16

You can now experiment with chords and multiple-voice sound effects. Try having all the notes play the same pitch, but with different waveforms and envelopes.

Part II

AN INTRODUCTION TO PROGRAMMING MUSIC

With all the puzzle pieces in hand, let's program a scale. This program is going to be a bit different. Up to this point you've been playing *real time* notes: press a key and get the note. Most computer music, however, is not in real time. Our GET command in Line 15 created real-time sound for us. In our new program, we will use a different approach, the READ/DATA commands.

Before typing this program, you might want to save SID's Single Solo Sound Sampler with SAVE "SSS" if you have a Datassette, or SAVE "SSS",8 if you are using a disk.

TONING THE WHOLE

```
5  REM *** SID's Whole Tones ***
10  FOR M = 54272 TO 54296 : POKE M,0 : NEXT
20  POKE 54296,15
30  POKE 54277,9 : POKE 54278,0
40  READ H,L,DR
50  IF H<0 THEN 900
60  POKE 54273,H : POKE 54272,L
70  POKE 54276,17
80  FOR T = 1 TO DR : NEXT
90  POKE 54276,16
```

```
100  GOTO 40
200  DATA 7,12,200,7,233,300,8,225,300,9,247,250,11,48,250,12,
      143,250,14,24,200
210  DATA 15,210,200,17,195,200,19,239,150,22,96,150,25,30,150,
      28,49,100
220  DATA 31,165,200,35,134,100,39,223,50,44,193,50,50,60,50,56,
      99,1750
230  DATA -1,-1,-1
900  POKE 54276,0
```

Here is a line-by-line explanation of SID's Whole Tones.

Line 5 is a REM statement for the title.

Line 10 clears the SID Chip.

Line 20 sets the volume at maximum.

Line 30 sets the Attack/Decay at 9, the Sustain/Release at 0.

Line 40 is a READ statement. It tells the computer to read the first three elements stored in the DATA bank and names them H, L, and DR. Each time the computer passes the READ statement (see Line 100), it READs in the next three DATA values. A pointer keeps track of which value comes next.

Line 50 is an IF/THEN statement that tells the computer to go to Line 900 if the value of H is less than 0. If you look at DATA Line 230, you will see -1, -1, -1. This line is called a flag. Since -1 is less than 0, when the computer READs this line it will execute Line 900. It's good idea to store as many elements in your FLAG as called for in your READ statement—three, in this case.

Line 60 POKes in the high frequency number for Voice 1 and assigns it the value of H. It then POKes in the low frequency number with the value L, the next variable in DATA.

Line 70 turns on the triangle waveform for Voice 1 and starts the note.

Line 80 is a timing loop for each note. It means that the duration of each note will equal 1 to DR. DR will be READ as the third element in DATA.

Line 90 turns off the note or waveform in Voice 1.

Line 100 says go back to Line 40 and run through the program again. This time, however, the READ statement will take the next three variables in DATA.

Lines 200-230 encode the high frequency number, low frequency

number and duration for each note. When entering DATA, always remember to type the word DATA after the line number. Place a comma after each variable but *do not* place a comma after the last variable in any given line; just hit [RETURN]. Always carry your numbers out 40 columns on your screen when entering long DATA statements; your Commodore 64 will automatically advance the numbers down to the next line on the screen. It's OK if a number starts on one line and finishes on another. Finally, *never* exceed two display lines (80 columns) of numbers for any single DATA grouping; your computer can handle only 80 columns at a time.

Line 900 turns off the sound.

If you want to play the tones over again, just add these two lines:

```
910 RESTORE
```

```
920 GOTO 40
```

Line 910 resets the memory pointer back to the first number in your DATA statements, thus re-storing your DATA.

Line 920 sends the computer back to reREAD the DATA and play it again.

To stop those whole tones, hit [RUN/STOP].

Now that you have the format of this program, you can use it to compose other songs; all you have to do is change the DATA. For example, here's a familiar song, "Oh, Susannah," played on our synthesized banjo.

OH, SUSANNAH!

```
5 REM *** OH SUSANNAH BY STEPHEN FOSTER ***
10 FOR M = 54272 TO 54296 : POKE M,0 : NEXT
20 POKE 54296,15
30 POKE 54277,7 : POKE 54278,5
40 READ H,L,DR
50 IF H < 0 THEN 900
60 POKE 54273,H : POKE 54272,L
70 POKE 54276,33
80 FOR T = 1 TO DR : NEXT
90 POKE 54276,32
```

```

95  FOR B = 1 TO 34 : NEXT
100  GOTO 40
200  DATA 18,209,75,21,31,75,23,181,150,28,49,150,28,49,225,31,
    165,75
210  DATA 28,49,150,23,181,150,18,209,225,21,31,75,23,181,150,
    23,181,150
220  DATA 21,31,150,18,209,150,21,31,450,18,209,75,21,31,75,23,
    181,150
230  DATA 28,49,150,28,49,225,31,165,75,28,49,150,23,181,150,18,
    209,225
240  DATA 21,31,75,23,181,150,23,181,150,21,31,150,18,209,900
250  DATA -1, -1, -1
900  POKE 54276,0

```

You can speed up the beat by adjusting Line 80:

```

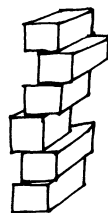
80  FOR T = 1 TO DR/2 : NEXT

```

Line 95 is a second time loop inserted between notes to make the banjo articulation crisper.

Now that you've sampled what the SID can do, you're ready for "SID's Song," which combines sound effects, multiple voices, and color graphics.

But first, here are a few suggestions to save typing time. You can abbreviate the word POKE by typing P then [SHIFT]:[O] (see Appendix D in the *User's Guide*). The shifted O will produce a graphics symbol, but when you LIST your program, the computer will print POKE. You can shorten the POKE numbers by entering S = 54272 at the beginning of your program; then all you have to do is enter POKE S + a number.



Thus, POKE S + 24 means POKE 54296, while POKE S + 5 means POKE 54277. (You don't have to use the letter S—we just use it to stand for SID.)

```
5  REM *** SID'S SONG BY J.C.VOGEL ***
10  S = 54272
20  FOR M = 0 TO 24 : POKE S+M,0 : NEXT
30  POKE S+0,240 : POKE S+1,33
40  POKE S+5,9 : POKE S+6,0
50  POKE S+24,15
60  FOR B = 1 TO 2
70  POKE S+4,129
80  FOR X = 1 TO 500 : NEXT : POKE S+4,128
90  FOR X = 1 TO 30 : NEXT : NEXT
110 POKE S+5,9 : POKE S+6,0
120 POKE S+12,102 : POKE S+13,0
140 READ H1,L1,H2,L2,BR,BG,DR
150 IF H1 < 0 THEN 910
160 POKE S+1,H1 : POKE S+0,L1 : POKE S+8,H2 : POKE
    S+7,L2
165 POKE 53280,BR : POKE 53281,BG
170 POKE S+4,17 : POKE S+11,17
180 FOR T = 1 TO DR : NEXT
190 POKE S+4,16 : POKE S+11,16
210 GOTO 140
300 DATA 56,99,25,30,0,15,200,50,60,37,162,0,14,200,75,69,18,
    209,1,14,200
310 DATA 63,75,29,223,1,13,200,59,190,50,60,2,13,200,59,190,22,
    96,2,12,200
320 DATA 63,75,28,49,3,12,200,75,69,28,49,3,11,200,50,60,22,96,
    4,11,200
```

```
330 DATA 56,99,50,60,4,10,200,56,99,29,223,5,10,200,50,60,18,
    209,5,9,200
340 DATA 75,69,37,162,6,9,200,63,75,25,30,6,8,200,59,90,25,30,
    7,8,200
350 DATA 59,190,37,162,7,7,200,63,75,18,209,15,0,200,75,69,29,
    223,14,0,200
360 DATA 50,60,50,60,14,1,200,56,99,22,96,13,1,200,56,99,28,49,
    13,2,200
370 DATA 50,60,28,49,12,2,200,75,69,22,96,12,3,200,63,75,50,60,
    11,3,200
380 DATA 59,190,29,223,11,4,200,59,190,18,209,10,4,200,63,75,37,
    162,10,5,200
390 DATA 75,69,25,30,9,5,200,50,60,25,30,9,6,200,56,99,37,162,
    8,6,200
400 DATA 56,99,18,209,8,7,200,50,60,29,223,7,7,200,75,69,50,60,
    7,6,200
410 DATA 63,75,29,96,5,6,200,59,190,28,49,5,5,200
900 DATA -1, -1, -1, -1, -1, -1, -1, -1
910 RESTORE
920 FOR T = 1 TO 250 : NEXT
930 GOTO 30
```

Now let's examine the program.

Line 10 assigns 54272 to the letter S.

Line 20 clears the sound chip.

Line 30 to Line 100 creates the sound effect twice.

Line 30 sets the frequency.

Line 40 sets the ADSR.

Line 50 sets the volume.

Line 60 sets up the repetition.

Line 70 turns on the white noise waveform.

Line 80 controls the duration of the effect, then turns off the waveform.

Line 90 tells the computer to wait and then do it a second time.

Lines 110-210 set up the same song format as “Oh, Susannah,” only this time we added a second voice and the POKE statements controlling border and background color.

Line 110 sets the ADSR for Voice 1.

Line 120 sets the ADSR for Voice 2.

Line 140 tells the computer to read the first seven numbers in the DATA bank and assign them variable names: H1 (high frequency 1), L1 (low frequency 1), H2 (high frequency 2), L2 (low frequency 2), BR (border), BG (background), DR (duration).

Line 150 instructs the computer to go to Line 910 if H1 is less than 0.

Line 160 POKES the high and low frequency numbers for Voices 1 and 2.

Line 165 POKES values for screen border and background colors.

Line 170 sets the waveform for Voices 1 and 2.

Line 210 tells the computer to go back to Line 140 and READ in the next note color and duration.

Lines 300-410 are DATA numbers for H1, L1, H2, L2, BR, BG, D (each note’s color).

Line 900 is part of the DATA bank, the flag to break out of the READ/DATA loop.

Line 910 restores the DATA.

Line 920 tells the computer to wait.

Line 930 tells the computer to go back to Line 30 and play again.

To break out of this program, you must hit the [RUN/STOP] key.

If you want to have a little fun with this program, try hitting [SHIFT]:[CLR/HOME]. Now, press [CTRL]:[9] and the space bar, then [CTRL] and a number between 1 and 8. Scatter a few color bars around the screen. Now, re-RUN the program: different colors will appear and disappear during the song.

Playing music on a computer is like playing music on an instrument: it takes time and patience to learn. As your programming skills grow, you will find more and more ways of shaping sounds. Experimenting with new ideas is the secret of success.

NICK’S FAVORITE

Here’s a program for all three of the SID voices – “Swing Low Sweet Chariot” in three-part harmony. In it we’ve used the jiffy clock for keeping time instead of a FOR/NEXT time loop. Each DATA line in this pro-

gram encodes the duration as well as the high and low frequency numbers for each voice.

```
1  PRINT "♥"  
5  PRINT "SWING LOW SWEET CHARIOT (TRAD)"  
10 PRINT "MUSIC ARRANGED BY JIM VOGEL"  
100 M = 54272  
110 POKE M+24,15  
120 POKE M+5,9 : POKE M+6,0  
130 POKE M+12,26 : POKE M+13,36  
140 POKE M+19,24 : POKE M+20,202  
150 T = TI  
160 POKE M+4,32 : POKE M+11,32 : POKE M+18,16  
170 READ X : IF X<0 THEN 690  
180 READ H1,L1,H2,L2,H3,L3  
190 POKE M+1,H1 : POKE M,L1 : POKE M+4,33  
200 POKE M+8,H2 : POKE M+7,L2 : POKE M+11,33  
210 POKE M+15,H3 : POKE M+14,L3 : POKE M+18,17  
220 T = T+X  
230 IF T>TI GOTO 230  
240 GOTO 160  
300 DATA 30,28,49,33,135,5,152  
310 DATA 60,22,96,14,24,0,0  
320 DATA 30,28,49,33,135,5,152  
330 DATA 45,22,96,29,223,9,104  
340 DATA 15,22,96,0,0,0,0  
350 DATA 15,18,209,29,223,0,0  
360 DATA 45,16,195,10,143,6,71  
370 DATA 15,22,96,16,195,7,12  
380 DATA 15,22,96,11,48,0,0  
390 DATA 15,22,96,14,24,9,104
```

400 DATA 15,22,96,11,48,0,0
410 DATA 15,28,49,21,31,8,97
420 DATA 15,33,135,0,0,0,0
430 DATA 30,33,135,21,31,7,12
440 DATA 30,33,135,22,96,8,97
450 DATA 30,0,0,0,0,18,209
460 DATA 30,0,0,21,31,16,195
490 DATA 30,0,0,0,0,14,239
500 DATA 15,37,162,22,96,14,24
510 DATA 15,33,135,0,0,14,24
520 DATA 60,28,49,0,0,16,195
530 DATA 30,33,135,22,96,9,247
540 DATA 45,22,96,14,239,9,104
550 DATA 15,22,96,14,239,9,104
560 DATA 15,18,209,0,0,0,0
570 DATA 45,16,195,14,239,10,143
580 DATA 15,22,96,16,195,11,48
590 DATA 15,22,96,0,0,0,0
600 DATA 15,22,96,18,209,9,104
610 DATA 15,22,96,0,0,0,0
620 DATA 15,28,49,18,209,7,119
630 DATA 15,28,49,0,0,0,0
640 DATA 30,25,30,14,239,8,97
650 DATA 30,22,96,14,24,11,48
660 DATA 30,0,0,0,0,8,97
670 DATA 30,0,0,0,0,5,152
675 DATA -1
680 DATA -1
690 C = C+1 : IF C = 2 THEN 1000
700 RESTORE

```
710  FOR J = 1 TO 200 : NEXT
720  GOTO 150
1000  FOR Z = M TO 54296 : POKE Z,0 : NEXT Z
```

Here's a line-by-line description.

Line 1 clears the screen.

Line 5 prints the title.

Line 10 prints the arranger's name.

Line 100 sets M at 54272.

Line 110 sets master volume at 15

Line 120 sets ADSR for Voice I.

Line 130 sets ADSR for Voice II.

Line 140 sets ADSR for Voice III.

Line 150 sets the variable T equal to the value of the jiffy clock.

Line 160 turns off the waveform for Voices I, II, and III.

Line 170 READs the first piece of information in the DATA, which is given the variable name X and represents the duration number. If the value of X is less than 0 then go to Line 690.

Line 180 READs the next 6 pieces of DATA and assigns them the variable names—1, L1, H2, L2, H3, L3, which represent the high and low frequency numbers for Voices I, II and III.

Line 190 POKES the high and low frequency number and turns on the waveform for Voice I.

Line 200 performs the same function as Line 190 but for Voice II.

Line 210 performs the same function as Line 190 but for Voice III.

Line 220 sets up the duration by adding the duration value, read as X from the DATA, to the variable T. T was equal to the jiffy clock but is now greater by the value we have just added.

Line 230 holds the note on for the duration of time it takes the jiffy clock to catch up to the new value of T.

Line 240 sends the computer back to Line 160 where it turns off the note and begins the whole cycle over again.

Lines 300-670 contain the DATA encoded as the duration numbers and the high and low frequency numbers for Voices I, II, and III.

Line 700 restores the DATA.

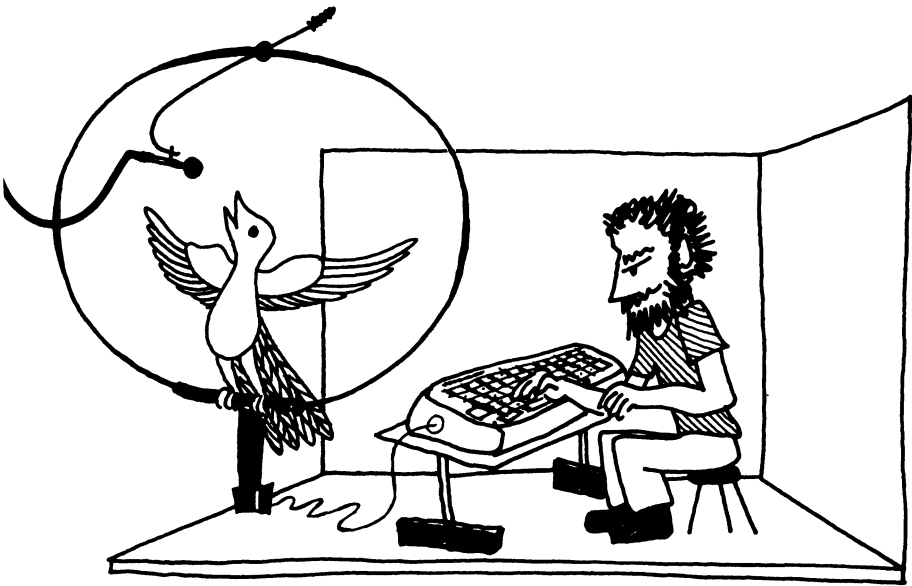
Line 710 is a time loop that puts a space between the first and second times through the song.

Line 720 returns the computer to Line 150 where it resets the variable T equal to the jiffy clock. Then the whole program begins its second pass.

Line 1000 turns off the sound chip.

There's a whole world of sound possibilities locked in the SID. If you'd like to try more sound effects, skip ahead to Chapter 15, "Tricks of the SID."

If you'd like to learn more about music programming and the electronic music capabilities of your 64, try *The Commodore Music Book* by Jim Vogel and Nevin Scrimshaw.



Pale jale shadows flashed red as the spacecraft lifted off...

11 Real-Time Graphics

We are now ready to learn how to move sprites around the screen. We will use two sprites in a program that can form the core of all kinds of adventure programs. Since our task has many components, we'll have to be organized.

The first step is deciding what we want. How about a rocket that rises off the ground and disappears with a deafening roar?

We'll begin with the task of moving sprites in block shapes. Later we can adapt and experiment until we get our spaceship. Here goes:

```
5  REM *** SPACETREE ***
7  REM *****
10 POKE 53280,11 : POKE 53281,0
20 FOR M = 54272 TO 54296 : POKE M,0 : NEXT
30 PRINT"☐" : REM *** CLEAR SCREEN
```

These first few lines present a typical problem: so many POKE commands make it hard to keep track of what does what. You'll need some careful bookkeeping to keep track of all those switchboard addresses — keep a scratch pad handy and use REM statements liberally.

Line 5 displays the name of the our future adventure.

Line 10 sets the screen and border color.

Line 20 clears the SID chip's switchboard.

Line 30 clears the screen using [SHIFT]:[CLR/HOME].

The next group of program lines sets two constants for later use. M is a sprite shape pointer and V is the starting address for the sprite switchboard. Lines 40 and 50 set some of the sound addresses. For practice

you should check the use of these settings in the SID Chip Control chart (see Appendix).

```
40 POKE 54272,64 : POKE 54273,33
50 POKE 54277,64 : POKE 54278,128
60 V = 53248 : M = 12288 : X = 125
```

We won't be using sound right away; these lines are inserted as a future convenience. When we have the whole program running you can come back and change the POKE values to get different sound effects.

TURNING ON SPRITES

We need two sprites for this job. That means doing some research into the sprite portion of VIC II's video memory switchboard. First we need to decide which of the eight sprites, numbered 0 to 7, to use. When they move, Sprite 0 always passes in front of Sprite 2, Sprite 2 always in front of Sprite 3, and so on; collisions never happen. Let's use Sprites 4 and 5; that way, if we later want to add a character that could pass in front or in back of our spaceship, we have a sprite available for the job.

To turn these sprites on, we have to POKE the right value in address register $V + 21$. This single byte, with its internal code of eight bits—one for each sprite—controls which of the sprites is on and off at any given moment. Remember learning to calculate the decimal values of bytes in Chapter 7? Finding the right POKE value for turning sprites on and off requires calculating the decimal value of the byte in address register $V + 21$. Imagine the register looking like this:

| Sprite # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------|-----|----|---------|----|---|---|---|------|
| 0 = off | | | | | | | | |
| 1 = on | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| box value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| | | | 32 + 16 | | | | | = 48 |

To find the POKE number that will turn on Sprites 4 and 5 and leave the others off, add up the decimal values of the "one's" in the boxes corresponding to Sprites 4 and 5. Then POKE that sum into address $V + 21$. In this case we want $32 + 16$ which equals 48, so the next line is:

```
70 POKE V + 21,48
```

If you check the sprite switchboard chart in the Appendix, you will note that the sprite color registers of 4 and 5 are $V + 43$ and $V + 44$. Let's choose a color code from the color chart on the next page. Say you choose 2. Then the following two POKE lines will make our sprites red.

```
80 POKE V + 43,2 : POKE V + 44,2
```

The following line is temporary. It POKes into memory the code for the shape of our sprites. The variable M takes on the values of those memory addresses where the shape of our two sprites is stored.

```
90 FOR Q = M TO M + 62 : POKE Q,255 : NEXT
```

We are almost ready for a test. All we have to do is to tell our sprites where to go on the screen. These lines POKE in the X and Y coordinates for both sprites.

```
99 REM *** MAIN LOGIC UNIT ***
```

```
100 POKE V + 8,75 : POKE V + 9,75
```

```
110 POKE V + 10,125 : POKE V + 11,125
```

That should do it for this pass. Type RUN and hit [RETURN]. Any bugs? If you haven't made any syntax errors, your screen will show a rather pale stationary sprite and also a flickering one. If you did mistype something, go back and hunt it down.

A pale flicker is not what we really want in a sprite, so let's go down the following checklist and see what we forgot.

1. Turn on sprites.
2. Set sprite color.
3. Define sprite shape.
4. Set sprite coordinates.
5. Set memory pointers.

If you go back through the program you'll see that we have inserted program lines for the first four items on the list. But we did not set the memory pointers, so there's more work to do.

MEMORY POINTERS

The memory pointer for any given sprite points the VIC II chip to the right block in RAM to get the shape for that sprite. The pointer value is multiplied by 64 to find the starting place of the block of addresses that the computer looks into to find the shape of your sprite.

We insert a solid square shape into block 192. This block inhabits addresses 12288-12350. (Note that $12288 = 64 \times 192$.) If you look carefully at Line 90 you'll see that we fill the entire block of memory with the value 255.

```
90  FOR Q = M TO M + 62 : POKE Q,255 : NEXT
```

Go back to the sprite chart and look up the registers that set the memory pointers for Sprites 4 and 5; these are addresses 2044 and 2045. The following line will set 192 into each pointer address.

```
75  POKE 2044,192 : POKE 2045,192
```

Let's test our correction. Find an empty line and type RUN, hold your breath...hit [RETURN]. If you have made no further mistakes you will see two red squares on your screen.

LIST your program on the screen. Change the character color to white by pressing [CONTROL]:[2] and LIST again. The sprites have priority over characters so they cover any program lines that try to upstage them. If this gets annoying just press [RUN/STOP:RESTORE], which will clear the video switchboard and return you to the familiar blue screen.

Before we go back and tinker with our program, save it under the name TEMPLATE 1. That way if you crash or wander too far with additions and experiments you can always come back to a clean slate.

Get a clear LIST of the program if you haven't already, and we'll use the screen editor to explore the sprite switchboard.

SWITCHBOARD TRICKS

Line 80 controls the color of our two sprites. (The color codes are in the sprite switchboard chart in the Appendix.)

When you're ready, change the POKE values in Line 80 of the LISTed program to read:

```
80  POKE V + 43,3 : POKE V + 44,7
```

Let's see what this does. Bring the cursor to a free line and reRUN the program. Your sprites have changed color! Experiment with different



“8 bits to byte”

color codes. This might be a good time to adjust the color of your TV set or monitor. Choose your favorite color; you'll want to use it later.

On and Off

Line 70 turned our sprites on; it can be edited to turn them off as well. Use the screen editor to change the 48 on that line to 16, the box number of Sprite 4. (Sprite 4 is the higher of the two sprites on your screen.)

Now when you RUN the program the lower sprite will blink off. If you change the number POKEd in Line 70 from 16 to 32, and RUN the program again, the lower sprite will appear. But the other is now turned off. To get them both back add 16 plus 32 to get 48, and replace the 32 on Line 70 with 48. Hit [RETURN] (as usual) to make your change official, and then test that both your sprites are switched on by RUNNING the program once again.

Controls For Sprite Size And Shape

Let's add a program line to see what vertical expansion is all about. If you look in the switchboard chart, you'll see that there is a single vertical expansion register for all eight sprites. We control this register the same way we handled the register we just played with, $V + 21$ (called the sprite enable register). Add the following line to the program:

```
120 POKE V + 23,16
```

When you RUN the program now, Sprite 4 will suddenly grow. As before, use the screen editor to change the 16 in Line 120 to 32. Now RUN the program; Sprite 5 will be the tall one.

Finally, change that 32 to $32 + 16$ which equals 48 (sound familiar?). Now Line 120 POKes 48 into $V + 23$, which will expand both sprites at once. This line can be used later in a program as an on-and-off toggle switch to expand and contract your sprites.

The following line will put in a toggle for horizontal expansion:

```
130 POKE V + 29,48
```

Experiment until you get a feel for the dimensions and color that the sprites offer.

Moving our sprites involves changing the values in Lines 100 and 110. This is somewhat simpler than what we just did because each sprite has two whole addresses that control location. To make exploring these registers a little easier, let's add a simple input loop. Change Line 100 to read:

```
100 POKE V+8,X : POKE V+9,Y
```

V+8 is the register that holds the horizontal coordinate for Sprite 4; V+9 holds the vertical. (Confirm this in the chart.) Note that V+10 and V+11 control Sprite 5; you'll need that fact later. Meanwhile, add the following lines to the program:

```
199 PRINT "♥" : REM *** CLEAR SCREEN
200 INPUT "X VALUE PLEASE: ";X
210 INPUT "Y VALUE PLEASE: ";Y
220 IF Y = 0 THEN END
230 GOTO 100
```

Now the program will prompt you to input different X and Y values so as to move Sprite 4 around the screen. The sprite can move off the screen to a point but if the numbers get too large you get a curt message saying that you tried to slip in an illegal quantity. Play around with this a bit. The sprite switchboard chart contains a description of the rules of sprite positioning. When you are through, make sure the colors of the sprites are different.

We will return to this program in the last chapter to finish this adventure. Meanwhile, get used to the sprite switchboard. See if you can go through the checklist on page 73 and get Sprite 0 up and running.

Keep using 192 as the memory pointer until we have a chance to explain more about this extremely handy feature. Remember that 192 is the value that points to the block occupying RAM addresses 12288 through 12350.

Meanwhile, here's a glimpse of how we can get our rocket to take off after we have designed it. Type in the lines shown below. They will replace the input loop, so if that version of the program appeals to you, SAVE it now. Make sure that your sprites are different colors so that one sprite can upstage the other.

```
98 FOR TT = 1 TO 54
199 REM *** BLAST OFF ***
200 U = U+1
210 Y = 235 - (INT(1.1↑U)+42)
220 IF Y < 23 THEN POKE V+21,32 : END
230 NEXT TT
```

The surgeon leaned forward. He didn't like what he saw in there. So with quiet determination he excised it with a deft flip of the MID\$. . .

12 Microsurgery

Until now, we've been working with BASIC number variables, POKEing numbers into memory address registers to create video images. In this chapter, we'll consider **string variables** and build a colorful holiday tree out of one particular string.

A string is an ordered collection of characters and/or numbers. This very sentence, including the spaces in between the words, is a string. A string variable such as A\$ stands for a piece of text in the same way that a number variable stands for a number. The dollar sign at the end of the variable symbol tells the computer to expect a string instead of a number.

String variables behave much like number variables. We can even add two strings together, though not quite the same way that we add numbers. Clear the computer with NEW and type in the following example of string addition:

```
10 A$ = "WHY"
20 B$ = "ME?"
30 C$ = A$ + B$
40 PRINT C$
```

When you RUN this program, A\$ and B\$ combine to make a new, larger string C\$. Combining strings in this way is called **concatenation**. But A\$ and B\$ are bunched together. C\$ would look better if we put a space between A\$, "WHY," and B\$, "ME?." We could simply use the screen editor to insert a space after the "y" in "Why," but we will use

another method to show you that blank spaces make perfectly good strings. Add these lines to your program:

```
25  Z$ = "  "
40  C$ = A$ + Z$ + B$
50  PRINT "SPOCK HERE; SPACES ADDED TO C$"
60  PRINT "C$ IS NOW" : C$
```

You can see from Line 60 that we use the same syntax for PRINTing strings as we use for numbers. Go ahead, RUN the program.

If you really want a string of nothing, instead of spaces that look like nothing, you must use...

THE EMPTY STRING

The empty string is the "zero" of string arithmetic. We type it as two sets of quotation marks with no intervening space. The classic use of the empty string is with the GET statement (recall "Cursory Cursor," Chapter 5 and "SID's Single Solo Sound Sampler," Chapter 10). Clear the computer with NEW and enter:

```
50  GET A$ : IF A$ = "" THEN 50
60  PRINT "THE KEY PRESSED WAS ";A$
70  GOTO 50
```

Type RUN, then press any key: the character of that key will appear on the screen (with a few exceptions). Hit [INST/DEL]. The computer not only GETs the key but obeys it as well! The same is true of the other control keys, including the uppercase/lowercase toggle [COM-MODORE];[SHIFT]. To get back the READY prompt, hit [RUN/STOP];[RETURN].

That simple GET loop in Line 50 works like this: if you haven't pressed a key, A\$ is empty, so the IF/THEN statement returns the computer to Line 50 to keep searching for a non-empty string. If you hit a key, the corresponding character is assigned to the string variable A\$.

WARNING... WARNING

There is a crucial difference between the numerical value of a number and the typed character for that number. Consider the following program:

```
10 A$ = "4"
20 B$ = "5"
30 C$ = A$ + B$
```

This program will not add the value four to the value five, but it will concatenate the two symbols "4" and "5," making a new string "45."

MICROSURGERY: MID\$(XX\$,Y,Z)

We will now introduce a peculiar BASIC function that will help us operate on a peculiar string variable to create an odd hexmas tree.

Do you remember using the TAB(X) function back in Chapter 3 to scatter "NOW HERE NOW THERE" across your screen? The MID\$(XX\$,Y,Z) function works much the same way: MID\$ (like TAB) tells the computer *what* to do; the values in parentheses specify how or where to do it. MID\$(XX\$,Y,Z) cuts out a substring of XX\$ starting from position Y and containing the next Z characters. Remember, a space counts as one position. To demonstrate:

```
130 MUD$ = "THE MIDDLE OF"
135 PRINT "MUD$ = ";MUD$
140 M$ = MID$(MUD$,5,6)
150 PRINT MUD$;" MUD$ IS ";M$
```

Line 150 is a little terse, but it should say: "THE MIDDLE OF MUD\$ IS MIDDLE."

ODD FORESTRY

Now for the surgery. The string going under the knife is "13579BDF." No, this is not a license plate number; it is a bite-sized string of all the odd single digits in the base 16 counting system (remember base 10 and base 2 from Chapter 7?). Since people don't have any way of writing more than 10 digits, we use letters for the digits 10-16. Thus, B is base 16 for 11, D is base 16 for 13, and so on. Base 16 numbers are called **hexadecimal** numbers ("hexadecimal" means 16); they convert easily to binary numbers ($2^{14} = 16$) and are important in many programming languages. Hence, our distinguished string consists of odd hex numbers — just the thing for an odd hexmas tree!

```

      *
      B3B
      DBD1F
      33B7D11
      9F3B353FF
      97551D3FF57
      75D9353B7FBFD
      FFDD75BBD71FB1B
      17DF1FB9197913991
      3D3FD3915BB7FD3DFF3
      FD1797F3D91511BB3931D
      F113F71BFBFD715B5977D3D9
      D5B1BBFBD3BF5DB993F1D13B3
      D53775DDBD799F9BD953DD555D9
      7B1BFD79D5173D1B75B777B311BB9
      9F7
      793
      755
      FD3
      7FB
      D73

```

This program uses loops, the TAB function, and a little algebra to print out a Christmas tree shape on the screen. The MID\$ function will splice out random digits from the string of odd hex numbers. Begin by setting up the screen. For the two special symbols in Line 30 press [SHIFT]:[CLR/HOME] and [CONTROL]:[6].

```

10  REM *** IT IS ODD ***
20  REM *****
30  PRINT " ♥ ↑ "
40  POKE 53280,6 : POKE 53281,1
50  A$ = "13579BDF"
60  FOR Q = 1 TO 30 STEP 2
70  COP = COP + 1
80  FOR J = 1 TO Q
90  D$ = "*"
100 PRINT TAB(20 - COP + J) D$;
120 NEXT J
130 PRINT
140 NEXT Q

```

If you RUN this program as it stands, it will print a handsome triangle of asterisks.

The following lines define the tree's trunk. For the two special symbols in Line 150 press the [SHIFT]:[LCRSR] and [COMMODORE]:[2] combinations in Quote Mode. The first key combination moves the cursor up one line to counteract the very last PRINT in the loop. The second turns the character color to brown.

```
150 PRINT " ● □ "
160 FOR P = 1 TO 6
170 FOR Y = 1 TO 3
180 PRINT TAB (18 + Y)D$;
190 NEXT Y
200 PRINT
210 NEXT P
```

An ordinary tree. Now for some magic. Add the following subroutine:

```
220 STOP
499 REM *** GET A HEX ***
500 X = INT(RND(0)*8) + 1
510 D$ = MID$(A$,X,1)
520 RETURN
```

RUN the program. What happened to the subroutine? We must call it up; add:

```
90 GOSUB 500
175 GOSUB 500
```

There, an odd hexmas tree! Can you figure out how to put a star at the top?

13 Number Theory II

When we used the computer to add up the first 1000 odd numbers (Chapter 9) and to build our odd hexmas tree (Chapter 12), we told the computer what was even or odd. Now we pose the problem: how do we teach our 64 to figure for itself whether a given integer is even or odd? The key to this problem is to make the computer look for some property of the integer in question. In other words, if an integer satisfies one condition, it must be odd; if it satisfies another, it must be even.

We can apply the same process of checking for properties like odd or even to detect whether a graphics character is on the edge of the screen. Once the computer “knows” whether the character is on- or off-screen, we can get a delightful display illustrating a classic example of probability theory: the random walk.

EVEN OR ODD?

The computer can easily detect even or odd, but we need to supply the clues. There are a number of ways to do this. We could ask the computer to do an exhaustive search of all even numbers:

```
5  REM *** I DO IT ***
10  INPUT“NUMBER PLEEYUZ”;N
20  W = W + 2
30  IF N = W THEN GOTO 60
40  IF N = W - 1 THEN GOTO 80
50  GOTO 20
```

```
60 PRINT"IT IS EVEN"
70 STOP
80 PRINT"THAT'S ODD"
90 STOP
```

Line 20 keeps adding 2 to the variable W to get the next even number.

Line 30 checks to see if W equals our mystery number N. If it does, the program skips to the PRINT statement on Line 30.

Line 40 checks to see if W is one greater than N. Since this can only happen if W is odd, the program skips to the PRINT statement on Line 80. In both these cases the program proceeds to Line 90 and STOPS.

Line 50 is reached when the preceding two IF statements are false. It simply says "go back and try the next even number."

The 64 will contentedly loop around all afternoon if need be. Try inputting a value for N of say, 200. Hmmm...how about something faster than that!

What we need is some property that distinguishes between even and odd numbers. Examine the following program:

```
100 REM *** I DO IT FASTER ***
110 INPUT" INPUT NUMBER";ZZ
120 IF ZZ/2 = INT(ZZ/2) THEN GOTO 140
130 PRINT"THE NUMBER ";ZZ;" IS ODD" : STOP
140 PRINT ZZ;"IS EVEN"
```

The key line is 120. When you divide an even number by two you get a whole number. When you divide an odd number by two, 0.5 remains in the result. Line 120 compares $ZZ/2$ to the INTeger portion of $ZZ/2$. This is where the INT function goes to work. Recall that INT function picks off the integer part of a number:

$\text{INT}(33.12345) = 33$

$\text{INT}(0.999999) = 0$

$\text{INT}(-22.00234) = -22$

Notice that the computer always takes the next lower whole number.

Line 120 uses the INT function to test whether a given number meets our criterion for being even, i.e., $ZZ/2 = \text{INT}(ZZ/2)$. If it does, the program jumps to Line 140 and prints out the happy news. If not, the

program goes on to the next line, prints out the sad news, and STOPS.

Try our benchmark value of 200 to compare speed. Try a larger number: this is more like it!

With this trick in hand, we are ready to take a drunken graphics bug on a...

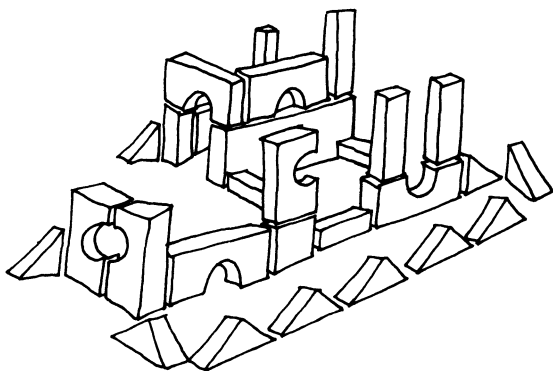
RANDOM WALK

The rules for this exercise are simple. The bug starts at the center of the screen. Every 10 seconds the bug has to decide whether to sway back and forth for another 10 seconds or stagger in one of four directions. We want to keep track of two things: how many times the bug gets back to the starting point and, on average, how long it takes for the poor thing to fall off the edge.

Note that if we start our bug in the center of the screen, the bug can disappear at left or right, in 12 straight steps.

A TEMPORARY RESPITE

Before we set the bug to walking, let's examine the POKE commands for placing characters on the screen. The screen is divided into a 40×24 grid. Two blocks of memory addresses store the current screen display. A 1000-byte block begins at address 1024. Each byte stores the binary code for a particular location on the screen. Another 1000-byte block starts at address 55296 and contains codes that determine the color of the characters on the screen. (The Appendix of your *User's Guide* contains Screen and Color Memory Maps.)



The following test will fill the screen with white A's.

```
10 D = 1024 : C = 55296
20 FOR X = 0 TO 999
30 POKE D + X,1 : POKE C + X,1 : NEXT X
40 STOP
```

It is a good idea to POKE the symbol code and the color code into the same program line. If you wonder why, delete the second POKE statement on Line 30. Now you are not specifying a code for color. Press [RUN/STOP]:[RESTORE] and try RUNning the program.

PROGRAM DESIGN

For long programs, budgeting line numbers allows you to write programs in modules that you can test independently. Working out the budget in advance helps keep your program organized. Here's the budget for "A BUGS DILEMMA":

| | | |
|----------|---|-------------|
| Line 10 |] | Initialize |
| Line 100 | | |
| Line 110 |] | Subroutines |
| Line 200 | | |
| Line 300 |] | Main Logic |
| Line 400 | | |
| Line 410 |] | Keep Score |
| Line 500 | | |

We will start off with the initializing section:

```
10 REM *** A BUGS DILEMMA ***
12 REM *****
20 POKE 53281,9 : POKE 53280,0 : REM *** SCREEN COLOR
30 PRINT "♥" : REM *** CLEAR SCREEN
40 CP = 500 : REM *** STARTING VALUE
100 GOTO 300
```

leaving plenty of room for future additions and modifications. This sec-

tion sets up the screen and defines a constant that will start the bug off in the middle of the screen. You can test the screen by putting in a temporary Line 300 that contains a STOP:

```
300 PRINT"INIT INIT" : STOP
```

Now the program will go through the initialization procedure when you RUN it. When you're through testing, remember to erase Line 300.

MAIN LOGIC UNIT

This unit tests whether the bug has fallen off the screen. If so, the computation jumps to the Keep Score unit. Otherwise an X is printed at the current position. A subroutine call is then made to determine (randomly) where the bug will stagger next. During this call, a little square footprint shows where the bug is at the moment.

You can try arranging your statements in different orders. The particular sequence below keeps the bug hopping. When the subroutine is finished, the program loops back to Line 300. The only tricky bit is defining when the bug has taken one step too many and fallen off the screen.

```
298 REM *****
299 REM *** MAIN LOGIC UNIT ***
300 IF INT(CP/40) = CP/40 THEN 400
310 IF INT((CP+1)/40) = (CP+1)/40 THEN 400
320 IF CP < 0 OR CP > 1000 THEN 400
330 POKE CP+1024,86 : POKE CP+55296,8
340 GOSUB 110
390 GOTO 300
400 PRINT"BYE BYE BUG!"
```

The trick in Line 300 and Line 310 is similar to defining odd or even. We test to see if CP is a multiple of either 40 or 39. Can you figure out how (and why) these two lines work without looking at the following explanation? (Hint: What are the screen grid dimensions?)

These lines check to see if the bug has wandered to the last column on the right or the left of the screen. Every location in the leftmost column shares a common algebraic property: it is evenly divisible by 40. Likewise, when 1 is added to the rightmost column of video location numbers, the sum is evenly divisible by 40. Go ahead and check for yourself that the above claim is indeed true.

Line 320 checks to see if the bug has tried to go off either the top or the bottom of the screen. The criteria for that are a good deal simpler, as you can see.

Now let's turn to the subroutine that determines where the bug will step next:

```
109 REM *** TAKE A STEP ***
110 POKE CP + 1024,108 : POKE CP + 55296,7
120 LET L = INT(RND(0)*5)
130 IF L = 0 THEN CP = CP + 1
132 IF L = 1 THEN CP = CP - 1
134 IF L = 2 THEN CP = CP + 40
136 IF L = 3 THEN CP = CP - 40
150 RETURN
```

This is a straightforward use of conditional logic. The only subtlety here is what happens if the variable L is given the value 4. Since none of the conditions are met, the value of CP remains unchanged and the bug sways back and forth for another 10 seconds unable to make up its mind. The program will (barring your own bug) run now, so go ahead!

KEEPING SCORE

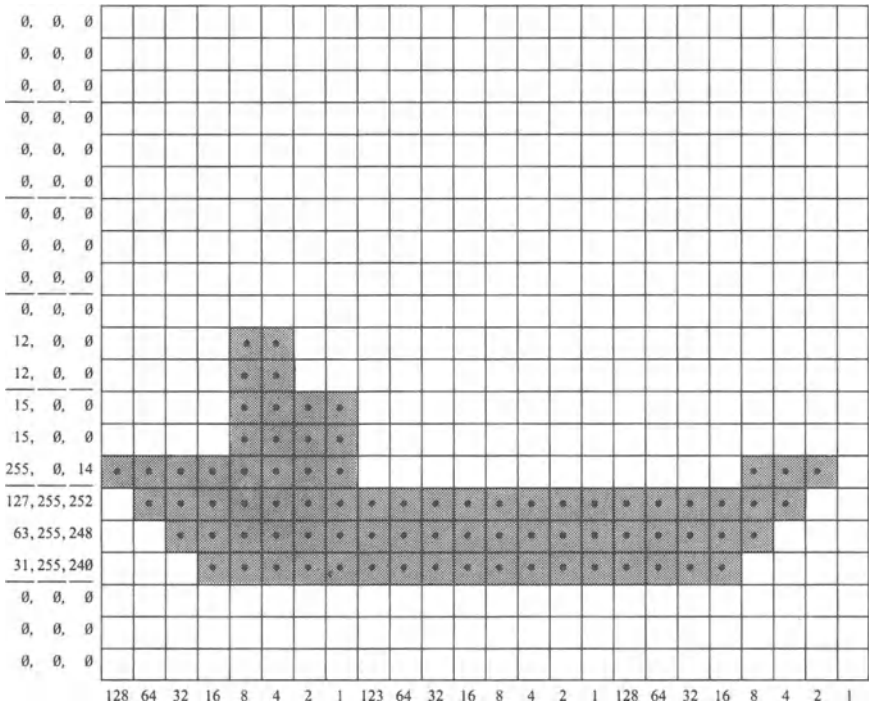
We will now insert a few more lines that create variables for keeping score:

```
360 COP = COP + 1
365 IF CP = 500 THEN ET = ET + 1
```

Line 360 is a simple counter that increments every time the bug has to make a decision. If we have assigned 10 seconds per step then the total (simulated) time in seconds is $COP * 10$. The simulated time in minutes is $(COP * 10) / 60$. Now we can add:

```
399 REM *** KEEP SCORE ***
410 PRINT"THE BUG TOOK ";(COP*10)/60;"MINUTES"
420 PRINT"TO GO OVER THE EDGE " : PRINT
430 PRINT"THE NUMBER OF TIMES RETURNED HOME";
440 PRINT" IS ";ET
```

SHIP (TUGBOAT)



14 Launching a Sprite

In Chapter 11 we built solid block sprites that could move around the screen. We are now ready to program different shapes — for instance, a fleet of sprightly ships.

The actual mechanics of defining sprite shapes involve coding the shape into 63 bytes in memory. The explanation of how to do this is somewhat tedious, but the results (believe me!) are wonderful.

The first step is to get out some graph paper (or else use a ruler and make your own lines) and mark out a 24 by 21 grid as shown in the chart on the opposite page. This chart is already filled in with the blocks that define our first boat.

After you select the shape, you need to consider sprite color. You can program either a sprite in multi-color mode or a sprite in normal, single-color mode. Although in multi-color mode you get three colors in addition to the background color (with a slight loss in resolution), we will use the one-color sprites since multi-color coordination is a bit of an art. When programming “normal” sprites, you may use a single color in any of the 504 little blocks on the chart. This color may vary during the program, but you can have only one color at a time. One way around this is to team several sprites into a single image. We’ll do this in Chapter 16 to get flickering red rocket fire.

So, first we must fill in the chart, and second, translate the pattern on the worksheet into a code 63 numbers long. Later we will store the code in DATA statements.

CODING THE SPRITE

There are 21 rows in the grid. In each row there are 24 blocks. Each

corresponded to a *pixel*, or video picture element. We divide the 24 blocks into three groups of eight. The pattern in each group is coded by a number between 0 and 255. Sixty-three groups of eight will give us our sprite shape. So, we need to produce 63 numbers. The first 27 are easy: they are all 0. Why? Look at the first row of the grid:



Since there are no pixels filled in any of the first three groups of eight, we assign each a 0 and record that fact at the left of the row. These are the first three numbers of our DATA code. The second row is identical to the first—no pixels, so the next three codes are also 0. In fact, since the first 10 rows are empty, the first $3 \times 10 = 30$ numbers in our code are 0.

Now for the first occupied row, Row 11:



The first group of eight has two filled blocks. The decimal values of these blocks are four and eight so we add the values to get a decimal translation number that tells the computer that we want those two pixels filled. You've seen this block system before, when we used the VIC II switchboard to turn on a sprite in the first place. Here we are using the same code to turn on or off the individual pixels that make up the sprite.

Since the other two groups in Row 11 are empty, the full code for the row is 12,0,0. Row 12 is identical to Row 11 so its code is also 12,0,0. Let's skip to the more challenging Row 15.



The first column has seven out of eight blocks filled: the only one empty is the one whose value is 128. Add up the values of the other blocks: $64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$. All eight blocks of the second column are filled, so the code for that column is: $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$ (which, you'll remember, the computer interprets as 11111111). The third column has the 128, 64, 32, 16, 8, and four blocks filled. Adding up these numbers gives the final value and code number for Row 15, namely, 252.

The Character Design Chart in the Appendix shows the filled blocks for numbers 0-127. You may find the chart easier to use than adding the series of eight boxes each time you design a new sprite shape.

THE BOUNDING MAIN

You may wish to go back to Chapter 11 to reacquaint yourself with the sprite switchboard. To start the program:

```
10 REM *** HOME WATERS ***
20 REM *****
30 PRINT "♥" : REM *** CLEAR SCREEN
40 POKE 53280,0 : POKE 53281,6
50 V = 53248 : POKE V + 21,4 : POKE V + 29,4
52 X = 125
60 POKE V + 41,1 : POKE 2042,199
69 REM *** PUT IN WATER ***
70 FOR W = 0 TO 399 : POKE 56295 - W,5
80 POKE 2023 - W,102 : NEXT
```

We are using Sprite 2 as you perhaps noticed when you typed Line 50. RUN the program at this stage to check for typing errors and to see what it looks like. You may wish to choose your own color.

STORING THE SHAPE IN MEMORY

The next section of the program POKES the sprite shape code into memory. We use the READ and DATA statements as we did when programming music. The DATA statements can occur anywhere in the program, but generally they come at the end. Back in Line 60 the computer was told to look for the sprite shape in memory block 199. Now in Line 200 we insert the code to carry out the task. A FOR/NEXT loop is used to READ the DATA and then POKE the values into the 199th block of the 64 bytes in memory. (Only 63 bytes in the block are actually used for the sprite shape code.)

```
199 REM *** PUT BOAT IN RAM ***
200 FOR B = 199*64 TO (199*64) + 62
210 READ F : POKE B,F
220 NEXT
```

Don't let that 62 in Line 200 throw you. If you check, you will find that block of code really POKES in 63 values.

STORING THE CODE

You should develop the habit of organizing your DATA statements. When you start using several sprite shapes you can easily lose track of where one part of shape code ends and another begins. In addition, the SID chip may have dibs on some of that DATA for sound effects. Here's one way to organize:

```
999  REM *** BOAT SHAPE ***
1000 DATA 0,0,0,0,0,0,0,0,0
1002 DATA 0,0,0,0,0,0,0,0,0
1004 DATA 0,0,0,0,0,0,0,0,0
1006 DATA 0,0,0,12,0,0,12,0,0
1008 DATA 15,0,0,15,0,0,255,0,14
1010 DATA 127,255,252,63,255,248,31,255,240
1012 DATA 0,0,0,0,0,0,0,0,0
1014 REM *****
```

Each DATA line contains the information for three rows in our grid. This layout makes it much easier to spot mistakes.

THINGS TO REMEMBER

Review the sprite checklist one last time—always a good habit that will make sure you didn't forget something important.

1. Turn on sprites.
2. Set sprite color.
3. Define sprite shape.
4. Set sprite coordinates.
5. Set memory pointers.

A TEST

One more item on the sprite checklist remains: determining the position of the ship on the screen. Insert this temporary line to check what we have done so far (and to look at the shape of your new ship!).

```
230 POKE V + 4,125 : POKE V + 5,166
```

Without further ado, let's get our ship into the water.

FULL SPEED AHEAD

Now that we have a ship we need a motor. We use the GET command and the conditionals IF/THEN along with a new tool: AND/OR operators. We used the same basic idea in "Cursory Cursor" to move the cursor around. The two special symbols in Lines 330 and 340 are the Quote Mode version of the left and right cursor keys respectively. The simplified GET command allows us to move our sprite and blow its horn at the same time.

```
299 REM *** FULL STEAM AHEAD ***
320 GET A$
330 IF A$ = "[Q]" THEN X = X - 1
340 IF A$ = "[J]" THEN X = X + 1
```

Now we turn to the part of the program that allows us to move our boat back and forth across the screen. There are 512 positions on the horizontal (or X) axis—more than one byte can handle because the binary-counting computer can only put one of 255 different numbers in a single eight-bit address. For 512 positions, we therefore need two eight-bit addresses. We use AND/OR to handle the diplomacy between the two control bytes.

The AND/OR operators behave much like their English counterparts: IF ((you have done your homework) OR (you don't have any)) AND (your chores are done) THEN you can go for a swim. The parentheses in the preceeding sentence correspond to their algebraic counterpart and indicate the order in which the operations are performed. These next program lines control the two bytes that determine the horizontal position of our ship.

```
350 IF (X > 253) AND (A$ = "[J]") THEN X = 0 : RX = 4
360 IF ((X < 1) AND (RX = 0)) OR ((X > 60) AND (RX = 4))
    THEN PRINT "ICEBERG" : END
370 IF (X < 2) AND A$ = "[Q]" AND (RX = 4) THEN X =
    254 : RX = 0
380 POKE V + 4, X : POKE V + 16, RX
390 GOTO 320
```

If you look at Line 380, you'll see that address $V + 16$ is poked with the value RX . $V + 16$ is a single-byte switch that controls all eight sprites. When the switch is on for any particular sprite, the computer will assume that the X coordinate for that sprite is to be added to 255. This allows us to send the sprite to all 512 positions on the X axis.

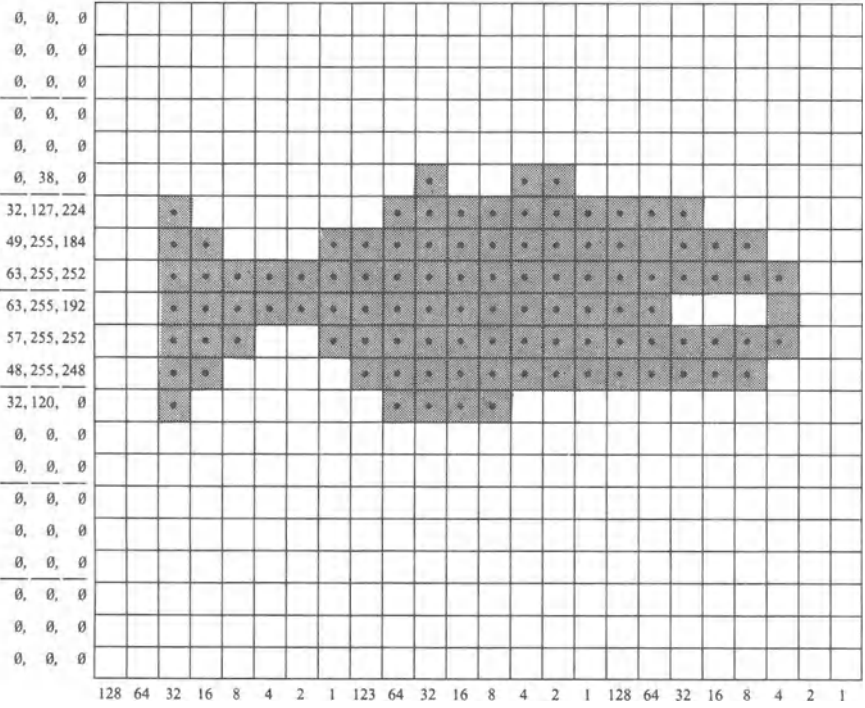
Let's consider the case for Sprite 2. The toggle control number of Sprite 2 is 4. If we POKE $V + 16$ with 4, the computer will interpret the X value in $V + 4$ to indicate that the sprite is to appear at position $255 + X$. If we turn off the left X for Sprite 2 by poking a 0 into the $V + 16$ register, the sprite will suddenly hop 255 spaces to the left. Go ahead and experiment.

If you want, insert tracers into the program by adding the following line:

```
355 PRINT X,RX : REM *** JUST A TEST ***
```

This slows the program and scrolls away the sea, but it will allow you to inspect the values of X and RX as your sprite moves around the screen. When you're satisfied erase the line from the program. This trick helps

BIG WHALE



you to debug when the computer complains of being given an illegal quantity.

By the way, if you like this sort of thing, there is a very nice program called "The Dancing Mouse" on page 166 in Commodore's *Programmer's Reference Guide*.

If you have trouble with the RIGHT/LEFT X concept, then simply put a block graphic mountain on the right side of the screen. Now your boats and whales can happily swim on the left and you can ignore the V + 16 register.

ADDING SOUND

The next section gives our boat a foghorn.

```
119  REM *** SET FOGHORN ***
120  FOR M = 54276 TO 54296 : POKE M,0 : NEXT
121  POKE 54296,15
122  POKE 54277,0 : POKE 54278,240
123  POKE 54284,0 : POKE 54285,240
125  POKE 54273,6 : POKE 54272,71
126  POKE 54280,12 : POKE 54279,143
```

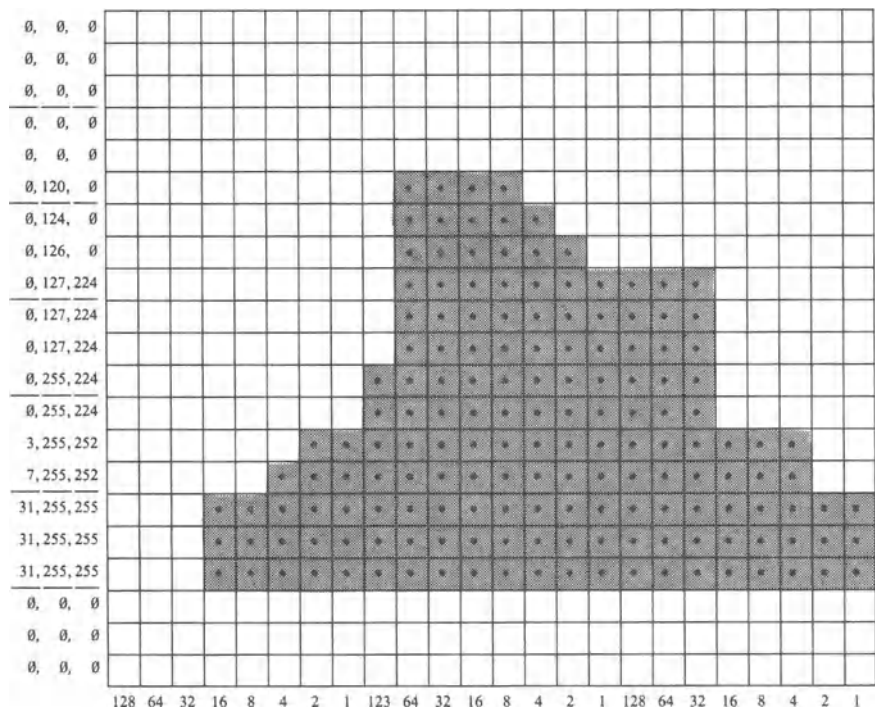
Then we program the "on" switch:

```
1199 REM *** FOGHORN ON ***
1200 POKE 54276,33
1210 POKE 54283,17
1230 FLAG = TI
1240 RETURN
```

And the "off" switch:

```
1259 REM *** FOGHORN OFF ***
1260 POKE 54276,0 : POKE 54283,0
1280 RETURN
```

And finally, we program the [F1] function key to be the pull cord. The special symbol in Line 310 on the next page is the result of pressing the [F1] key in Quote Mode.



```
310 IF A$ = " " THEN GOSUB 1200
345 IF TI - FLAG > 80 THEN GOSUB 1260
```

Now you're sailing in your home (computer) waters. Use the cursor controls to move back and forth and the [F1] key to sound the foghorn. (Some keys have priority over other keys; for example, the foghorn will not sound while the left cursor key is depressed.) But you can sound the horn while moving backwards: Lift the the left cursor key while pressing [F1].

You can extend this program in all sorts of ways. But you will have to pay attention to timing. If the program starts getting too long, insert time tracers as we did in Chapter 9 to find what is holding up the works.

The important point is that you needn't be afraid of long programs. Make them modular, work out a budget in advance, and just keep adding on—like an old New England farmhouse.

PROGRAM PROJECTS

How about a tugboat to add to your fleet? Edit Line 50 to turn on Sprite 3:

```
50 V = 53248 : POKE V + 21,12 : POKE V + 29,4
```

To set the memory pointer and color add:

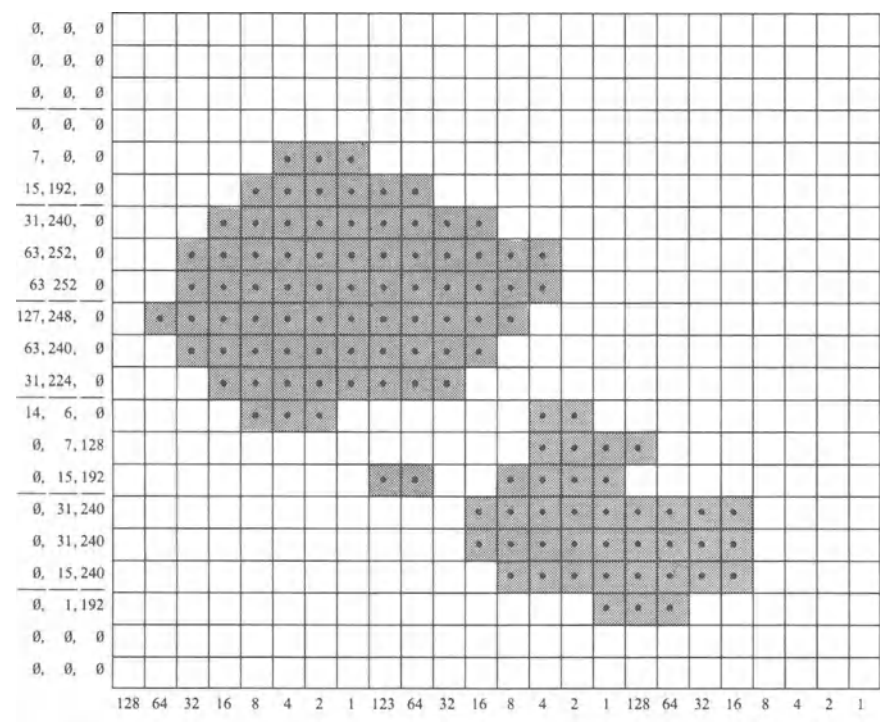
```
55 POKE V + 42,0 : POKE 2043,199
```

and to locate it on the screen:

```
65 POKE V + 6,100 : POKE V + 7,35
```

The rest is up to you, add whales, put in clouds and grow your own little world. Bon voyage...

CLOUDS



15 Tricks Of The SID

One of the truly entertaining aspects of your Commodore 64 is the creation of sound effects. We've included a few here to give you some idea of the fantastic variety. You can use these effects to liven up any program. You should never hesitate to add sound to your programs.

Z-RAY

This program creates a Z-ray. Use it with animation to zap objects.

```
5  REM *** Z-RAY ***
10  POKE 54296,15
20  POKE 54277,0 : POKE 54278,240
30  POKE 54276,17
35  FOR Z = 1 TO 7
40  FOR X = 1 TO 255 STEP 5
50  POKE 54273,X : NEXT X : NEXT Z
60  POKE 54276,16
```

Line 10 sets the volume.

Line 20 sets the ADSR.

Line 30 turns on the triangle waveform.

Line 35 controls the sound's duration.

Lines 40-50 form the backbone of this effect.

Line 50 assigns every 5th value between 1 and 255 to the variable X and then POKES that value into the low frequency number register.

Line 60 turns off the sound.

For fun, alter some of the variables. First try altering sound length by changing Line 35.

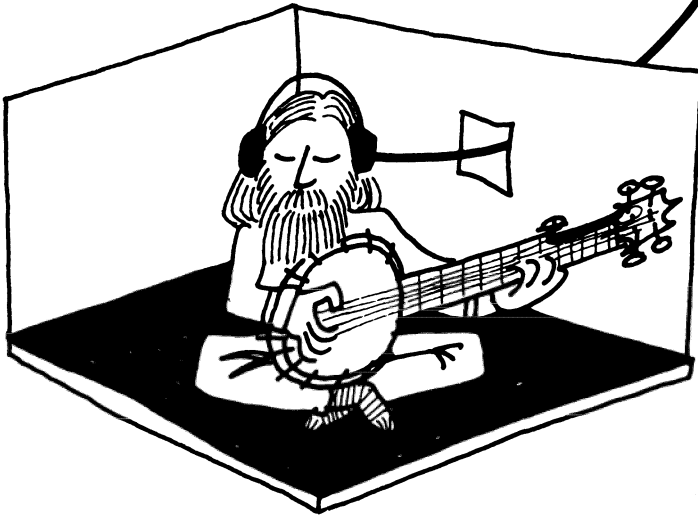
You can also alter the values of X by changing the command in Line 40. Try:

```
FOR X = 1 TO 255 STEP 25
```

SIREN

Here is a program for a warning siren.

```
5  REM *** SIREN ***
10 POKE 54296,15
20 POKE 54277,0 : POKE 54278,240
30 POKE 54276,33
40 FOR X = 1 TO 255
50 POKE 54273,X : FOR J = 1 TO 15 : NEXT J : NEXT X
60 FOR Z = 255 TO 1 STEP -1
70 POKE 54273,Z : NEXT Z
80 C = C + 1 : IF C = 3 THEN POKE 54296,0 : END
90 GOTO 40
```



Line 10 sets the volume.
Line 20 sets the ADSR.
Line 30 sets the waveform.
Line 40 assigns X to all the values between 1 and 255.
Line 50 POKES the low frequency register with the values of X, placing a time loop of 15 between each value.
The next portion of the program handles the siren's descending sound:
Lines 60-70 assign the variable Z to all the values from 255 down to 1 and POKES these values into the low frequency register.
Line 80 is a counter that causes the effect to play three times and then end.
Line 90 creates the repeat loop.

A SLOW ROAD TO NOWHERE

This next special effects program, "TREADMILL," creates the effect of constant movement while going nowhere. To break out of it hit [RUN/STOP] and [RESTORE].

```
5  REM *** TREADMILL ***
10  FOR Z = 54272 TO 54296 : POKE Z,0 : NEXT
20  M = 54272
30  POKE M+24,15
40  POKE M+5,9 : POKE M+6,15
50  READ H,L : IF H<0 THEN RESTORE : GOTO 50
60  POKE M+1,H : POKE M,L
70  POKE M+4,33
80  FOR Q = 1 TO 40 : NEXT
90  POKE M+4,32
100 GOTO 50
200 DATA 4,251,6,167,7,119,7,119
210 DATA 6,167,5,152,-1,-1,-1
```

As you can see, this effect uses the same READ/DATA format as some of our earlier music programs.

You get another kind of effect from this program by changing the waveform setting on Line 20 from 33 to 17, while leaving the 32 setting

in Line 90.

Change the speed of the treadmill by altering the time loop in Line 80.

ALTERED STATES

We have some “other-worldly” sound effects for you to try. They are all built around one program which you can modify to create astounding audio changes. The program is called “Scaler” and in its unaltered state produces a chromatic scale over the entire eight-octave range of your Commodore 64. (That’s why there are so many numbers in the DATA bank.) Be sure to save this program because you can use it for many different sound effects.

```
5  REM *** "SCALER" ***
10  FOR L = 54272 TO 54296 : POKE L,0 :NEXT
20  POKE 54296,15
30  POKE 54277,0 : POKE 54278,128
40  POKE 54276,17
50  FOR T = 1 TO 200 : NEXT
60  READ A,B
70  IF A = -1 THEN GOTO 900
80  POKE 54273,A : POKE 54272,B
90  GOTO 20
100 DATA 1,12,1,28,1,45,1,62,1,81,1,102,1,123,1,145,1,169,1,195,
      1,221,1,250
110 DATA 2,24,2,56,2,90,2,125,2,163,2,204,2,246,3,35,3,83,3,134,
      3,187,3,244
120 DATA 4,48,4,112,4,180,4,251,5,71,5,152,5,237
130 DATA 6,71,6,167,7,12,7,119,7,233,8,978,225,9,104,9,247,10,
      143,11,48,11,218
140 DATA 12,143,13,78,14,24,14,239,15,210,16,195,17,195,18,209,
      19,239
150 DATA 21,31,22,96,23,181,25,30,26,156,28,49,29,223,31,165,
      33,135,35,134
```

```
160 DATA 37,162,39,223,42,62,44,193,47,107,50,60,53,57,56,99,
    59,190,63,75,67,15
170 DATA 71,12,75,69,79,191,84,125,89,131,94,214,100,121,106,
    115,112,199,119,124
180 DATA 126,151,134,30,142,24,150,139,159,126,168,250,179,6,
    189,172,200,243
190 DATA 212,230,225,143,238,248,253,46
200 DATA - 1, - 1
900 POKE 54296,0
910 RESTORE
920 GOTO 20
```

To quit, hit [RUN/STOP]:[RESTORE]. If you wish to save your screen display, hit [RUN/STOP] and type POKE 54296,0. This will turn off the volume.

Out of Step Sound Effect

Now let's create our first change. On a new line type the command:

```
LIST 100-200
```

Now go to Line 100 and delete the first number in the scaler DATA bank along with the comma that follows. Your old line started:

```
100 DATA 1,12,1,28 etc.
```

Your new line will be:

```
100 DATA 12,1,28 etc.
```

Press [RETURN] to enter the change. You have just thrown a monkey wrench into the orderly group of frequency numbers. So the monkey wrench doesn't "damage" the program too much, take the 1 you removed from Line 100 and add it to the end of Line 190. The new Line 190 will look like this:

```
190 DATA 212,230,225,143,238,248,253,46,1
```

Now RUN this new program for a strange piece of computer music.

Robot Speech Sound Effect

Let's change Line 50, which is our time loop. If we increase the speed we can dramatically change the entire effect. Try it. Change:

```
50  FOR T = 1 to 200 : NEXT
```

to:

```
50  FOR T = 1 to 2 : NEXT
```

With that simple change, you have a program that sounds a little like R2-D2's speech in "Star Wars."

Speaking of "Star Wars," with two more short changes you can use this program to create a rocket blast-off sound effect. Return to the program version before you changed the DATA and timing loop.

BLAST-OFF

LIST Lines 40 and 50. Now change:

```
40  POKE 54276,17
```

to:

```
40  POKE 54276,129
```

and change Line 50 from:

```
50  FOR T = 1 TO 200 : NEXT
```

to:

```
50  FOR T = 1 TO 20 : NEXT
```

By changing the waveform from triangle to white noise and by speeding up the time loop, we can produce the rocket effect. RUN and see.

SCALER SYNC EFFECT

To produce an entirely new effect that swings like an outerspace jazz group try this:

Change Line 40 to read:

```
40  POKE 54276,19
```

and Line 50, the timing loop, to read:

```
50 FOR T = 1 TO 50 : NEXT
```

Finally change Line 80, which currently reads:

```
80 POKE 54273,A : POKE 54272,B
```

to:

```
80 POKE 54273,A : POKE 54287,B
```

These changes open the **sync bit**, which synchronizes the fundamental frequencies of Voices 1 and 3. For variations try:

```
80 POKE 54287,A : POKE 54273,B
```

You can create hundreds of effects from this single program. Have fun!

16 Out of This World Graphics

In this chapter we continue the Spacetree adventure begun in Chapter 11. We plan to create a rocket that rises from the ground with a roar and disappears into hyperspace. We can animate the flame sprites by rapidly switching the location in memory where the shapes are stored. We start with the program template already developed:

```
5  REM *** SPACETREE ***
7  REM *****
10 POKE 53280,11 : POKE 53281,0
20 FOR M = 54272 TO 54296 : POKE M,0 : NEXT
30 PRINT " [♥] " : REM *** CLEAR SCREEN
40 POKE 54272,64 : POKE 54273,2
50 POKE 54277,64 : POKE 54278,128
60 V = 53248 : M = 12288 : X = 125
70 POKE V + 21,48
75 POKE 2044,192 : POKE 2045,192
80 POKE V + 43,1 : POKE V + 44,2
90 FOR Q = M TO M + 62 : POKE Q,255
98 FOR TT = 1 TO 54
99 REM *** MAIN LOGIC ***
100 POKE V + 8,X : POKE V + 9,Y
110 POKE V + 10,125 : POKE V + 11,125
```

```

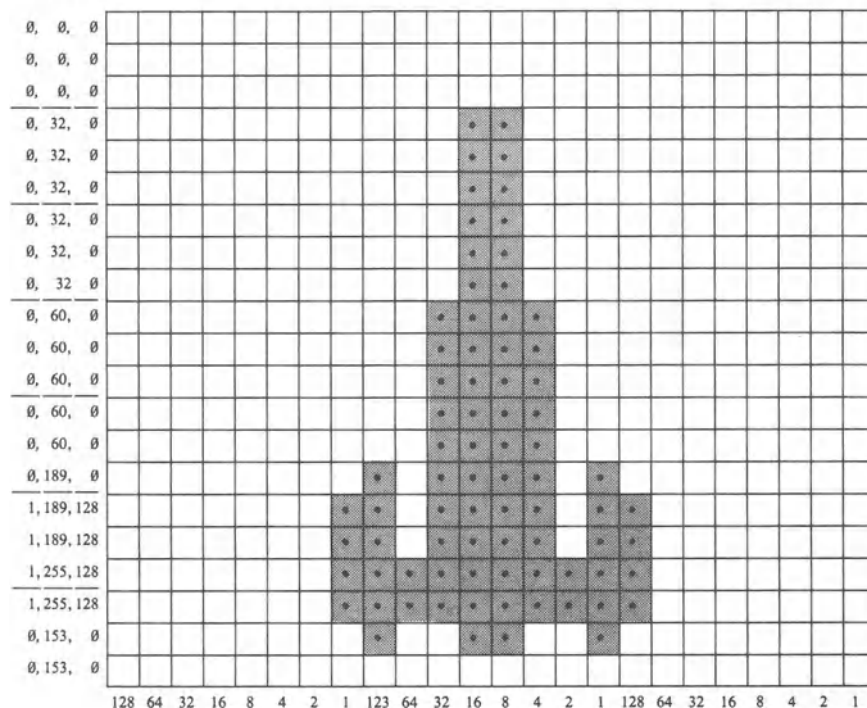
120 POKE V+23,16
200 U = U+1
220 Y = 235 - (INT(1.2↑U)+42)
230 NEXT TT

```

Note that line 80 has been slightly changed; this line controls the color of our sprites—white for the rocket and red for the flame. Test the program to see if all is well. (It worked before.)

Now we design the shape of our spacecraft with techniques we developed in Chapter 14:

ROCKET



```

1000 REM *** ROCKET SHAPE ***
1010 DATA 0,0,0,0,0,0,0,0,0
1020 DATA 0,32,0,0,32,0,0,32,0
1030 DATA 0,32,0,0,32,0,0,32,0
1040 DATA 0,60,0,0,60,0,0,60,0
1050 DATA 0,60,0,0,60,0,0,189,0

```

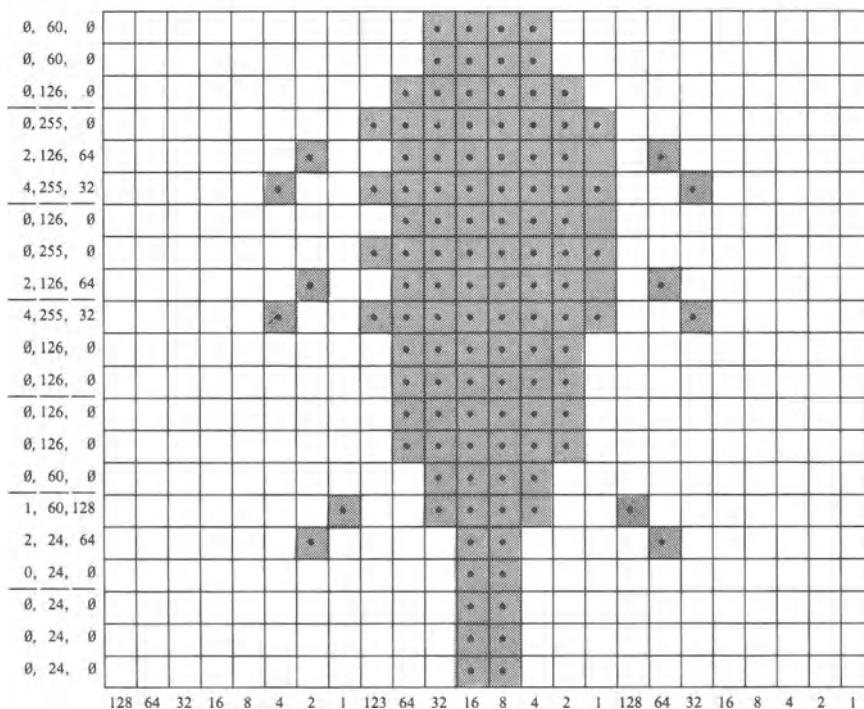
1060 DATA 1,189,128,1,189,128,1,255,128

1070 DATA 1,255,128,0,153,0,0,153,0

1080 REM *****

For the flickering rocket fire we use two flame shapes:

ROCKET FLAME 1



1200 REM *** FLAME SHAPE 1 ***

1210 DATA 0,60,0,0,60,0,0,126,0

1220 DATA 0,255,0,2,126,64,4,255,32

1230 DATA 0,126,0,0,255,0,2,126,64

1240 DATA 4,255,32,0,126,0,0,126,0

1250 DATA 0,126,0,0,126,0,0,60,0

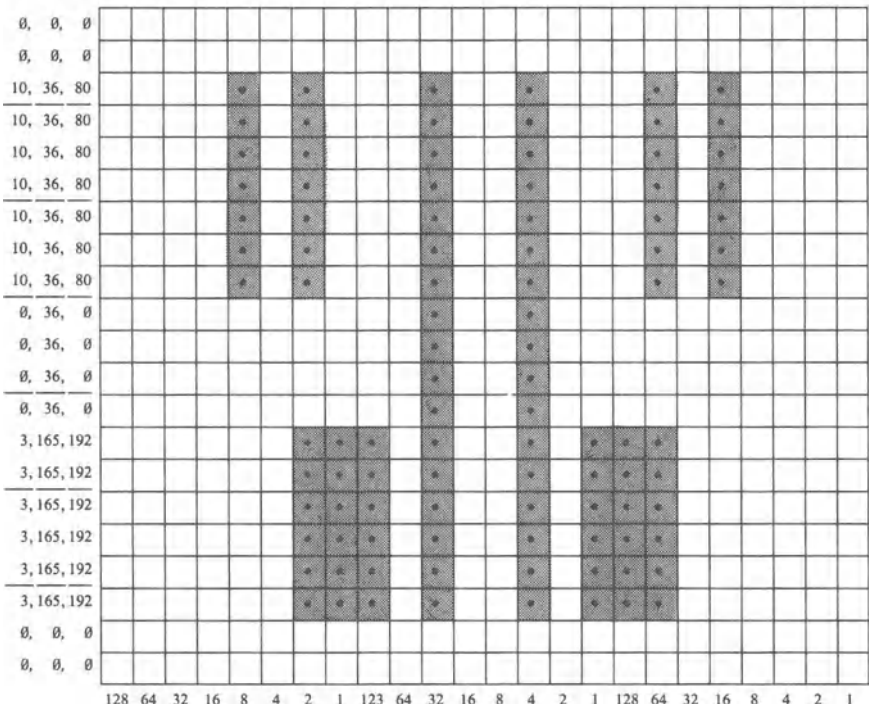
1260 DATA 1,60,128,2,24,64,0,24,0

1270 DATA 0,24,0,0,24,0,0,24,0

1280 REM *****

and:

FLAME 2



```
1300 REM *** FLAME SHAPE 2 ***
1310 DATA 0,0,0,0,0,0,10,36,80
1320 DATA 10,36,80,10,36,80,10,36,80
1330 DATA 10,36,80,10,36,80,10,36,80
1340 DATA 0,36,0,0,36,0,0,36,0
1350 DATA 0,36,0,3,165,192,3,165,192
1360 DATA 3,165,192,3,165,192,3,165,192
1370 DATA 3,165,192,0,0,0,0,0,0
1380 REM *****
```

With the DATA taken care of we have to insert the program lines that will READ the shape codes into memory. We will use memory blocks 192, 193 and 194:

```
90 FOR Q = 192*64 TO (192*64)+62 : READ Z
```

```
92 POKE Q,Z : NEXT
```

Now create Line 94 by changing the 0 in 90 to 4, and then change each 192 to a 193 and press [RETURN]. To create Line 96, change the 92 to 96 and again press [RETURN]. LIST the program to check what you just did. This gives you a pair of loops. We need one more. Use the same procedure to insert these next two lines:

```
86 FOR Q = 194*64 TO (194*64)+62 : READ Z
```

```
88 POKE Q,Z : NEXT
```

For bookkeeping purposes you should note:

Memory block 192 contains Flame 1.

Memory block 193 contains Flame 2.

Memory block 194 contains ROCKET.

We need to set the memory pointers, so edit Line 75 to match:

```
75 POKE 2044,194 : POKE 2045,193
```

We're ready for a test: the rocket should take off but the flame will just float in the middle of the screen. Any errors? Find out now, before we venture too far into the unknown.

All's well? OK, time to get the flame behaving properly. The following section ties the two sprites together; the flame should now follow the rocket as it rises:

```
110 POKE V+10,X : POKE V+11,Y+40
```

```
220 Y = 235 - INT(1.1*U) + 42)
```

The Y+40 value in Line 110 gets us the correct spacing between the flame and rocket. If you want a smaller rocket, change Line 120 to POKE in a 0 instead of a 16 (this is the vertical expansion switch); you'll have to experiment to get the right spacing (but do try 21 in Line 120).

Run the program and correct any syntax errors. If you want to tinker at this point be sure to SAVE what you have lest you crash.

The following lines produce the flicker effect. First we insert the lines that alternate between the blocks in memory where the sprite shapes are stored:

```
224 IF J = 2 THEN POKE 2045,193 : J = 0
```

```
226 IF J = 1 THEN POKE 2045,192
```

To make this work we need a counter, so edit Line 200 to match:

```
200  U = U+1 : J = J+1
```

This fits inside the big FOR/NEXT loop we set up in Chapter 11. Look at Lines 98 through 230 to see what happens in a given pass through the loop.

Before the big test, we need to install the sound effects: For volume control and to set the waveform, add these lines:

```
55  POKE 54296,15
```

```
105 POKE 54276,129
```

Test RUN the program. We still need to insert the hyperspace jump. (You may begin to understand why some programmers use line numbers that increment by 100.)

HYPERSPACE

This section can be customized in all sorts of ways. The first step is to switch off the sprites and turn off the sound:

```
300 POKE V+21,0 : POKE 54276,0
```

Then an explanation is in order (the square box in the following is [CONTROL]:[0] in Quote Mode):

```
310 PRINT TAB(210)“ □ H Y P E R S P A C E ”
```

and hyperspace itself:

```
320 FOR I = 0 TO 14 : POKE 53281,I
```

```
330 FOR M = 1 TO 10 : NEXT M
```

```
340 POKE 53281,I+1 : NEXT I
```























```
350 GOTO 320
```

Well, you've done it! A successful blast-off into hyperspace. We hope you've enjoyed this guided tour of your Commodore 64 and that you feel inspired to experiment further on your own. We'll continue our experiments and hope to take you BEYOND HYPERSPACE in a future book.

Appendix

QUOTE MODE SPECIAL SYMBOL CHART

Reading or typing in program listings with special Quote Mode symbols can be trying. The following chart will help.

| FUNCTION | KEY | APPEARS AS |
|--------------|----------------------|---|
| Cursor home | [CLR/HOME] |  |
| Clear screen | [SHIFT] : [CLR/HOME] |  |
| Down cursor | [↑CRSR↓] |  |
| Up cursor | [SHIFT] : [↑CRSR↓] |  |
| Right cursor | [←CRSR→] |  |
| Left cursor | [SHIFT] : [←CRSR→] |  |
| Black | [CTRL] : [1] |  |
| White | [CTRL] : [2] |  |
| Red | [CTRL] : [3] |  |
| Cyan | [CTRL] : [4] |  |
| Purple | [CTRL] : [5] |  |
| Green | [CTRL] : [6] |  |
| Blue | [CTRL] : [7] |  |
| Yellow | [CTRL] : [8] |  |
| Orange | [COMMODORE] : [1] |  |
| Brown | [COMMODORE] : [2] |  |
| Light red | [COMMODORE] : [3] |  |
| Grey 1 | [COMMODORE] : [4] |  |
| Grey 2 | [COMMODORE] : [5] |  |
| Light green | [COMMODORE] : [6] |  |
| Light blue | [COMMODORE] : [7] |  |
| Grey 3 | [COMMODORE] : [8] |  |

SPRITE SWITCHBOARD CHART

(V = 53248)

ON/OFF
TOGGLE

V + 21
(53269)

MEMORY
POINTERS

| SPRITE# | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------|------|------|------|------|------|------|------|------|
| | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 | 2046 | 2047 |

SPRITE
COLOR

| SPRITE# | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| | V + 39 (53287) | V + 40 (53288) | V + 41 (53289) | V + 42 (53290) | V + 43 (53291) | V + 44 (53292) | V + 45 (53293) | V + 46 (53294) |

X COORDINATE
(HORIZONTAL)

| SPRITE# | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------|------------------|------------------|------------------|------------------|------------------|-------------------|-------------------|-------------------|
| | V + 0 (53248) | V + 2 (53250) | V + 4 (53252) | V + 6 (53254) | V + 8 (53256) | V + 10 (53258) | V + 12 (53260) | V + 14 (53262) |

LEFT X

0
V + 16
(53264)

MULTICOLOR

V + 28
(53276)

Y COORDINATE
(VERTICAL)

| SPRITE# | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------|------------------|------------------|------------------|------------------|------------------|-------------------|-------------------|-------------------|
| | V + 1 (53249) | V + 3 (53251) | V + 5 (53253) | V + 7 (53255) | V + 9 (53257) | V + 11 (53259) | V + 13 (53261) | V + 15 (53263) |

VERTICAL
EXPANSION



V + 23
(53271)

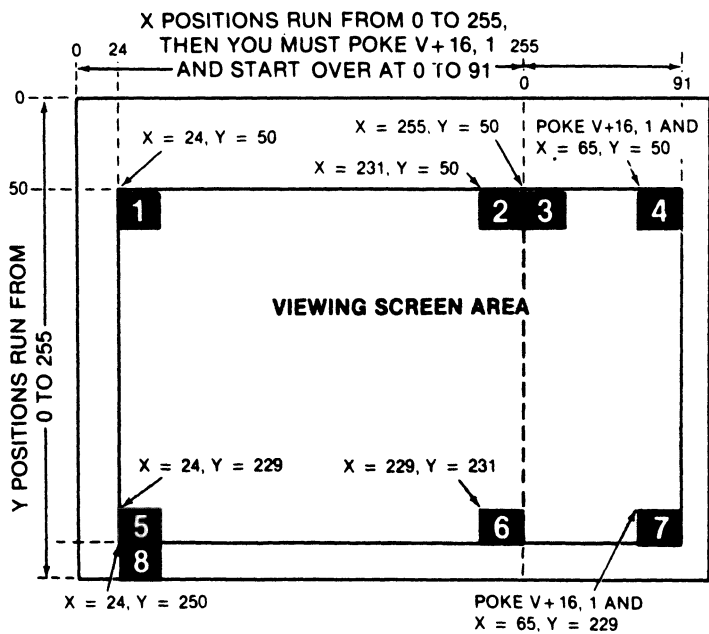
HORIZONTAL
EXPANSION



V + 29
(53277)

For the switches controlled by a single memory address use the sprite switchboard code to individually control all eight sprites.

Sprite Position Chart



The X and Y values that you POKE into memory indicate the position of upper left hand corner of the sprite in question. Chapter 11 develops a program that will let you explore positioning sprites.

Sprite Switchboard Code Summary

| | | | | | | | |
|-----------|-----|----|----|----|---|---|---|
| OFF VALUE | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| OFF BIT | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ON BIT | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| ON | 128 | 64 | 32 | 16 | 8 | 4 | 2 |

Note: 8 bits = 1 byte. Add a group of ON *bits* to determine the sprite control values which are used to activate the switches of single *byte* addresses. For example,

| | | |
|-----------------|-----------------|-----------------|
| 1 0 0 1 0 1 0 0 | 0 1 0 1 0 1 0 1 | 0 0 1 1 1 0 1 0 |
| 128 + 16 + 4 | 64 + 32 + 4 + 1 | 32 + 16 + 8 + 2 |
| = 148 | = 85 | = 58 |

COLOR COMBINATION CHART

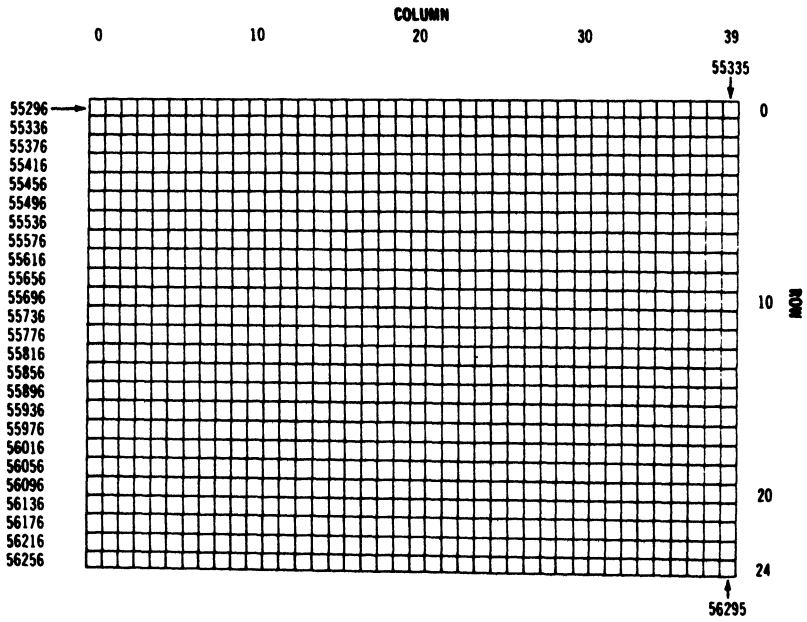
| | | CHARACTER COLOR | | | | | | | | | | | | | | | |
|--------------|----|-----------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| SCREEN COLOR | 0 | X | ● | X | ● | ● | ● | X | ● | ● | X | ● | ● | ● | ● | ● | ● |
| | 1 | ● | X | ● | X | ● | ● | ● | X | ● | ● | ● | ● | ● | X | ● | ● |
| | 2 | X | ● | X | X | ● | X | X | ● | ● | X | ● | X | X | X | X | ● |
| | 3 | ● | X | X | X | X | ● | ● | X | X | X | X | ● | X | X | ● | X |
| | 4 | ● | ● | X | X | X | X | X | X | X | X | X | X | X | X | X | ● |
| | 5 | ● | ● | X | ● | X | X | X | X | X | X | X | ● | X | ● | X | ● |
| | 6 | ● | ● | X | ● | X | X | X | X | X | X | X | X | X | ● | ● | ● |
| | 7 | ● | X | ● | X | X | X | ● | X | ● | ● | ● | ● | ● | X | X | X |
| | 8 | ● | ● | ● | X | X | X | X | ● | X | ● | X | X | X | X | X | ● |
| | 9 | X | ● | X | X | X | X | X | ● | ● | X | ● | X | X | X | X | ● |
| | 10 | ● | ● | ● | X | X | X | X | ● | X | ● | X | X | X | X | X | ● |
| | 11 | ● | ● | X | ● | X | X | X | ● | X | X | X | X | ● | ● | ● | ● |
| | 12 | ● | ● | ● | X | X | X | ● | X | X | ● | X | ● | X | X | X | ● |
| | 13 | ● | X | X | X | X | ● | ● | X | X | X | X | ● | X | X | X | X |
| | 14 | ● | ● | X | ● | X | X | ● | X | X | X | X | ● | X | X | X | ● |
| | 15 | ● | ● | ● | X | ● | ● | ● | X | X | ● | ● | ● | ● | X | ● | X |

■ = EXCELLENT
 ● = FAIR
 X = POOR

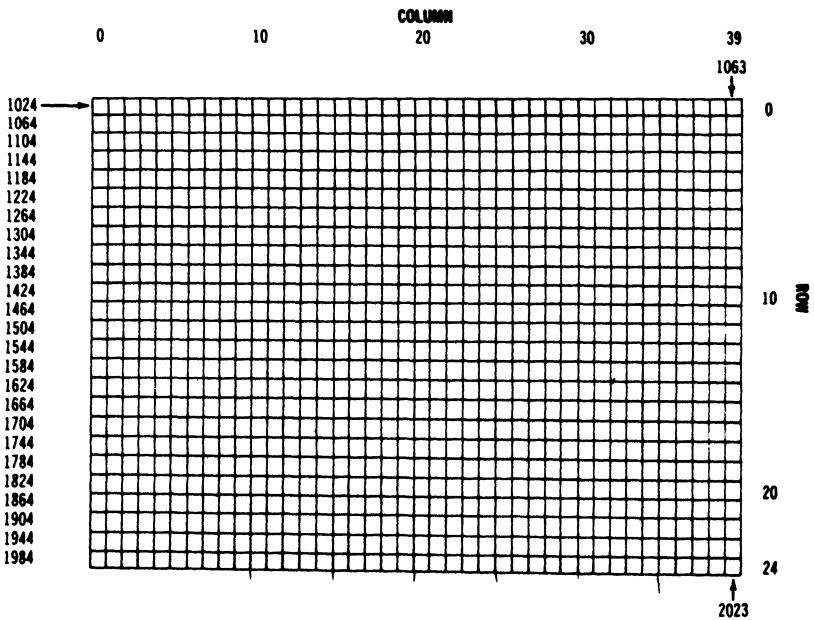
| | | | |
|---|--------|----|-------------|
| 0 | BLACK | 8 | ORANGE |
| 1 | WHITE | 9 | BROWN |
| 2 | RED | 10 | Light RED |
| 3 | CYAN | 11 | GRAY 1 |
| 4 | PURPLE | 12 | GRAY 2 |
| 5 | GREEN | 13 | Light GREEN |
| 6 | BLUE | 14 | Light BLUE |
| 7 | YELLOW | 15 | GRAY 3 |

Certain color choices for character and screen choices yield blurred and distorted images. You can use such choices for creative graphics effects. However for legible text on the screen you need better resolution. The above chart will guide you to appropriate choices for colorful programs.

COLOR MEMORY MAP



SCREEN MEMORY MAP



SID CHIP CONTROL CHART

| | BIT no. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------------------|----------|-----------|-------|-------|-------|-------|--------|--------|--------|
| | DEC no. | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| ADDRESS | FUNCTION | VOICE 1 | | | | | | | |
| 0 | 54272 | LO FQ | f 7 | f 6 | f 5 | f 4 | f 3 | f 2 | f 1 |
| 1 | 54273 | HI FQ | f 15 | f 14 | f 13 | f 12 | f 11 | f 10 | f 9 |
| 2 | 54274 | LO PW | pw 7 | pw 6 | pw 5 | pw 4 | pw 3 | pw 2 | pw 1 |
| 3 | 54275 | HI PW | | | | | pw 11 | pw 10 | pw 9 |
| 4 | 54276 | WAVE FORM | NOISE | | | | TEST | R MOD | SYNC |
| 5 | 54277 | ATK-DEC | ATK 3 | ATK 2 | ATK 1 | ATK 0 | DEC 3 | DEC 2 | DEC 1 |
| 6 | 54278 | SUS-REL | SUS 3 | SUS 2 | SUS 1 | SUS 0 | REL 3 | REL 2 | REL 1 |
| VOICE 2 | | | | | | | | | |
| 7 | 54279 | LO FQ | f 7 | f 6 | f 5 | f 4 | f 3 | f 2 | f 1 |
| 8 | 54280 | HI FQ | f 15 | f 14 | f 13 | f 12 | f 11 | f 10 | f 9 |
| 9 | 54281 | LO PW | pw 7 | pw 6 | pw 5 | pw 4 | pw 3 | pw 2 | pw 1 |
| 10 | 54282 | HI PW | | | | | pw 11 | pw 10 | pw 9 |
| 11 | 54283 | WAVE FORM | NOISE | | | | TEST | R MOD | SYNC |
| 12 | 54284 | ATK-DEC | ATK 3 | ATK 2 | ATK 1 | ATK 0 | DEC 3 | DEC 2 | DEC 1 |
| 13 | 54285 | SUS-REL | SUS 3 | SUS 2 | SUS 1 | SUS 0 | REL 3 | REL 2 | REL 1 |
| VOICE 3 | | | | | | | | | |
| 14 | 54286 | LO FQ | f 7 | f 6 | f 5 | f 4 | f 3 | f 2 | f 1 |
| 15 | 54287 | HI FQ | f 15 | f 14 | f 13 | f 12 | f 11 | f 10 | f 9 |
| 16 | 54288 | LO PW | pw 7 | pw 6 | pw 5 | pw 4 | pw 3 | pw 2 | pw 1 |
| 17 | 54289 | HI PW | | | | | pw 11 | pw 10 | pw 9 |
| 18 | 54290 | WAVE FORM | NOISE | | | | TEST | R MOD | SYNC |
| 19 | 54291 | ATK-DEC | ATK 3 | ATK 2 | ATK 1 | ATK 0 | DEC 3 | DEC 2 | DEC 1 |
| 20 | 54292 | SUS-REL | SUS 3 | SUS 2 | SUS 1 | SUS 0 | REL 3 | REL 2 | REL 1 |
| VOLUME FILTERS MISC. | | | | | | | | | |
| 21 | 54293 | LO FC | | | | | | fc 2 | fc 1 |
| 22 | 54294 | HI FC | fc 10 | fc 9 | fc 8 | fc 7 | fc 6 | fc 5 | fc 4 |
| 23 | 54295 | RES-FILT | RES 3 | RES 2 | RES 1 | RES 0 | FILTEX | FILT 3 | FILT 2 |
| 24 | 54296 | VOL-MODE | 3 OFF | HP | BP | LP | VOL 3 | VOL 2 | VOL 1 |
| 25 | 54297 | POT 1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 26 | 54298 | POT 2 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 27 | 54299 | OSC 3 | 07 | 06 | 05 | 04 | 03 | 02 | 01 |
| 28 | 54300 | ENV 3 | E 7 | E 6 | E 5 | E 4 | E 3 | E 2 | E 1 |

LO FQ = LOW FREQUENCY#
 HI FQ = HIGH FREQUENCY#
 LO PW = LOW PULSE WIDTH
 HI PW = HI PULSE WIDTH
 Δ = TRIANGLE WAVEFORM
 = SAWTOOTH WAVEFORM
 = PULSE WAVEFORM

NOISE = WHITE NOISE WAVEFORM
 ATK-DEL = ATTACK/DELAY
 SUS-REL = SUSTAIN/RELEASE
 LO FC = LOW FREQUENCY CUTOFF
 HI FC = HI FREQUENCY CUTOFF
 RES-FILT = RESONANCE FILTER#
 VOL MODE = VOLUME FILTER MODE

HP = HIGH PASS FILTER
 LP = LOW PASS FILTER
 BP = BAND PASS FILTER
 POT = POTENTIOMETER
 OSC 3 = OSCILLATOR 3
 ENV 3 = ENVELOPE
 GENERATOR 3

TABLE OF NOTE VALUES

| NOTE | DECIMAL | HI FQ | LOW FQ | NOTE | DECIMAL | HI FQ | LOW FQ |
|----------|---------|-------|--------|----------|---------|-------|--------|
| OCTAVE 0 | | | | OCTAVE 4 | | | |
| C-0 | 268 | 1 | 12 | C-4 | 4291 | 16 | 195 |
| C#-0 | 284 | 1 | 28 | C#-4 | 4547 | 17 | 195 |
| D-0 | 301 | 1 | 45 | D-4 | 4817 | 18 | 209 |
| D#-0 | 318 | 1 | 62 | D#-4 | 5103 | 19 | 239 |
| E-0 | 337 | 1 | 81 | E-4 | 5407 | 21 | 31 |
| F-0 | 358 | 1 | 102 | F-4 | 5728 | 22 | 96 |
| F#-0 | 379 | 1 | 123 | F#-4 | 6069 | 23 | 181 |
| G-0 | 401 | 1 | 145 | G-4 | 6430 | 25 | 30 |
| G#-0 | 425 | 1 | 169 | G#-4 | 6812 | 26 | 156 |
| A-0 | 451 | 1 | 195 | A-4 | 7217 | 28 | 49 |
| A#-0 | 477 | 1 | 221 | A#-4 | 7647 | 29 | 223 |
| B-0 | 506 | 1 | 250 | B-4 | 8101 | 31 | 165 |
| OCTAVE 1 | | | | OCTAVE 5 | | | |
| C-1 | 536 | 2 | 24 | C-5 | 8583 | 33 | 135 |
| C#-1 | 568 | 2 | 56 | C#-5 | 9094 | 35 | 134 |
| D-1 | 602 | 2 | 90 | D-5 | 9634 | 37 | 162 |
| D#-1 | 637 | 2 | 125 | D#-5 | 10207 | 39 | 223 |
| E-1 | 675 | 2 | 163 | E-5 | 10814 | 42 | 62 |
| F-1 | 716 | 2 | 204 | F-5 | 11457 | 44 | 193 |
| F#-1 | 758 | 2 | 246 | F#-5 | 12139 | 47 | 107 |
| G-1 | 803 | 3 | 35 | G-5 | 12860 | 50 | 60 |
| G#-1 | 851 | 3 | 83 | G#-5 | 13625 | 53 | 57 |
| A-1 | 902 | 3 | 134 | A-5 | 14435 | 56 | 99 |
| A#-1 | 955 | 3 | 187 | A#-5 | 15294 | 59 | 190 |
| B-1 | 1012 | 3 | 244 | B-5 | 16203 | 63 | 75 |
| OCTAVE 2 | | | | OCTAVE 6 | | | |
| C-2 | 1072 | 4 | 48 | C-6 | 17167 | 67 | 15 |
| C#-2 | 1136 | 4 | 112 | C#-6 | 18188 | 71 | 12 |
| D-2 | 1204 | 4 | 180 | D-6 | 19269 | 75 | 69 |
| D#-2 | 1275 | 4 | 251 | D#-6 | 20415 | 79 | 191 |
| E-2 | 1351 | 5 | 71 | E-6 | 21629 | 84 | 125 |
| F-2 | 1432 | 5 | 152 | F-6 | 22915 | 89 | 131 |
| F#-2 | 1517 | 5 | 237 | F#-6 | 24278 | 94 | 214 |
| G-2 | 1607 | 6 | 71 | G-6 | 25721 | 100 | 121 |
| G#-2 | 1703 | 6 | 167 | G#-6 | 27251 | 106 | 115 |
| A-2 | 1804 | 7 | 12 | A-6 | 28871 | 112 | 199 |
| A#-2 | 1911 | 7 | 119 | A#-6 | 30588 | 119 | 124 |
| B-2 | 2025 | 7 | 233 | B-6 | 32407 | 126 | 151 |
| OCTAVE 3 | | | | OCTAVE 7 | | | |
| C-3 | 2145 | 8 | 97 | C-7 | 34334 | 134 | 30 |
| C#-3 | 2273 | 8 | 225 | C#-7 | 36376 | 142 | 24 |
| D-3 | 2408 | 9 | 104 | D-7 | 38539 | 150 | 139 |
| D#-3 | 2551 | 9 | 247 | D#-7 | 40830 | 159 | 126 |
| E-3 | 2703 | 10 | 143 | E-7 | 43258 | 168 | 250 |
| F-3 | 2864 | 11 | 48 | F-7 | 45830 | 179 | 6 |
| F#-3 | 3034 | 11 | 218 | F#-7 | 48556 | 189 | 172 |
| G-3 | 3215 | 12 | 143 | G-7 | 51443 | 200 | 243 |
| G#-3 | 3406 | 13 | 78 | G#-7 | 54502 | 212 | 230 |
| A-3 | 3608 | 14 | 24 | A-7 | 57743 | 225 | 143 |
| A#-3 | 3823 | 14 | 239 | A#-7 | 61176 | 238 | 248 |
| B-3 | 4050 | 15 | 210 | B-7 | 64814 | 253 | 46 |

CHARACTER DESIGN CHART

| 2 ⁷ | 2 ⁶ | 2 ⁵ | 2 ⁴ | 2 ³ | 2 ² | 2 ¹ | 2 ⁰ | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----|
| | | | | | | | | |
| | | | | | | | | 0 |
| | | | | | | | • | 1 |
| | | | | | | • | | 2 |
| | | | | | | • | • | 3 |
| | | | | | • | | | 4 |
| | | | | | • | | • | 5 |
| | | | | | • | • | | 6 |
| | | | | | • | • | • | 7 |
| | | | | • | | | | 8 |
| | | | | • | | | • | 9 |
| | | | | • | | • | | 10 |
| | | | | • | | • | • | 11 |
| | | | | • | • | | | 12 |
| | | | | • | • | | • | 13 |
| | | | | • | • | • | | 14 |
| | | | | • | • | • | • | 15 |
| | | | • | | | | | 16 |
| | | | • | | | | • | 17 |
| | | | • | | | • | | 18 |
| | | | • | | | • | • | 19 |
| | | | • | | • | | | 20 |
| | | | • | | • | | • | 21 |
| | | | • | | • | • | | 22 |
| | | | • | | • | • | • | 23 |
| | | | • | • | | | | 24 |
| | | | • | • | | | • | 25 |
| | | | • | • | | • | | 26 |
| | | | • | • | | • | • | 27 |
| | | | • | • | • | | | 28 |
| | | | • | • | • | | • | 29 |
| | | | • | • | • | • | | 30 |
| | | | • | • | • | • | • | 31 |

| 2 ⁷ | 2 ⁶ | 2 ⁵ | 2 ⁴ | 2 ³ | 2 ² | 2 ¹ | 2 ⁰ | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----|
| | | | | | | | | |
| | | • | | | | | | 32 |
| | | • | | | | | • | 33 |
| | | • | | | | • | | 34 |
| | | • | | | | • | • | 35 |
| | | • | | | • | | | 36 |
| | | • | | | • | | • | 37 |
| | | • | | | • | • | | 38 |
| | | • | | | • | • | • | 39 |
| | | • | | • | | | | 40 |
| | | • | | • | | | • | 41 |
| | | • | | • | | • | | 42 |
| | | • | | • | | • | • | 43 |
| | | • | | • | • | | | 44 |
| | | • | | • | • | | • | 45 |
| | | • | | • | • | • | | 46 |
| | | • | | • | • | • | • | 47 |
| | | • | • | | | | | 48 |
| | | • | • | | | | • | 49 |
| | | • | • | | | • | | 50 |
| | | • | • | | | • | • | 51 |
| | | • | • | | • | | | 52 |
| | | • | • | | • | | • | 53 |
| | | • | • | | • | • | | 54 |
| | | • | • | | • | • | • | 55 |
| | | • | • | • | | | | 56 |
| | | • | • | • | | | • | 57 |
| | | • | • | • | | • | | 58 |
| | | • | • | • | | • | • | 59 |
| | | • | • | • | • | | | 60 |
| | | • | • | • | • | | • | 61 |
| | | • | • | • | • | • | | 62 |
| | | • | • | • | • | • | • | 63 |

| 2 ⁷ | 2 ⁶ | 2 ⁵ | 2 ⁴ | 2 ³ | 2 ² | 2 ¹ | 2 ⁰ | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----|
| | | | | | | | | |
| | • | | | | | | | 64 |
| | • | | | | | | • | 65 |
| | • | | | | | • | | 66 |
| | • | | | | | • | • | 67 |
| | • | | | | • | | | 68 |
| | • | | | | • | | • | 69 |
| | • | | | | • | • | | 70 |
| | • | | | | • | • | • | 71 |
| | • | | | • | | | | 72 |
| | • | | | • | | | • | 73 |
| | • | | | • | | • | | 74 |
| | • | | | • | | • | • | 75 |
| | • | | | • | • | | | 76 |
| | • | | | • | • | | • | 77 |
| | • | | | • | • | • | | 78 |
| | • | | | • | • | • | • | 79 |
| | • | | • | | | | | 80 |
| | • | | • | | | | • | 81 |
| | • | | • | | | • | | 82 |
| | • | | • | | | • | • | 83 |
| | • | | • | | • | | | 84 |
| | • | | • | | • | | • | 85 |
| | • | | • | | • | • | | 86 |
| | • | | • | | • | • | • | 87 |
| | • | | • | • | | | | 88 |
| | • | | • | • | | | • | 89 |
| | • | | • | • | | • | | 90 |
| | • | | • | • | | • | • | 91 |
| | • | | • | • | • | | | 92 |
| | • | | • | • | • | | • | 93 |
| | • | | • | • | • | • | | 94 |
| | • | | • | • | • | • | • | 95 |

| 2 ⁷ | 2 ⁶ | 2 ⁵ | 2 ⁴ | 2 ³ | 2 ² | 2 ¹ | 2 ⁰ | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----|
| | | | | | | | | |
| | • | • | | | | | | 96 |
| | • | • | | | | | • | 97 |
| | • | • | | | | • | | 98 |
| | • | • | | | | • | • | 99 |
| | • | • | | | • | | | 100 |
| | • | • | | | • | | • | 101 |
| | • | • | | | • | • | | 102 |
| | • | • | | | • | • | • | 103 |
| | • | • | | • | | | | 104 |
| | • | • | | • | | | • | 105 |
| | • | • | | • | | • | | 106 |
| | • | • | | • | | • | • | 107 |
| | • | • | | • | • | | | 108 |
| | • | • | | • | • | | • | 109 |
| | • | • | | • | • | • | | 110 |
| | • | • | | • | • | • | • | 111 |
| | • | • | • | | | | | 112 |
| | • | • | • | | | | • | 113 |
| | • | • | • | | | • | | 114 |
| | • | • | • | | | • | • | 115 |
| | • | • | • | | • | | | 116 |
| | • | • | • | | • | | • | 117 |
| | • | • | • | | • | • | | 118 |
| | • | • | • | | • | • | • | 119 |
| | • | • | • | • | | | | 120 |
| | • | • | • | • | | | • | 121 |
| | • | • | • | • | | • | | 122 |
| | • | • | • | • | | • | • | 123 |
| | • | • | • | • | • | | | 124 |
| | • | • | • | • | • | | • | 125 |
| | • | • | • | • | • | • | | 126 |
| | • | • | • | • | • | • | • | 127 |

Note: for characters that have a dot in the 2⁷ position, add 128 to the

appropriate line of numbers (i.e., to form

| | | | | | | | | |
|---|--|--|--|---|---|---|--|---|
| • | | | | • | • | • | | • |
|---|--|--|--|---|---|---|--|---|

 add 128 to line 29 for a total of 157).

| | | | | | | | | | | | | | | | | | |
|-------------|---------------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---|--------------------|--------------------|------------|--|
| — 95 | 1 49 | 2 50 | 3 51 | 4 52 | 5 53 | 6 54 | 7 55 | 8 56 | 9 57 | 0 48 | + | — | £ | CLR HOME 147 | INST DEL 148 | | |
| | | | | | | | | | | | | | | | | | |
| CTRL | | Q 81 | W 87 | E 69 | R 82 | T 84 | Y 89 | U 85 | I 73 | O 79 | P 80 | @ 64 | * | I 94 | RESTORE | | |
| RUN STOP | SHIFT LOCK | A 65 | S 83 | D 68 | F 70 | G 71 | H 72 | J 74 | K 75 | L 76 | : | ; | = | RETURN 13 | | | |
| | SHIFT | Z 90 | X 88 | C 67 | V 86 | B 66 | N 78 | M 77 | . | ' | / | SHIFT | | | CRSR 17 | CRSR 29 | |
| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |
| | | F1 133 | | F2 137 | | F3 134 | | F4 138 | | F5 135 | | F6 139 | | F7 136 | | F8 140 | |
| | | F8 140 | | | | | | | | | | | | | | | |

KEYBOARD ASCII CODES

Index

| | | | |
|---------------|-------------|------------------|-----------------|
| abbreviations | 63 | GET | 78 |
| address | 10,35 | GOSUB/RETURN | 37,42,44 |
| ADSR | 52,54-56 | GOTO | 37 |
| arithmetic | 11-13 | hexadecimal | 78 |
| ASCII | 122 | IF/THEN | 26,29,49 |
| BASIC | 8-9 | immediate mode | see direct mode |
| binary code | 39 | INPUT | 16-18 |
| bit | 8 | [INST/DEL] | 6,9,24 |
| byte | 8 | INT | 83 |
| calculator | 11 | integer variable | 12 |
| [COMMODORE] | 5 | keyboard | 1-7 |
| [CLR/HOME] | 3,4,15-16 | [LCRSR] | 2 |
| concatenation | 77 | line renumbering | 23 |
| counter | 48 | LIST | 5,11,14,21 |
| crash | 15,35 | loops | 18,29-31 |
| [CTRL] | 4,5 | lower case | 3 |
| [CRSR] | 2 | memory | 8,10 |
| cursor | 2,26 | MID\$ | 79 |
| DATA | 61-62 | modes | 2 |
| debug | 14 | modularization | 42,85,96 |
| delete | 3 | nested loop | 29 |
| direct mode | 11 | NEW | 9,14,15 |
| editing | 19-25 | P1 | 7 |
| errors | 22 | pixel | 90 |
| FOR/TO/NEXT | 18,29,32,37 | pointer | 18 |
| function keys | 7 | | |

| | | | |
|--------------------|----------------|-------------------|----------------------------|
| POKE | 10,21,35 | space bar | 6 |
| PRINT | 3-4 | sprite | |
| pun | 20 | shape | 35,71-76,89-97, 105-110 |
| quote mode | 3-4,5,15-16,25 | switchboard chart | 112-113 |
| | | checklist | 36,73,92 |
| RAM | 8 | STEP | 30-31 |
| random | 32,84 | STOP | 86 |
| [RCRSR] | 2 | string | 77-81 |
| READ | 61-62 | string variable | 12,77,81 |
| register | 35 | subroutines | 37,42,44 |
| REM | 20 | | |
| [RESTORE] | 4 | TAB | 15 |
| [RETURN] | 2,9 | T1 | 50,51 |
| reverse field | 4 | toggle | 7 |
| RND | 32 | tracer | 33-34 |
| [RUN/STOP] | 4,5,9,26 | | |
| RUN | 11,14 | VAL | 34 |
| | | variables | 11,32-34, 48-51,77 |
| SAVE | 22 | variable names | 13 |
| screen editor | 9 | VIC II chip | 1,35 |
| screen memory: | | waveform | 56-58 |
| character map | 84-85,115 | | |
| color map | 84-85,115 | | |
| [SHIFT]:[CLR/HOME] | 3,4,9 | | |
| [SHIFT/LOCK] | 6 | < | 7 |
| SID chip | 1,52,98 | > | 7 |
| simulation | 29,32 | † | 7,15 |
