

How to Program Your commodore

**MORE THAN
130,000
COPIES SOLD**

64

BASIC for BEGINNERS

PLUS HOW TO USE TAPE & DISK DRIVES, PRINTER, SOUND, MUSIC & GRAPHICS

Carl Shipman

How to Program Your **COMMODORE 64®** **BASIC for Beginners**

Carl Shipman

About This Book	2
1 The Keyboard ... <i>Includes color-selection table</i>	3
2 How To Write A Simple Program ... <i>Simplified disk-drive and cassette-operating instructions</i>	12
3 How A Program Makes A Decision ... <i>Using relational operators and logical operators</i>	26
4 Loops ... <i>How to set up and control loops to do nearly anything</i>	41
5 Entering And Editing Program Lines ... <i>Includes control-symbol tables for reference</i>	56
6 ASCII Codes ... <i>With Commodore ASCII code tables</i>	69
7 Screen-Display Codes ... <i>With screen-display code tables</i>	84
8 How To Input Data From The Keyboard ... <i>How the user controls program operation</i>	104
9 Arrays ... <i>How to store and retrieve large amounts of data in an orderly way</i>	115
10 Strings And Things ... <i>Powerful string-manipulation methods</i>	127
11 Mathematical Operations ... <i>Includes useful roundoff routines</i>	147
12 Screen Displays And Menus ... <i>Four ways to display a menu and make choices</i>	161
13 Sorting Routines ... <i>How to sort data items and keep them in order</i>	173
14 Disk And Cassette Files ... <i>Used to make a permanent record of programs and data</i>	188
15 Using A Disk Drive ... <i>Sequential and relative data files. Automatic disk-formatting program</i>	190
16 Using A Cassette Recorder ... <i>Store programs and data on tape. Index program for tape files</i>	221
17 Using A Printer ... <i>List programs, print under program control and as a typewriter</i>	238
18 Binary And Logical Operations ... <i>Advanced computer-control methods</i>	250
19 Sprites And Bit Graphics ... <i>Design and display color graphics using a "sprite-maker" program</i>	266
20 Sound And Music ... <i>Make beeps, sound effects and music. Includes a "music-maker" program</i>	282
21 Programming And Debugging Techniques ... <i>Essential information to help you find bugs</i> ..	305
Appendixes ... <i>Indispensable tables, charts and diagrams for better programming</i>	309
Index	332

NOTICE: The information contained in this book is true and complete to the best of our knowledge. All recommendations are made without any guarantees on the part of the author or HPBooks. The author and HPBooks disclaim all liability incurred in connection with the use of this information. Commodore 64 is a registered trademark of Commodore Electronics Ltd.

All rights reserved. No part of this work may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying and recording, or by any information-storage or retrieval system, without written permission from the Publisher, except in the case of brief quotations embodied in critical articles or reviews.

Published by Knight-Ridder Press, a division of HPBooks, Inc.

P.O. Box 5367, Tucson, Arizona 85703 602/888-2150

ISBN: 0-89586-310-3 Library of Congress Catalog No. 83-82483

©1983 HPBooks, Inc. Printed in USA

Book Design: Paul Fitzgerald

7 8 9 10 11 12 13

About This Book

This book is for beginners. No programming or computer knowledge is necessary because it starts at the beginning. You learn what you need to know in a simple and interesting way. By the end, you can be writing your own programs using color, graphics, sound and music.

Learn how to program your computer in BASIC, the standard programming language for the Commodore 64. Become familiar with your computer so you can use purchased programs in plug-in cartridges, on cassette or on disk.

See and learn how to use a Commodore disk drive, a cassette recorder and the Commodore graphics printer with your computer. But these accessories are not required to start using and enjoying your computer.

You will enjoy the simple programming examples that you type and then run on your computer. Your understanding is reinforced by doing. Place this book beside your computer while typing and running the examples.

When you finish, you will know a lot about computers and how to use them. You'll be able to get maximum enjoyment and benefit from your Commodore 64.

WHAT DOES BASIC MEAN?

The word BASIC is formed from the initials of its full name, *Beginner's All-purpose Symbolic Instruction Code*. That is why BASIC is written in capital letters.

The BASIC programming language was developed at Dartmouth College in the 1960s by John G. Kemeny and Thomas E. Kurtz. Its original purpose was to teach the ideas of programming by using a programming language that was simple and easy to understand.

It proved so useful that it was adopted by computer makers and gradually became a practical programming language rather than just a teaching method. Today, BASIC has matured and become one of several standard programming languages. It is the standard language for personal computers. More computers and computer programmers use BASIC than any other programming language.

VERSIONS OF BASIC

There are several versions of BASIC, sometimes called *dialects*. They are similar. And, if you understand one, you can quickly learn to use another.

Usually, all computer models made by one manufacturer use the same or similar versions. But the BASIC used by one brand may differ from that used by other brands. This book uses Commodore BASIC, sometimes called *CBM BASIC*, meaning *Commodore Business Machines BASIC*.

1

The Keyboard

I'll begin by explaining the keyboard and how it is used. You control and communicate with your computer through the keyboard by typing BASIC words.

Then I discuss and demonstrate BASIC words in the order you should learn them. BASIC is a language that must be learned like any other language. First you learn the meanings of a few words. Then you learn to use the words in groups, like sentences.

As you build your vocabulary of BASIC words, you will gradually increase your understanding of computers—how they work and how to program them. When you do it step-by-step, it's easy.

GETTING READY

If your computer is new, unpack it and connect the keyboard to a TV set or video monitor, following the instructions in the *User's Guide* packaged with the keyboard. I will use the word *display* to mean the TV set or video monitor, whichever you are using.

Turn on the display and then the computer. The computer on-off switch is on the right side of the keyboard. Always turn on the display and other accessories first. Then turn on the computer.

The display should show a dark-blue rectangle with a light-blue border. The dark-blue rectangle is the active area of the screen where letters and other characters are displayed. At the top of the screen, you should see

```
**** COMMODORE 64 BASIC V2 ****  
64K RAM SYSTEM  38911 BASIC BYTES FREE
```

READY.

Letters and numbers are the same color as the light-blue border. You may adjust controls on the display so colors are accurate and characters on the screen are clear and easy to read.

WHAT DOES THE SCREEN DISPLAY MEAN?

The message on the screen tells you that you are using a Commodore 64—which is probably not a surprise. BASIC V2 means that the BASIC programming language is ready to use and that it is Version 2. If Commodore introduces a different version of BASIC, sometime in the future, it will be called V2.1 or V3 or something similar to distinguish it from the present version.

The phrase **64K RAM SYSTEM** refers to the amount of “user” memory in the computer—memory you can use. There is some additional memory that the computer uses.

The letter K stands for the number 1024. The amount of memory is 64 times 1024, which is approximately 64,000. It is exactly 65,536.

Memory is measured in units called *bytes*. One byte can store one number, letter or some other character. A memory of 64K bytes can store 64K characters.

RAM means *Random-Access Memory*. It's called that because the computer can go directly to any part of the memory to get the data it needs. This type of memory is used to store programs and data when you are using the computer. The data stored in RAM is information used or created by a program, such as a mailing list.

Programs and data stored in random-access memory can be changed from the keyboard or by a program. Random-access memory does not store data permanently. When you turn the computer off, everything stored in RAM disappears.

The Commodore 64 has 64K of RAM memory.

The phrase **38911 BASIC BYTES FREE** means that 38,911 bytes are available to store BASIC programs and the data created or used by a program. The rest of the memory is used by support programs that help you use the computer. These support programs are discussed a little later in this chapter.

When the necessary support programs are placed in computer memory, 38,911 bytes are left for BASIC programs and data. That is enough for a very large program.

READY

The word **READY** displayed on the screen is the BASIC prompt symbol. It tells you that BASIC is *ready* to use. When BASIC is ready, you can write a program in the BASIC programming language and run it. Or, you can run a program that is stored on disk or cassette.

THE CURSOR

The blinking rectangle below the prompt symbol is called the *cursor*. When you type characters on the keyboard, they appear on the screen. The cursor shows where the next character you type will appear.

Press the letter key A. On the display the letter A appears where the cursor was, and the cursor moves one space to the right. Press keys BCDEF.

To control where characters or graphics appear on the screen, you must control the location of the cursor. Information in this book shows you how to do that.

WHAT IS BASIC?

BASIC is actually a computer program that is always available in the computer. Its full name is *BASIC Interpreter*. It translates or interprets BASIC words, which people can understand, into computer language that the computer understands.

The BASIC interpreter program is stored in a different type of memory, called *ROM*. This means *Read-Only Memory*. The BASIC interpreter is put into a ROM unit when the unit is manufactured. After that, it cannot be changed. The computer can “read” the data from ROM, but cannot “write” to that segment of memory. That is why it is called *read-only*.

In the Commodore 64, BASIC is stored in an 8K ROM memory. It is one of the support programs that use some of the 64K memory space in the computer.

OPERATING SYSTEM

Another support program is an operating system. It controls the flow of data in the computer and does other things to make the computer easy to use. In the Commodore 64, the operating system is called the *KERNAL*. It is stored in another 8K ROM memory.

The BASIC interpreter, the operating system and a few other things are part of the 64K of usable memory when you write or run a program using the BASIC programming language. These programs leave 38,911 bytes of memory for your BASIC program and data.

DELETE KEY

At the top-right corner of the keyboard is the delete key, labeled INST DEL. It has two functions, INSERT and DELETE. Press it and quickly release it. Notice that it causes the cursor to move to the left and delete the character to its left. You are using the delete function of that key. You will use the insert function later in this chapter.

The delete key repeats if held down. Press it and hold it down until the screen shows no more characters and the cursor is at the top-left corner. It will remain at that location even if you continue holding down the delete key.

COMMODORE KEY

At the lower-left corner of the keyboard is a special key with the Commodore symbol on it. It's the Commodore key. Locate it now. You will use it soon.

SHIFT KEYS

There are two SHIFT keys at the bottom of the keyboard, one on each side. Find them also. What they do is similar to the shift keys on a standard typewriter.

On the left side of the keyboard is a SHIFT LOCK key. When pressed down, it locks and the keyboard behaves as though one of the SHIFT keys were being held down. To release the SHIFT LOCK key, press it again.

MULTIPLE CHARACTERS FROM ONE KEY

Most of the keys can produce four different characters on the screen. For example, the key labeled A can produce both upper-case and lower-case versions of that letter. On the front of this key are two graphics symbols, inside boxes. The key can also produce both of those symbols.

I am discussing the graphics symbols shown on the fronts of the keys as one of the first things in this book because it is one of the first things you notice about the keyboard. As you learn more about the computer, you will probably decide not to make graphics using the symbols on the keyboard because there is another way to do it that you will see later.

I suggest that you not spend a lot of time on the following discussion of keyboard graphics characters. It is mainly to satisfy your curiosity about the symbols on the keyboard. Don't try to memorize anything except the method of making the graphics symbols. Just notice the rest of the things discussed and demonstrated.

Graphics symbols are produced by pressing one key while holding down another key. The following procedure will demonstrate that.

Press key A by itself. The result is a capital A on the display.

Hold down one of the SHIFT keys while pressing key A. This causes a spade symbol to appear on the screen. Notice that this symbol is shown at the right front of the A key.

Hold down the Commodore key while pressing key A. The other graphics symbol is displayed on the screen—the one shown at the left front of the key.

You can use the graphics symbols just like letters or other characters on the screen. You can use them to draw lines or make boxes and other figures on the screen.

So far, three characters have been produced by key A—a capital letter and two graphics symbols. Those are three of the four characters that can be produced by key A.

How to Remember That—At the lower left of the keyboard, notice that the Commodore key and a SHIFT key are side-by-side. To make graphics symbols, one of these special keys is held down while pressing another key. The Commodore key is on the *left* and the SHIFT key is to the *right*.

The Commodore key produces the symbol shown on the *left* front of the letter key. The SHIFT key produces the symbol on the *right*—the same as the positions of these two keys in relation to each other.

Press other letter keys and produce the two associated graphics symbols by using the Commodore and SHIFT keys. Then delete everything on the screen by holding down the delete key.

KEYBOARD MODES

There are two keyboard modes:

Upper-Case-and-Graphics Mode—This is the mode you have been using. When you press a letter key, it displays that letter in upper-case. The key can also produce the two graphics symbols shown on the front of the key. When you turn on the computer, it is automatically set to the upper-case-and-graphics mode.

Upper-and-Lower-Case Mode—The second mode provides both upper-case and lower-case letters. To use this mode, you must select it.

CHANGING MODES

To switch back and forth between the two keyboard modes, hold down the Commodore key while pressing a SHIFT key. In this book, keystroke combinations of that type are written *Commodore-SHIFT*. That means hold down the Commodore key while pressing the SHIFT key.

USING THE UPPER-AND-LOWER-CASE MODE

The screen should be clear and the computer should still be in the upper-case-and-graphics mode. To check that, press key A. The upper-case letter A should appear in the upper-left corner of the screen.

To select upper and lower case, press Commodore-SHIFT. Please do that now. The capital A on the screen changes immediately to a lower-case a. Delete it so the screen is clear again. Now you can experiment to see what the letter keys do in this mode.

Press key A. A lower-case a appears. Lower-case a is the fourth character that can be produced by key A.

Press SHIFT-A. That produces a capital A. In this mode, the SHIFT keys work like those on a standard typewriter. When a SHIFT key is held down, or the SHIFT LOCK key is locked down, the letter keys all produce upper-case letters. When a SHIFT key is not held down, lower-case letters are produced.

Press Commodore-A. The graphics symbol on the left of the key is produced—the same as in the other mode. Experiment with other keys in this mode. Try them with the SHIFT key held down and with the Commodore key held down. Then clear the screen completely, using the delete key.

USE ONE MODE OR THE OTHER

When you press Commodore-SHIFT to change keyboard modes, *all* characters on the screen change to agree with the mode selected, even if they were originally typed in the other mode.

WHICH MODE IS BETTER?

It depends on what you are doing. If you are making a screen display using a lot of graphics symbols, the upper-case-and-graphics mode is better. In this mode, all graphics symbols and only capital letters are available.

To write programs in BASIC, you should normally use the upper-case-and-graphics mode. All BASIC words are normally written using upper-case letters. The computer will understand and run the program correctly if BASIC words are written using lower-case letters, if you have a reason to do that.

For the programs and demonstrations in this book, use the upper-case-and-graphics mode unless I specifically ask you to switch to the other mode.

For ordinary typing, such as a letter or report, the upper-and-lower-case mode is better. Half of the graphics symbols are still available in this mode, so it is possible to use upper and lower case and also have some screen graphics.

When the computer is first turned on, the upper-case-and-graphics mode is automatically selected. The way the computer is set when the power is first turned on is called the *power-up* condition.

CHARACTER SETS

The group of characters that can be produced in each mode is called a *character set*. So far, you have seen parts of two character sets. You will learn more about them later in this book.

The important message in the preceding discussion is that the Commodore 64 has two character sets.

CURSOR CONTROL

One way to control where a character appears on the screen is to move the cursor to the desired location and then produce the character.

As you are now using the computer, this is done with the two cursor-control keys at the lower-right corner of the keyboard. Both are labeled CRSR, meaning *cursor*. The one at the left has an up-arrow engraved at the top of the key and a down-arrow engraved at the bottom. It moves the cursor up and down on the screen. The cursor key at the right has a left-arrow and a right-arrow. It moves the cursor left and right.

When a key has two symbols on the top surface, the SHIFT key produces the upper symbol. Unshifted, the key produces the lower symbol.

The up-down cursor key will move the cursor up if the SHIFT key is held down while pressing the up-down cursor key. If the SHIFT key is not held down, this key moves the cursor down. Try it both ways.

The left-right cursor key will move the cursor to the left if the SHIFT key is depressed while pressing the key. If the SHIFT key is not held down, this key moves the cursor to the right. Try it both ways.

It is important to learn to control the cursor without having to think about which keys you are pressing. Learn to operate the cursor keys with three fingers of your right hand. Place your right index finger on the SHIFT key, your middle finger on the up-down key and your ring finger on the left-right key.

Press the SHIFT key with your index finger. Now you can move the cursor up and to the left, using your middle and ring fingers on the two cursor keys. Try it. Without pressing the SHIFT key, you can move the cursor down and to the right. Try it.

Teach yourself to do it that way until it becomes a habit. Then, you will be able to control the cursor smoothly, without stopping to think about which keys to press.

CORRECTING A TYPING ERROR

If you type a letter or number incorrectly, you can fix it by just striking over the old character with a new one. Clear the screen by holding down the delete key. In the upper-left corner, type ABBDEFG, using upper-case letters.

The cursor should be immediately to the right of the letter G. The mistake is that there are two Bs. The second B should be a C. Move the cursor to the left until it is on top of the incorrect B. Press the letter C. Move the cursor to the right until it is in a clear area. The error is fixed.

USING THE INSERT AND DELETE KEY

It also takes a little practice to become familiar with the operation of this key. Not shifted, it deletes. Shifted, it inserts. You should have the letters ABCDEFG on the screen.

Suppose you want to delete the letter D. To do that, move the cursor so it is on top of the letter E. Without pressing a SHIFT key, press the INST DEL key. It deletes the *next character to the left*, the letter D. Move the cursor to the right until it is in a clear area of the screen. You have deleted a character.

Now, insert three new characters between C and E. Move the cursor until it is on top of the E. Hold down a SHIFT key and press the INST DEL key. That opens up one space by moving the E and all characters to the right of the E. They move to the right on the screen.

Because you want to insert three characters, press SHIFT-INST DEL two more times. That opens up three spaces.

The cursor remains at the left of the open spaces, ready for you to make the insert. Type XXX.

Move the cursor to the right, into a clear area. You have inserted three characters.

Insert Mode—When you have opened up some spaces by pressing SHIFT-INST DEL, the computer is automatically in the *insert mode*.

Here is a characteristic of the insert mode you should know about. Move the cursor over the E. Press SHIFT-INST DEL five times. That opens up five spaces. The computer now *expects* you to insert five characters.

Suppose you actually needed to insert only three characters. Type YYY. Now there are two open spaces that you don't want, but the computer is still expecting you to put something there.

Press INST DEL one time. The computer displays a reversed letter T. I'll explain the reversed T later, in the chapter on editing. Press INST DEL one more time. It displays another reversed T. Now you have entered two more characters, even though you really intended to delete the two open spaces in the line.

The computer is now satisfied because you have filled up the five spaces that you opened. It automatically leaves the insert mode and returns to normal keyboard operation. Press INST DEL two times. It deletes the two characters that you didn't want anyway. You opened up five spaces, typed three new characters and deleted the remaining two spaces.

Suppose you want to insert a space between B and C. Move the cursor on top of the C. Press SHIFT-INST DEL. That puts the computer in the insert mode and opens up a space for one character.

Press the space bar one time. That puts a space in the open space and takes the computer out of the insert mode. Move the cursor to the right into a clear area. You have inserted a space.

MAKING CORRECTIONS INSIDE QUOTATION MARKS

When you are typing an expression in quotation marks, the cursor keys and the insert key don't work as just described. Instead, they operate in the *quote mode*. It is discussed in Chapter 5.

Until you read Chapter 5, follow these suggestions if you need to make corrections to a line that has quotation marks.

If you type an incorrect character, use the delete key to erase it. Then retype it correctly.

If you have any other problem, finish the expression and type the ending quotation mark before correcting the problem. That ends the quote mode. Then move the cursor back between the quotation marks and make corrections as described earlier in this chapter.

If you have a problem and find no other way to solve it, press the RETURN key and retype the entire line.

CLEARING THE SCREEN

You have been clearing the screen the hard way by holding down the delete key. There is an easy way. Near the top right of the keyboard is a key labeled CLR HOME.

If there are no characters on the screen, type your name. Then, move the cursor near the lower-right corner of the screen. Press CLR HOME. The cursor moves immediately to the top-left corner of the screen but the display is otherwise not affected. This is called *homing* the cursor.

Move the cursor to the lower-right part of the screen again. Press SHIFT-CLR HOME. The screen is cleared and the cursor is homed. CLR means *clear* the screen.

Unshifted, the CLR HOME key just homes the cursor. When shifted, it homes the cursor and also clears the screen.

MAKING A BOX

When you are familiar with the graphics characters and how to use them, creating effective screen graphics is rewarding, but usually not easy. Learning is fun, but also not easy. Here is a practice example.

Clear the screen using SHIFT-CLR HOME and move the cursor anywhere near the center of the screen. Press Commodore-P five times. That draws a horizontal line—the top border of a box.

Now make the two sides and the bottom of the box using Commodore-J, Commodore-L and

Commodore-Y. You will probably have to do some deleting and perhaps start over before you finish the box. You will probably move the cursor in the wrong direction several times. That's OK. The way to learn to use graphics characters is to use them.

Look at the graphics characters on the left front of those keys. Decide how they should be used to make the box. Place the cursor where you want each graphics character to print and then press the key.

When you have finished, put a letter X inside the box and move the cursor to the lower-right corner of the screen.

REVERSED CHARACTERS

Characters are made on the screen by a pattern of dots in a rectangle. The shape of the rectangle is the same as that of the blinking cursor. Think of the rectangle as a character space that the computer puts characters in.

For a character to be visible, the dots that make the character must be a different color from the remaining dots in the rectangle. As you are now using the screen, the character dots are light blue and the background dots are dark blue.

It is possible to reverse these colors, which is sometimes effective in a screen display. When reversed, the character dots become the background color and the background becomes whatever color the character used to be. This is called *reversed video*.

To see reversed video, type a capital A. Then move the cursor back so it is on top of the letter A.

Each time the cursor blinks on, it reverses the character. You can see it changing from a normal light-blue character against a dark-blue background to a reversed dark-blue character against a light-blue background in the character space.

PRINTING SPACES

The space bar at the bottom of the keyboard produces spaces. What is actually shown on the screen is a rectangle the same color as the background. If you use the space bar to move the cursor across a character on the screen, the character is erased. Erase the letter A on the screen by using the space bar to print a space on top of it. Then clear the screen using SHIFT-CLR HOME.

CONTROL KEY

Near the top left of the keyboard is a key marked CTRL, which means *control*. It is used in combination with other keys in a way similar to the Commodore and SHIFT keys.

SELECTING REVERSED VIDEO

The number keys are along the top of the keyboard. The front of key 9 is labeled RVS ON. Key 0 is labeled RVS OFF. These labels mean *reverse on* and *reverse off*. To select these functions, use the CTRL key with keys 9 and 0. CTRL-9 turns on reversed video. CTRL-0 turns it off.

Press the space bar a few times to verify that spaces are invisible. Press CTRL-9 to select reversed video. Then press the space bar again. Because reversed video was selected, spaces are reversed. They become a rectangle full of light-blue dots, which are visible against the dark-blue screen background.

SELECTING CHARACTER COLORS

The control key is used with number keys 1 through 8 to select character colors. The color selected is engraved on the front of the number keys.

Press CTRL-1. Then press the space bar a few times. Because spaces are still reversed, you see a good sample of the character color selected by CTRL-1. It is black.

Make a color-bar test of each of the eight colors. Press CTRL-2 followed by the space bar, and so forth up to CTRL-8. The color selected by CTRL-7 will be invisible because it is dark blue, the same as the background.

If the colors don't look right, adjust the color controls of your display.

The Keyboard

A total of 16 character colors is available. The second set of eight colors is selected by using the Commodore key with the number keys 1 through 8. Display them in the same way, starting with Commodore-1. The colors engraved on the number keys apply only when the CTRL key is used.

The accompanying table shows all 16 character colors and the keystrokes used to produce them.

HOW TO SELECT CHARACTER COLORS			
CTRL-1	Black	Commodore-1	Orange
CTRL-2	White	Commodore-2	Brown
CTRL-3	Red	Commodore-3	Light Red
CTRL-4	Cyan	Commodore-4	Gray 1
CTRL-5	Purple	Commodore-5	Gray 2
CTRL-6	Green	Commodore-6	Light Green
CTRL-7	Blue	Commodore-7	Light Blue
CTRL-8	Yellow	Commodore-8	Gray 3

USING CHARACTER COLORS

Turn off reversed video by pressing CTRL-0. Clear the screen. The cursor will remain whatever color you used last. Press CTRL-1 to select black as the character color. The cursor should become black.

Type something such as NOW IS THE TIME FOR ALL GOOD MEN. Change the color by pressing CTRL-2 and continue typing. Display text in all 16 colors, using the same method.

You should observe three things. When a color is selected, it affects all following characters until another color is selected. Changing character color does not affect characters that have already been printed on the screen. Some character colors are easier to read against a dark-blue background than others.

Light-blue characters against a dark-blue background are very easy to read. Some character colors that are difficult to read against dark blue are OK with a different background color. Later, you will learn how to control all three colors on the screen: character color, background color and border color.

RESTORE KEY

At the right side of the keyboard is a key labeled RESTORE. Pressed alone, it doesn't do anything. Try it.

At the left side of the keyboard is a key labeled RUN STOP, which has several uses. One is to make the RESTORE key do something.

The keystroke combination RUN STOP-RESTORE returns the computer to the original power-up condition. It clears the screen, homes the cursor and restores screen colors to the normal light blue against dark blue. It also cancels reversed video. Try it.

RETURN KEY

Just below the RESTORE key is the RETURN key. When pressed, it moves the cursor down one row on the screen and fully to the left. This is similar to a carriage return on a typewriter, which is why the key is marked RETURN.

When pressed, it is a signal to the BASIC interpreter that you have typed an instruction or some data on the screen and you want BASIC to accept whatever you typed and proceed accordingly.

As you are now using the computer, BASIC is in memory and ready to use. It is ready to follow instructions. If you type something on the screen and then press RETURN, the BASIC interpreter will accept whatever was typed. If it doesn't make sense in the BASIC programming language, the BASIC interpreter will tell you so by displaying an error message.

Clear the screen and type GOOD MORNING. With the cursor still on the same line, press RETURN.

BASIC responds with the error message ?SYNTAX ERROR. In English, the word *syntax* refers to the way words are arranged to form sentences. It means about the same thing in relation to BASIC.

GOOD MORNING has no meaning in BASIC, so an error message is displayed. When you type something and send it to the BASIC interpreter by pressing the RETURN key, your choice of words is limited to the vocabulary of BASIC words.

The question mark at the beginning of the error message is a symbol that is used in front of all error messages displayed by the BASIC interpreter.

After displaying the error message, the computer displays the READY symbol to show that it is ready to accept another instruction from the keyboard.

In following chapters, you will see how to get better results from BASIC by asking it to do things that it understands. In the meantime, if you get error messages when you press RETURN, ignore them.

OTHER KEYS

Most of the main keyboard keys have been discussed. Uses for those that were not discussed should be obvious. For example, the + and - keys print those symbols. When shifted, the number keys print punctuation marks and other symbols such as () and \$.

At the extreme right of the keyboard are four special keys called *function keys*. They can be programmed to provide short-cut methods of controlling programs, and to do other things. Function keys are discussed later.

WHY DO ALL THESE THINGS FROM THE KEYBOARD?

You are probably aware that you haven't written a program yet. All of the things discussed in this chapter produce screen displays of one kind or another, but the displays go away when you clear the screen or turn off the computer.

What you can do from the keyboard can also be done from a program. You can write a program to print things on the screen, including graphics symbols, boxes and lines. The program can select colors, do calculations and display the results.

Becoming familiar with things that can be done from the keyboard prepares you to do those things in a program.

PRACTICE

I suggest that you spend a half hour or so using the keyboard in the various ways that have been discussed. To select character colors, refer to the table shown earlier. Don't try to memorize the graphics characters or color selection from the keyboard.

Practice moving the cursor on the screen, using the three-finger method. When you start writing programs, you will use that skill often.

2

How To Write A Simple Program

If your computer is turned on, turn it off and then back on again. You should see the start-up message that says BASIC is ready. Clear the screen by pressing SHIFT-CLR HOME.

PRINT

This is a BASIC word that you will use often. It has several uses. One is to PRINT something on the screen.

Type the following instructions exactly as shown, including the quotation marks. If you haven't found the quotation marks on your keyboard yet, they are SHIFT-2. If you make a typing error, use the delete key to remove the incorrect characters and retype them. If you make an error that can't be fixed that way, finish typing the line, then fix the error. If all else fails, press RETURN and retype the entire line. Type

PRINT "HELLO"

Then press the RETURN key. The RETURN key is a signal to the BASIC interpreter that you have typed something that you want it to accept.

On the top line of the screen, you see the instruction that you typed. You asked the computer to print the word HELLO. The word to be printed is enclosed in quotation marks.

On line 2, you see the computer's response. It printed the word HELLO without the quotation marks. Quotation marks are a way to define what is to be printed. The computer prints the characters enclosed in quotation marks, but not the quotation marks.

After printing HELLO, the computer has finished the instruction. It did exactly what you asked it to. It displays the BASIC prompt symbol, READY, to show that it's ready for your next instruction.

Line 3 on the screen is blank. This is done automatically to separate the prompt from whatever was printed. If the blank line were not there, the display would seem to say HELLO READY.

The cursor is automatically placed at the beginning of the screen line just below the prompt symbol. The next character displayed will be at that location.

STANDARD TERMINOLOGY

This discussion will define a few standard words used in this book and in most other publications about computers.

In the preceding section, I asked you to type an instruction and then press RETURN. Typing something and pressing RETURN is referred to as *entering* whatever was typed. The instruction enters the computer when you press the RETURN key.

When you are asked to *enter* something from the keyboard, type whatever is requested and then press RETURN.

I referred to PRINT "HELLO" as an instruction. It is usually called a *statement*. If I had been using standard terminology, I would have asked you to enter the statement PRINT "HELLO". You would have typed that statement and then pressed the RETURN key to *enter* it into the computer.

When HELLO was printed, I said that the computer had done what you asked it to. It is customary to say that the computer *executed* the statement. When the statement PRINT "HELLO" was entered, the computer executed it and then displayed the BASIC prompt.

What you see on the screen is a statement, the result of executing that statement, a blank line, the BASIC prompt symbol and the cursor.

IMMEDIATE MODE

You haven't written a program yet, but you are getting close. A program is a way of storing a group of statements that the computer executes in sequence when you run the program.

When you entered the statement PRINT "HELLO", the computer executed it *immediately*. The statement is not stored anywhere. It has been executed. To do it again, you would have to enter the statement again.

You are using the computer in the *immediate mode* because it executes statements immediately, as soon as they are entered. In computer literature, the immediate mode has several other names. It is also called *direct mode*, *command mode* and *calculator mode*. This book will refer to it as the *immediate mode*.

The next few demonstrations will be done in the immediate mode. Then I will show you how the program mode works.

DOING ARITHMETIC

Your computer can do simple arithmetic and also more complicated mathematics such as algebra and trigonometry. In general, it can perform any calculation that you want it to do, but you must know how to tell it what you want. There are four arithmetic symbols:

- + means add
- means subtract
- / means divide
- * means multiply

To see how the + symbol works, clear the screen and then enter this statement:

PRINT 2+2

When you pressed RETURN, the computer executed the statement. It told you that the *answer* is 4 and displayed the prompt symbol. You can learn several things from that simple demonstration. They are discussed in the following paragraphs.

EVALUATING AN EXPRESSION

In mathematics, a group of numbers and math symbols is called an *expression*. It states, or expresses, something. $2+2$ is an expression. It is a *numeric* expression because it is made with numbers and mathematical symbols and it has a numeric value. The *value* of the numeric expression $2+2$ is 4.

Finding the value of an expression is called *evaluating the expression*. If I say "How much is $2+2$?" your reply will be "4." I gave you an arithmetic expression and you evaluated it. In other words, you gave the answer.

HOW TO STOP READING THIS BOOK

While working your way through this book, you will enter demonstration programs and run them to learn what BASIC statements do. Some of the demonstration programs are used several times in the same chapter.

You should not attempt to read the entire book at one time. The best way is to do it a chapter at a time. If you do that, you can turn on the computer at the beginning of each chapter and turn it off at the end.

Some of the chapters are fairly long and may require two or three hours. If your schedule doesn't allow that much time, or if you decide to stop reading in the middle of a chapter, remember that the program you have in the computer will disappear from the computer's memory when you turn it off.

There are several things to do about that problem. You can just turn off the computer and let the demonstration program disappear from memory. Then when you start up again, back up in the book to find the beginning of the program you were using and type it again. That puts it back in memory and you can resume at that point. Most of the programs are short, so this will not take a lot of your time. Also, typing programs is beneficial to a beginner even if you have typed them before. Your fingers and your mind learn from repetition.

Another way is to save the program in memory to disk or tape — if you have a disk drive or Commodore cassette recorder. Then, when you resume, you can just play the program back into memory.

Following are the simplest possible instructions to save programs to disk or tape. Chapters 15 and 16 give you a lot more information about saving programs, but these instructions will give you enough information to save the demonstration programs until you read those chapters.

SAVING A PROGRAM TO TAPE

If your cassette recorder is not connected to the computer, turn off the computer and connect it. Follow the instructions in the booklet that is packed with the cassette unit. There is only one connection to make. Plug the cable from the cassette unit into the matching receptacle on the back of the keyboard.

When connected to the computer, the cassette unit draws electrical power from the computer. When the computer is turned on, the cassette unit is also turned on. When the computer is off, both are off.

Before pressing any other keys on the cassette unit, press the STOP key first. If any other key is depressed and locked down, that will release it.

Put an ordinary audio-tape cassette into the recorder. It doesn't have to be new. If it has something recorded on it, all or part of the old record will be replaced with the program that you will save to tape. Commodore recommends using a tape cassette with a maximum of 30 minutes playing time — 15 minutes on each side.

Rewind the tape to the beginning. Set the tape counter to zero by depressing the adjacent button. Always begin each tape operation by rewinding to the beginning.

When you have a program in the computer memory that you want to save on tape, type the word SAVE on the computer keyboard. Then press the RETURN key.

The computer will display the words PRESS RECORD & PLAY ON TAPE. Be sure the tape is rewound. Press the STOP key on the cassette unit. Then simultaneously press the RECORD key and the PLAY key on the cassette unit. The unit will start running. The red light marked SAVE on the cassette unit will glow. The screen display will show the word SAVING.

When the program has been saved to tape, the cassette unit will stop running and the red light will turn off. The screen will display the word READY. It's a good idea to rewind the tape then, so you don't forget to do it later.

The program in computer memory is now on tape. You can turn off the computer.

LOADING A PROGRAM FROM TAPE

The word LOAD means that the program is played back from the tape into the computer memory. Playing it back does not remove it from the tape.

To load the program that was saved to tape earlier, rewind the tape. Then hold down the SHIFT key on the computer keyboard while pressing the RUN/STOP key.

The word LOAD appears on the screen. Just below that, the display shows PRESS PLAY ON TAPE. Be sure the tape has been rewound to the beginning. Press the STOP key on the cassette unit. Then press the PLAY key.

The screen will become blank. The cassette unit will start running. In a few seconds, the words SEARCHING FOUND will appear on the screen. Then it will become blank again while the program is played into computer memory.

When the program is in memory, the computer runs it automatically. The next thing you will see on the screen is whatever the program does. If it displays something on the screen at the beginning of the program, that's what you will see.

If you don't want to run the program, hold down the RUN/STOP key while pressing the RESTORE key. The screen will display the word READY. To verify that the correct program has been loaded, type LIST and press RETURN. The program that was loaded from tape will be displayed on the screen.

HOW TO STOP READING THIS BOOK (continued)

CONNECTING A DISK DRIVE AND FORMATTING A DISK

You must do this only once. If you have already connected the disk drive to the computer, skip that part. If you have already formatted a disk, skip this entire section.

If your disk-drive unit is not connected to the computer, turn off the computer and connect it. If a diskette is in the disk drive, remove it. If you don't know how, read the remainder of this section first. It tells you how.

To connect the disk drive, follow the instructions in the booklet packed with the disk drive. There are two connections to make.

Plug the AC power cord into the back of the disk drive and into an AC outlet. If the disk drive runs, the power switch on the back of the disk drive is turned on. Turn it off.

Then plug the cable with round connectors on each end into the round receptacle on the back of the disk drive. These connectors have five pins. Plug the other end of that cable from the disk drive into the matching receptacle on the back of the keyboard—the one with five holes for five pins.

Before turning on the computer, turn on the disk drive, the display, and any other accessories connected to the computer. Then turn on the computer. Turning on the computer last is standard start-up procedure.

Disks are inserted into the disk drive through a wide horizontal slot at the front. A latch there moves up and down. If down, it blocks the slot, so a disk cannot be inserted or removed. If the latch is down, press it inward. A spring inside the machine will cause it to move upward.

If a disk is in the drive, it will be ejected partway. Grasp it by the label and remove it fully. Put it in the paper jacket that it was originally packaged in. Never touch any of the brown magnetic material through any opening in the disk outer cover.

Placing a Disk in the Disk Drive—Select the disk you will use to store demonstration programs. It doesn't have to be new. This procedure will erase anything stored on the disk.

Grasp the disk by the corner that has the label. Slide the disk into the disk drive with the label up and the elongated opening in the outer cover of the disk away from you. The elongated opening goes into the disk drive first. With the disk in that position, you should see a small notch on the left side. The notch is about 1/4-inch square. It is the *write-protect* notch. It may be covered with a piece of tape. If so, remove the tape.

Gently press the disk fully into the slot on the disk drive. When all the way in, it will lock in position. Close the latch by pulling it down and toward you, until it locks into the down position.

Formatting the Disk—This procedure is done only once—when you are starting to use a disk. Once the disk has been formatted, you don't have to do it again.

This procedure prepares a new disk so it can be used. If the disk has already been used, it erases whatever is on the disk and prepares it to be used again.

On the computer keyboard, type these statements and press RETURN after each statement. Be sure they are typed correctly before pressing RETURN.

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

```
PRINT# 15 "NEW0:DEMO.01"
```

The disk drive will begin running and make some clicking sounds. A red light on the front will glow. When the red light turns off, the disk is ready to use.

When a disk is formatted, it receives a name that identifies the disk. The disk name is DEMO.01.

SAVING A PROGRAM TO DISK

When you want to save a program to disk, so you can turn off the computer, type these statements and press RETURN after each statement.

```
CLOSE 15
```

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

```
SAVE "@0:PROGRAM",8
```

Be sure they are typed correctly. In the SAVE statement, the number following @ is a zero, not a letter O. The disk drive will run and the red light on the front will glow. When the disk drive stops, the program in computer memory has been put on the disk as a file named PROGRAM. To save more than one program, see additional instructions in Chapters 14 and 15.

You can turn off the computer.

LOADING A PROGRAM FROM DISK

When you are ready to put the previously saved program back into the computer and resume reading this book, turn on all accessories and then the computer. Type this statement and press RETURN.

```
LOAD"PROGRAM",8
```

The disk drive will run. When it stops, the program saved using the name PROGRAM will be loaded back into computer memory. To see it, type

```
LIST
```

Then, press RETURN. Now you can resume where you stopped earlier.

How To Write A Simple Program

If you give the computer an expression with unfinished arithmetic to do, it will do it automatically. Instead of printing $2+2$, the computer evaluated that expression and printed its value, 4.

PRINT REALLY MEANS “DO IT”

When you entered the statement `PRINT “HELLO”`, the computer did exactly that. As you know, quotation marks define what is to be printed.

When you entered the statement `PRINT 2+2`, the computer didn’t do exactly that. It evaluated that numeric expression and printed the result.

A good way to think about the BASIC word `PRINT` is to consider it to mean “do it.” The computer will follow the rules and do whatever is indicated. In the first example, it printed exactly what was inside the quotation marks. In the second example, there were no quotation marks. It evaluated the arithmetic expression and printed the result.

Later in this book, you will see several other uses for the word `PRINT`. They all mean “follow the rules and do it.”

MORE ARITHMETIC

To see how the four arithmetic symbols work, enter each of these statements:

```
PRINT 2+2
```

```
PRINT 5-1
```

```
PRINT 8/2
```

```
PRINT 2*2
```

The result of each demonstration should be 4 because I cleverly arranged it that way. You can see why the immediate mode is sometimes called the *calculator mode*. You can use your computer as a calculator.

PRACTICE

Spend a few minutes doing arithmetic on your computer. Experiment to see how it handles decimal numbers, such as `PRINT 105/2`.

Don’t use large numbers. If you do, you will get some strange-looking results that I haven’t discussed yet. For now, use numbers less than 999999999.

NUMERIC AND LITERAL EXPRESSIONS

You have demonstrated two ways to print something on the screen—in quotation marks and not in quotation marks. The difference is important. Please clear the screen and enter

```
PRINT “GOOD MORNING”
```

Don’t forget to press `RETURN` to enter that statement. Then enter

```
PRINT 2+2+2
```

The computer *evaluated* the second expression and printed the result.

Strings—“GOOD MORNING” has meaning, but not mathematical meaning. It is a *literal* expression. Its meaning results from the exact characters in the expression, in that exact order.

To emphasize that, we refer to it as a *literal string*. The word *string* implies that the characters are like beads on a string. They should not be changed, and they should remain in their original order. Otherwise, the meaning is changed or destroyed. The name *literal string* is usually shortened to *string*.

Numeric Expressions—If an expression has mathematical meaning, it is a *numeric* expression. When evaluated, the result is a number. $2+2+2$ is a numeric expression.

RULES ABOUT STRINGS AND NUMERIC EXPRESSIONS

The rules are simple. When you are typing a string, enclose it in quotation marks. This tells the computer that you are entering a string. The computer will print what is included between the

quotation marks and will not attempt to evaluate it or do arithmetic with it.

When entering a numeric expression, don't put it inside quotation marks. The computer will evaluate the expression and print the result. If there is no unfinished arithmetic to do, the computer will just print the value of the expression.

EXAMPLES

It is essential for both you and the computer to know which type of expression you are using. Please clear the screen and enter

```
PRINT "ABCDE"
```

When you press RETURN to enter that statement, the computer knows exactly what to do. It prints that string of characters. The computer will not do arithmetic with the content of a string.

Enter

```
PRINT 2
```

The computer has no difficulty executing that statement. It is not necessary to evaluate that numeric expression because it is already evaluated. Its value is 2. The computer prints that value.

Enter

```
PRINT "2+2+2"
```

Because that expression is in quotation marks, the computer treats it as a string and prints exactly what is between quotation marks. Notice that it did not do the indicated arithmetic. If that expression were not in quotation marks, it would handle it as a numeric and evaluate it.

Enter

```
PRINT 2+2+2
```

That numeric expression is evaluated and its value printed.

Enter

```
PRINT ABCDE
```

That seems to confuse the computer. It prints a value of zero. Because ABCDE is not in quotation marks, the computer handles it as a numeric expression and attempts to evaluate it. Because letters have no number value, it prints zero as the value.

The computer followed a rule that I haven't discussed yet. The rule is obvious: Letters have no number value.

SCROLLING

When the cursor is at the bottom of the screen and the computer executes a PRINT statement, all characters that are already on the screen move up to make room for new characters. They seem to move onto the screen from the bottom. The top rows of characters are forced off the top of the screen and disappear.

This is called *scrolling*. When you are printing a long list of items, scrolling can be a problem because some of the items may move off the top of the screen before you have time to read them. Later, I will show how to control scrolling. For now, it doesn't matter if the display scrolls or not.

PROGRAM MODE

When you are writing or running a program, the computer is in the *program mode*. The following statements demonstrate the difference between the immediate mode and the program mode. To begin, clear the screen and enter these two statements in the immediate mode:

```
PRINT "HELLO"
```

```
PRINT "THERE!"
```

As each statement is entered, it is executed immediately. The screen display doesn't look very good

How To Write A Simple Program

because the statements remain on the screen along with the results. The BASIC prompt appears twice—after each statement is executed.

To put the computer into the program mode, all you have to do is begin a statement with a number, called the *line number*. When the BASIC interpreter “sees” a statement from the keyboard that begins with a line number, it knows automatically that you are entering a *program line*. Instead of executing that statement immediately, it stores it in computer memory, to execute later. Enter these two program lines:

```
10 PRINT "HELLO"  
20 PRINT "THERE!"
```

Notice that neither statement was executed. The numbered program lines are in the computer memory. You have written a two-line program.

The cursor is below program line 20, waiting for you to enter another statement, or whatever you want to do.

If you enter a statement that is not preceded by a line number, you are using the computer in the immediate mode. The statement will be executed immediately.

If you enter another statement with a line number, the computer will accept that statement and store it in memory with the two program lines already there. That would make a three-line program.

When typing and entering statements from the keyboard, you select the program mode or the immediate mode just by using line numbers before statements or not using line numbers.

RUN

The BASIC word RUN causes the computer to run the BASIC program in memory. It switches automatically to the program mode and then runs whatever program it finds in memory. If there is no program in memory, a RUN statement has no result because there is nothing to run.

When running a program, the computer executes each statement in the program, starting with the lowest line number. It executes the statements in line-number order. That is, line 10, line 20, line 30 and so forth until the highest line number has been executed. Then it stops and displays the BASIC prompt symbol to show that it has finished running the program and is ready to do whatever you wish.

A RUN statement is normally entered in the immediate mode.

Do that now by entering

```
RUN
```

The computer executes line 10, then line 20. Then it stops and displays the prompt. Run it again by entering RUN again.

ADVANTAGES OF THE PROGRAM MODE

The BASIC interpreter is always on the job. Most of the things that can be done in a program can also be done by statements in the immediate mode. For example, if you just want the computer to divide 236 by 13, it makes sense to use the immediate mode.

If you want the computer to execute a series of statements and then display the result, it makes sense to write a program.

Programs can be stored on tape or disk and run repeatedly, any time you need them. To repeat an operation in the immediate mode, you have to enter the statements all over again.

Typing errors are a big problem for most people. When you get a program typed correctly, it won't change. It will be correct every time you run it.

A program can “remember” a long series of complicated statements. The computer's memory is better than yours.

There are a few things that can be done only in a program. You can't do them in the immediate mode because they require the structure of a program.

LIST

The BASIC word LIST causes the computer to display the program in memory. The word is normally entered in the immediate mode. If there is no program in memory, nothing will be listed. Enter

LIST

You should see the two-line program that you just ran, followed by the prompt.

LINE NUMBERS

Program line numbers can be any number from 0 to 63999. Notice that 63999 is written without a comma. Never use a comma in a line number. If you do, the computer stops reading the number at the comma. For example, enter

63 , 999 PRINT "A"

Then list the program. You have a new line 63. The statement on line 63 begins with ,999. This is a syntax error because no BASIC statement begins with a comma.

Run the program. Lines 20 and 30 execute correctly. At line 63 the computer stops and displays an error message. It says there is a syntax error in line 63.

Later, when you start writing real programs, you will find that error messages are very helpful. They tell you that something is wrong with your program, give you a clue to help find the problem and even tell you the line number.

HOW TO CHANGE A PROGRAM LINE

One way to change a line is to retype it, including the line number. The new line will replace the old line with the same line number. Enter

10 PRINT "HI"

LIST the program again. Line 10 should be changed to the new line 10 that you just entered. The program won't run correctly because line 63 is still there.

HOW TO DELETE A PROGRAM LINE

To delete a line, type the line number and press RETURN. That enters a program line with nothing on it. It replaces the original line with the same line number. The BASIC interpreter throws out lines with nothing on them, so the program ends up with that line deleted. To see that, enter

63

Then LIST the program again. Line 63 is gone. The program will run correctly, without a syntax error. Clear the screen and run it.

HOW TO ADD A PROGRAM LINE

You add a program line just by entering it. If you don't want the line to replace another line that is already in the program, use a line number that has not been used before.

The BASIC interpreter will put the line in correct numerical order in the program. If you have lines 10 and 20, and then enter a line 15, it will appear between 10 and 20 in the program. Enter

15 PRINT

List the program and there will be three lines in numerical order. Line 15 tells the computer to print nothing. It will print a line on the screen with nothing on it, which is a blank line. Run the program. The blank line is between HI and THERE!

Delete line 15 and run the program again.

SCREEN LINES

There's a difference between a line on the screen and a line in a program. The screen displays lines that are 40 characters long, including spaces. If you type more characters than that, the screen

performs a *wrap-around*. When one screen line is full, the cursor automatically moves to the beginning of the next lower line and the next characters start filling up that line.

In the immediate mode, type a long line of anything until you cause a wrap-around. Then clear the screen.

Sometimes it is confusing to talk about screen lines and program lines. It may not be clear which is which. Instead of *line*, the best name for one horizontal row of characters on the screen is a *row*.

PROGRAM LINES

A program line begins with the line number and ends when you enter it by pressing RETURN. The line number, everything between the line number and the RETURN keystroke, and the RETURN keystroke itself form the program line.

The maximum length of a program line is 80 keystrokes, including the line number and the RETURN keystroke. This means that one program line can occupy two screen rows, but not more.

When you are typing a program line, you can type more than 40 characters. The screen will wrap-around at the end of each screen row. You can type three or four screen rows, but the program will not accept lines longer than 80 keystrokes.

To demonstrate that, enter the following program line. Put a space after 88 and after PRINT. After the ending quotation mark on the second screen row, press RETURN. The cursor will be in space 40 on the second screen row. You will enter exactly 80 characters, including the RETURN keystroke.

```
88 PRINT "123456789012345678901234567890
12345678901234567890123456789012345678"
```

Notice that the wrap-around happens automatically at the end of the first screen row. There are 40 keystrokes on that row. There are also 40 keystrokes on the second screen row, but the last keystroke was RETURN.

List the program. The line was accepted by the program. To get a small reward for all that typing, run it. It should run OK.

Now try to enter a longer line using line number 99. Be sure that it extends into the third screen row. The line will be ignored by the computer and will not appear in your program. List the program to verify that. Then delete line 88 and clear the screen.

LOGICAL LINES

A program line may occupy more than one row on the screen. Program lines are called *logical lines*. A logical line is one complete program line, even if it takes more than one row on the screen to display it.

CHOOSING LINE NUMBERS

Most programmers begin with line number 10 and use line numbers that are multiples of 10, such as 10, 20, 30 and so forth. This leaves room to insert new lines later, between the existing lines. I suggest that you write programs that way.

DON'T USE LETTERS INSTEAD OF NUMBERS

If you are accustomed to using a standard typewriter, you may have the habit of typing a letter O for the number zero and the letter l for number one. On a typewriter, these characters are the same or similar, and the typewriter doesn't mind.

When you use a computer keyboard, the computer knows which key was pressed. If you press the letter O or the letter l, that's what the computer accepts as the input, even if you intended to type zero or one. The program will not run correctly.

CLEARING THE SCREEN BY A PROGRAM STATEMENT

You should still have the two-line program in memory that prints HI THERE! on two screen lines. List the program in memory to be sure. If it isn't there, or isn't correct, fix it.

When the program is OK, clear the screen by pressing SHIFT-CLR HOME. Then run the program by entering the word RUN. It seems to say RUN HI THERE! followed by the prompt.

The RUN statement was entered in the immediate mode, before the program ran, but it remains in view on the screen. It would look better if the word RUN were not displayed. You can do that by clearing the screen at the beginning of the program.

A statement that clears the screen is `PRINT CHR$(147)`. The symbols `CHR$(147)` are a code that tells the computer to clear the screen. I will explain how it works later. You can start using it now because you need to clear the screen with a program line.

Add a new line 5 to the program by entering

```
5 PRINT CHR$(147)
```

Run the program. Line 5 clears the screen. Then lines 10 and 20 print HI THERE! on two rows. Then there is a blank row followed by the BASIC prompt. The display no longer says RUN HI THERE!. I think it looks a lot better.

Line 5 is another example of considering the word PRINT to mean “do it.” That statement doesn’t display the “clear-the-screen” code. It follows rules built into the computer. The rule in this case is that “printing” that code from a program line clears the screen and homes the cursor.

It’s good programming practice to clear the screen at the beginning of a program. That removes the RUN statement and anything else that may be on the screen.

When writing a program, clear the screen whenever there is anything displayed that doesn’t need to be there. That will make your screen displays look better and your programs easier to understand and use.

CARRIAGE RETURNS AND LINE FEEDS

These terms were originally used with machines that print on paper, such as typewriters and teletype machines. They are also used to describe cursor movement on the screen.

Strictly speaking, a carriage return moves the cursor fully to the left, on the same screen row. A line feed moves the cursor down one row without moving it to the right or left. If the cursor is at space 25 on a row, a line feed moves it to space 25 on the next lower row.

Using those strict definitions, moving the cursor fully to the left and down one row would require both a carriage return and a line feed.

However, it is common to use the term *carriage return* to include both actions—moving the cursor down one row and fully to the left. I will use it that way in this book.

PRINT STATEMENTS PRODUCE A CARRIAGE RETURN

When a PRINT statement is executed, the last thing that happens is an automatic carriage return. The cursor moves down one row and fully to the left. Please list the program in memory. Line 10 prints HI followed by an automatic carriage return. That’s why line 20 prints THERE! on the next row, fully to the left.

SUPPRESSING CARRIAGE RETURNS

Run the program in memory. Notice that the display begins on row 2 of the screen, rather than row 1.

List the program again. Line 5 is the reason. Even though it actually clears the screen, it is a PRINT statement. All PRINT statements produce an automatic carriage return, no matter what they do, unless you prevent it.

One way to prevent it is to put a semicolon at the end of the line. Here is an easy way to add the semicolon. Using the cursor-control keys, move the cursor to the end of line 5 just to the right of the last quotation mark. Press the semicolon key. Then press RETURN.

Move the cursor to a blank row near the bottom of the screen. List the program. Line 5 should have a semicolon at the end. Run the program. The display now begins at the top row on the screen. The semicolon at the end of program line 5 *suppressed* the automatic carriage return.

MULTIPLE STATEMENTS ON A PROGRAM LINE

You can put more than one statement on a program line. Use a colon after each statement

How To Write A Simple Program

except the last one. The BASIC interpreter recognizes a colon as a separator between statements. When the program runs, the statements execute one after the other, from left to right.

ANOTHER IMPROVEMENT

The word RUN no longer appears on the screen when you run the program in memory, but the prompt symbol makes the display seem to say HI THERE! READY.

The prompt appears automatically when the last program line has executed. There is no way to prevent that. You can move the prompt downward on the screen, which sometimes makes the display easier to interpret. List the program and add this program line:

```
30 PRINT:PRINT:PRINT:PRINT
```

Each of those four PRINT statements prints nothing, but each produces an automatic carriage return. That will put four blank rows between THERE! and READY. As you know, one blank row is printed automatically, so there will be a total of five blank rows. List the program again and then run it.

By using more PRINT statements that print nothing, you can move the prompt closer to the bottom of the screen if you wish. If you move it too far down, it will cause a scroll. Try it.

By using PRINT statements in a similar way, before a display is printed, you can space it down from the top of the screen. That should be done after the screen is cleared. Put some PRINT statements on a new program line 8 to move the display down on the screen. List the program to check your typing and then run it.

COMMAND OR STATEMENT?

BASIC words are referred to as *commands* and *statements*. There is a small difference in meaning. Commands usually do something to help operate the computer. LIST is a command.

Statements are usually used in a program and do something in the program. PRINT is a statement.

However, some commands are also statements and can be used on program lines. The distinction between command and statement is not important.

END

END is a BASIC word used to mark the end of a program. The computer will stop running the program and return to the immediate mode when it executes an END statement.

Some programmers put an END statement in a program as the last line. It doesn't do anything because the computer will stop anyway when it has no more lines to execute.

END is useful in special cases. You will see an example in the next chapter.

NEW

If you have a program in memory and decide to write a different program, you should first remove the existing program from memory. Otherwise, the program lines from the new program will intermingle with those already in memory. Neither of the two programs will run correctly.

To clear memory for a new program, enter

```
NEW
```

Please do that now. Then list whatever is in memory, by entering

```
LIST
```

No program lines appear on the screen. This verifies that there is no program in memory.

DUPLICATING PROGRAM LINES

Occasionally, you will want to make several program lines that are exact duplicates except for the line numbers. Here's an easy way to do that. Enter

```
1 PRINT "A"
```

Then move the cursor back up so it is on top of the line number. Press key number 2. The

number 2 will replace the number 1 on the screen. Without moving the cursor, press RETURN. That enters a new line 2 into the program. It is the same as line 1 except for the line number.

You can't see line 1 because line 2 is "on top" of it. It is still in the program. Move the cursor down to a clear area on the screen and list the program.

When you press RETURN *with the cursor in a program line*, the BASIC interpreter accepts that entire program line and puts it in the program.

To enter a program line, the cursor can be *anywhere* in that line when you press RETURN. If the program line occupies two screen rows, the cursor can be on either of the two screen rows.

Duplicating program lines by this method is easy and much faster than typing a new line. I call it *overtyping line numbers*.

By overtyping line numbers, make program lines 1 through 9 that are all identical to line 1. List the program. Clear the screen and run it. It prints the letter A nine times, on nine screen rows.

SPACE AFTER THE LINE NUMBER

BASIC requires a space after the line number, but you don't have to put it there. If you don't, the BASIC interpreter will.

To demonstrate that, list the program. By overtyping, change line 9 to line 10. When typing the number 10, the zero will fall into the space that followed the 9. At this point, line 10 doesn't have a space after the line number. Press RETURN. The cursor stops on the R of the READY. prompt.

Press RETURN again. That causes an error message that says ?OUT OF DATA ERROR. Ignore the error message.

Now the cursor is on a blank row. List the program. There is a space after the line number in line 10. The interpreter put it there.

Using the RETURN key to move the cursor through the prompt will always produce that error message. It does no harm but the error message may cause the display to scroll. It's better to move the cursor down by pressing the cursor-down key.

Small Mistake—We forgot to clear the screen at the beginning of that program. Fix it by using program line 0 to print the code that clears the screen. It should look like this:

```
0 PRINT CHR$(147)
```

Move the cursor to a blank line on the screen. List the program and run it again. The display begins on row 2 because there is no semicolon at the end of program line 0.

MAKING SIMILAR PROGRAM LINES

You will often want to make several program lines that are similar but not identical. A good way to do that is to create a group of identical lines by overtyping line numbers. Then change the duplicate lines as needed so they do what you want them to.

PRINTING THE ALPHABET

Write a program that prints all letters of the alphabet, each on a separate screen line. The program lines will be similar. Start by modifying the 10 lines now in memory. Clear the screen from the immediate mode. List the program.

Line 2 should print the letter B instead of A. Overtyping works anywhere in a program line. Move the cursor over the letter A in program line 2. Press the B key. Then press RETURN to enter the changed line. Move the cursor to a blank row below the program. List it. Line 2 is changed.

Using the same technique, fix lines 1 to 10 so they print the letters A to J. List the program and run it.

To finish this program, you need some more similar lines. List the program again. By overtyping line numbers on program line 10, create lines 11 through 26 that are identical to line 10. List the program. Lines 11 through 26 all print J.

Here's an easy way to change them—if you have taught your fingers to control cursor position using the SHIFT key and the cursor keys. If not, you will have difficulty, but your fingers will be learning to operate the keyboard. Follow this procedure exactly:

How To Write A Simple Program

Move the cursor over the J in line 11. Press K. Don't press RETURN.

Move the cursor over the J in line 12. Press L. Don't press RETURN.

Continue doing that until you have changed line 26 to print Z. Don't press RETURN yet.

All of those lines have been changed on the screen, but none has been changed in memory yet because the new lines have not been entered. To enter a program line, you must press RETURN with the cursor anywhere in that line.

Move the cursor up to line 11 again. Press RETURN. That enters line 11 and the cursor moves automatically to line 12. Press RETURN to enter line 12. Continue pressing RETURN until you have entered all of the changed lines.

List the program. It scrolls because you have more program lines than there are screen lines. It is difficult to read.

HOW TO SLOW DOWN THE SCROLL

When listing a program, the CTRL key will slow down the scroll to give you more time to read each line. Type

LIST

Then place one finger on the RETURN key and another on the CTRL key. Press RETURN. As soon as program lines begin to appear on the screen, press CTRL and hold it down. Release the CTRL key and scrolling resumes its normal speed.

RUN STOP

What if you see an incorrect line while listing a program? The RUN STOP key will stop listing. If you press it quickly, the incorrect line will remain on the screen. Then you can retype it or fix it.

The RUN STOP key will also stop program execution. Whenever the computer is doing something you want to stop, press RUN STOP.

If you have a program on a tape cassette, pressing SHIFT-RUN STOP will run that program. Otherwise, SHIFT-RUN STOP doesn't do anything.

LISTING PART OF A PROGRAM

I asked you to make the 27-line program that prints the alphabet so I could demonstrate listing parts of a long program. You have been using the LIST command to list the entire program. There are some alternatives.

Please enter these commands in the immediate mode. Observe the results.

LIST 13

LIST 0-10

LIST 11-26

LIST -6

LIST 18-

LIST

A summary of LIST commands is in the accompanying table.

STOP

This is a BASIC word used in programs. To stop program execution from a program line, put a STOP statement in the program. You should still have the alphabet program in memory. Run it. Then list it. Change line 15 by retyping it:

15 STOP

Run the program again. It prints letters A through N, stops and returns to the immediate mode. It also tells you the number of the last line that executed. Stopping a program during execution is called *breaking*, or a *break*.

SUMMARY OF LIST COMMANDS (# symbol represents a line number)

LIST	Lists entire program. Display may scroll.
LIST #	Lists only specified line number.
LIST #-#	Lists all lines from first specified line number to second specified line number.
LIST -#	Lists from beginning of program to specified line number.
LIST #-	Lists from specified line number to end of program.
CTRL	Slows down scroll rate when pressed during listing.

CONT

This is a command that is used in the immediate mode. CONT means *continue*. When you have stopped a program by a STOP statement, you can restart it by entering the command CONT from the immediate mode. Try it. The program continues at line 16 and prints the letters P through Z.

This works only if the program is not changed while it's stopped. If you make a program change and then enter CONT, an error message results. It says ?CAN'T CONTINUE ERROR.

To see that error message, run the program again. When it stops, delete line 26 by typing the line number and pressing RETURN. That changes the program. Enter CONT.

When you stop a program by a STOP statement and then change the program, you can run it again with a RUN statement. There is another way that I will mention later.

If a program executes an END statement, it returns to the immediate mode, but it does not display the last line number executed. If a program stops due to an END statement, and the END statement is not the last line in the program, entering CONT will continue execution just as though the program had stopped due to a STOP statement. To demonstrate that, change line 15 to read

15 END

Run the program. When it stops, enter CONT. It will continue execution from where it stopped.

REVIEW

This chapter promised to show you how to write a simple program. It did that. The programs were very simple. The main purpose was to show you some things about entering and changing program lines, how to add lines, delete lines and list all or part of a program.

It also started your vocabulary of BASIC words. The BASIC words used in this chapter are listed in the accompanying table. I suggest that you look through this chapter again, paying particular attention to the BASIC words, their meanings and how to use them. BASIC words and definitions in alphabetical order are in Appendix B of this book. In that appendix, look up the words listed in the accompanying table.

You can turn off your computer if you wish. The program in memory will disappear, but you don't need it anymore.

BASIC WORDS USED IN THIS CHAPTER

CONT	PRINT
END	RUN
LIST	STOP
NEW	

3 How A Program Makes A Decision

Most of us have been exposed to the idea that computers can make decisions beyond the power of the human mind. We imagine a mad scientist typing data into a computer and peering at the screen. Or, we hear about someone who programmed his computer to buy and sell stocks on the stock market. It made him a millionaire.

Computers can make decisions. This chapter demonstrates the fundamentals of decision-making. But, before the computer can make a decision, a programmer must tell it how. When running a program that you wrote, your computer can never be smarter than you are.

There are some things that a computer can do better than you or I. It works very fast. It never gets tired or bored. It rarely makes mistakes. It never forgets anything unless you tell it to, or turn it off.

A computer is a very willing and reliable worker. You can program it to do things that you probably can't do, or won't do, because you are forgetful, or become bored, or don't have two weeks to spend doing a long calculation with pencil and paper or a pocket calculator.

THE DECISION PROCESS

After you've written a program and made it work correctly in your computer, it will seem that the computer has a brain. When you run the program and type in some data, the computer appears to think. It makes decisions based on the data, and takes the appropriate actions.

Under those circumstances, the computer does have a brain—but you provided it. When you write a program, you put your own thought process into the program and into the computer. When it runs the program, it is “thinking” in exactly the same way you were thinking when you wrote the program. But, that's all it knows.

When you begin to program, you may think that the magic is in the BASIC words. It isn't. They are dull and ordinary. Programs have magic, but your mind puts it there.

To solve a problem using a computer program, you must first think through the problem to be sure you understand it. Then you figure out how to solve that problem, step by step.

Instead of looking out at the world, you will spend a lot of time looking inward into your own mind—exploring how you think and how to think logically. You will find that your mind is a very interesting place to visit.

After you have figured out how to solve the problem step by step, you then write a program telling the computer how to do it. The BASIC programming language serves you by providing a way to give the computer instructions. But, there is nothing in BASIC or in the rules of programming that will tell you how to solve a problem. You do that yourself. BASIC is merely a way to write it down.

DECISIONS

Most computer programs are a series of decisions. All decisions involve making a choice between two or more alternatives. If there are no alternatives, there is no choice and therefore no decision to be made.

Decisions in a computer are based on comparisons. Here are some examples. In these examples, X represents one number and Y represents another number.

If X is larger than Y, then choose Z.

If X is not zero, then choose Z.

If X is zero, then choose Z.

If the value of X is negative, then choose Y.

Two alternatives always provide two possible decisions. In your mind, you often state one decision and imply the other: "If it is raining, I won't wash the car." The implied decision is that you will wash the car if it is not raining.

When writing a program, you must provide all possible decisions in the program so the computer knows exactly what to do under any condition.

IF-THEN

These are BASIC words used to make decisions. IF is used to state the comparison or condition that allows a decision. For example, *IF X is not zero*. The word THEN is used to state the decision that results. For example, *THEN choose Z*.

IF and THEN are used together in a single statement on a program line.

Rules for IF-THEN Statements—When a program reaches an IF-THEN statement, it checks to see if the IF condition is true or false. If true, it executes the THEN part of that statement.

If the IF condition is not true, the program jumps immediately to the *next numbered program line*. No part of the IF-THEN statement is executed. If there is more than one statement on that program line, nothing *following* the IF-THEN statement will be executed.

COMPARISON SYMBOLS

Decisions are based on comparisons. BASIC has a set of symbols used to describe the comparison to be made:

IF X=Y says IF X IS EQUAL TO Y

IF X<Y says IF X IS LESS THAN Y

IF X>Y says IF X IS GREATER THAN Y

IF X<=Y says IF X IS LESS THAN OR EQUAL TO Y

IF X>=Y says IF X IS GREATER THAN OR EQUAL TO Y

IF X<>Y says IF X IS NOT EQUAL TO Y

The name for those comparison symbols is *relational operators*. The word *operator* means that some operation is performed. A relational operation compares the value of X to the value of Y—in other words, examines their relationship to each other.

Instead of saying that the value of X is compared to the value of Y, programmers sometimes say that a statement *tests* the value of X.

For example, the statement IF X<>0 tests X to see if it has a value other than zero. Relational operators are used to compare or test values in a program to make a decision.

Those six relational operators are typed using only three symbols: =, < and >. The symbols are on the keyboard. If your computer is turned off, please turn it on. Then find the keys that make those symbols and type them a few times so your fingers become acquainted with them.

GOTO

Sometimes you want to skip some lines in a program, jumping either forward or backward. The BASIC word GOTO does that. The statement GOTO 100 causes the computer to go immediately to line 100 and execute it.

If line 50 says GOTO 100, the computer will jump over all lines between 50 and 100. They will not be executed. Then it will continue from line 100, executing program lines in sequence, such as 100, 110 120 and so forth.

GOTO is written as a single word in BASIC. GOTO statements are often used in programs that make decisions. If there are two choices, one choice is written on one program line, such as line 100. The other choice is written on another program line, such as line 200.

When the program makes a decision, a GOTO statement sends the program to the correct line for that decision. GOTO 100 provides a path for one decision. GOTO 200 provides a path for the other decision. I will demonstrate that later in this chapter.

THE LONESOME COWBOY

Most computer books have demonstration programs. To get full benefit from the book, you must enter and run the demonstrations. When I do that, I usually think the demonstration programs are trivial, boring and too long.

In this book, some examples are trivial and probably boring. I will try to make them short. Some of the examples are useful programs that will help you understand the computer. They are longer, but more interesting.

When you enter and run a demonstration program, there are two things that you should observe. One is the programming technique or method that is being demonstrated. If I show you a statement that says GOTO 100, you may think, “Golly, that’s a statement that says GOTO 100.” Probably you will learn to write statements that say GOTO 100.

The other thing that you must know is the thinking process behind the program. If a program solves a problem, you must know what the problem is and the logic of the solution. Then the programming technique makes more sense because you can see how it is used to do something useful.

I am about to ask you to enter a demonstration program. It may relieve the boredom if you will imagine, with me, that this program does the talking for a lonely cowboy camped beside a dusty trail. The program gives a greeting from the cowboy to a party approaching on the trail.

In case you are not familiar with cowboy grammar, a single individual is addressed as *Pardner* and a group of people is addressed as *Youall*. The thinking process in this interesting demonstration is to arrange it so the cowboy gives the correct greeting.

Cowboys don’t speak in capital letters, but I am using upper case in this program so your fingers won’t have keyboard problems. Please clear the screen and enter

```
NEW
10 PRINT CHR$(147):PRINT:PRINT:PRINT
20 PRINT "HOWDY"
30 GOTO *****
40 PRINT "PARDNER."
50 PRINT "YOUALL."
60 PRINT:PRINT:PRINT
```

List the program and check your typing. Line 30 isn’t finished yet. When writing a long program, you may know that you want a GOTO statement at some line but you don’t know yet what line number you want the program to jump to. Put a string of asterisks in the unfinished line. This makes the line easy to find when you return to it later, to put in the line number.

Introduce yourself to another error message by running the program. It says there is an undefined statement error in line 30. List the program again so you can finish line 30.

Imagine that a group of people is approaching on the trail. Fix line 30 so it looks like this:

```
30 GOTO 50
```

List it again and double check line 30. Now let's follow the program flow. Refer to the listing on the screen while reading the following discussion.

Line 10 clears the screen and prints three blank lines to move the display down from the top of the screen.

Line 20 prints HOWDY.

Line 30 jumps to line 50.

Line 40 *is not executed because the program jumps over it.*

Line 50 prints YOUALL, demonstrating correct cowboy grammar.

Line 60 prints three blank lines to move the prompt symbol down on the screen.

Now run the program. It should give you a great feeling of accomplishment. Keep that program in memory and don't turn off the computer, or else save it to disk or tape. You will use it again later in this chapter.

FORMING A LOOP

In my opinion, the greatest advantage of a computer is its ability to do a simple task repeatedly without making any mistakes or becoming bored. You can tell it to count to a million three times and it will do so happily.

Repeated operations are done in a program formation called a *loop*. There are several good kinds and one bad kind. First, I will show you the bad kind. It's called an *endless loop*.

Suppose line 50 of a program says GOTO 10. When the program first begins to run, it executes line 10 and all following lines until it reaches line 50. At line 50, it is sent back to line 10 to do it all over again. Every time it reaches line 50, it jumps back to line 10 again. It will do that continuously until you stop it somehow. The loop from line 50 back to line 10 is endless.

When a program is in an endless loop, you can stop it by pressing RUN STOP to interrupt execution. Then you can list the program, fix the problem that caused the endless loop, and run the program again to see if you corrected it.

THE FRIENDLY COWBOY

You should have the cowboy program in memory. If not, enter it again or load it from disk or tape. List it. It should look like this:

```
10 PRINT CHR$(147):PRINT:PRINT:PRINT
20 PRINT "HOWDY"
30 GOTO 50
40 PRINT "PARDNER."
50 PRINT "YOUALL."
60 PRINT:PRINT:PRINT
```

When you last ran the program, it executed through line 60 and then stopped because there were no more lines to execute.

Let's prepare the cowboy to greet the next party that comes down the trail by putting in a jump back to line 20 so he can say HOWDY again.

When adding or inserting a line in a program, don't type NEW because you certainly don't want to erase the existing program in memory. Enter

```
70 GOTO 20
```

List the program and check to be sure line 70 is correct. Now run it.

The program is in an endless loop. The screen display is scrolling. When you can't stand it any longer, press RUN STOP. Execution will stop at some line number, depending on when you press

How A Program Makes A Decision

RUN STOP. The display shows the last line number that was executed. Start the program again by entering

CONT

Stop it again as you did before. It may stop at a different line number.

List the program. It loops from line 70 back to line 20 and will stay in that loop forever.

A Quick Fix—Line 35 isn't being used in this program. Let's put a STOP there to prevent the program from looping endlessly. Enter

35 STOP

Run it again. The program is still in an endless loop! I did that on purpose so you will know how it feels to be in big trouble. You are really messing up this program!

Stop execution again, using RUN STOP. Then list the program and figure out why line 35 didn't stop execution.

The program jumps from line 30 to line 50 every time it goes through the loop, so it never "sees" the STOP statement at line 35.

Complicated programs sometimes have many loops and jumps of various kinds. A common problem is jumping over a necessary statement or group of statements. When that happens, the program doesn't work. It is sometimes difficult to figure out why. You have to follow program flow step by step until you find the mistake.

If you don't see a mistake in the program, you may think that the computer made an error. Computers rarely make mistakes. Programmers make lots of them. When my flying fingers are writing a program, I can make two or three mistakes per minute. If a program doesn't run correctly, assume that there is an error in the program, not in the computer.

Delete lines 35 and 70. List the program to be sure they are gone. Save the program because you will use it again.

ASSIGNMENT STATEMENTS

When you do arithmetic in a program, you deal with quantities called *variables*. They are called that because their values may vary as the program runs.

Each variable is given a name that the computer uses to keep track of it. This is called *assigning variable names*.

Suppose you are writing a program to calculate how much money you will have in a savings account in future years. In the program, you may assign the variable name SUM to the amount of money in the account. If the account is opened with \$100, then at that time, the value of SUM is 100.

The program then calculates the value of SUM after the first year. It may be \$112.20. At that point, the value of the variable named SUM is 112.20. As the program continues to calculate future values of SUM, its value varies.

Assigning Variable Names—A name is given to a quantity by an *assignment statement*. Early versions of BASIC used the word LET to do that.

For example, LET X=2 assigns the name X to the quantity 2. Some people prefer to say that it assigns the value 2 to the variable name X, which means the same thing.

As a result of that assignment statement, the computer stores the number 2 in memory and remembers that its name is X. When you ask the computer to retrieve X, it will go to the place where it stored the number 2 and deliver it.

If you assign a different value to the variable name X, such as 3, the new value will replace the old value in memory. When you ask for X, you get the number 3.

Commodore BASIC doesn't require the word LET at the beginning of an assignment statement, but it's OK if you use it anyway. Try it by entering

LET X=2

PRINT X

Notice that this was done in the immediate mode. It does not affect the program stored in memory. The prompt symbol appeared between the two lines that you typed. Then the number 2 was displayed and the prompt symbol appeared again.

Try it again without LET.

```
X=3  
PRINT X
```

The computer changed the value of X because you told it to. It printed the number 3. It does not require using LET in assignment statements. I suggest that you not use it.

INTERACTIVE PROGRAMMING

The advantage of a BASIC interpreter is that you can quickly see what a program does and change it as needed. Then you can run it again and fix it again if necessary.

You can interrupt a program while it is running to try statements in the immediate mode. The computer responds almost instantly. You say, "Let's try this." The computer does it and says, "OK, here's the result. Now what?"

That is called *interactive programming*. It's a great aid to learning.

USING THE IMMEDIATE MODE WHILE PROGRAMMING

The BASIC interpreter is always on the job. Most BASIC statements will have the same result in the immediate mode as in the program mode. When you are writing a program and want to check a statement to see what it does, stop programming and try it in the immediate mode.

You did that a while ago, when you stopped working on your cowboy program and made a test to see if the computer requires the word LET in assignment statements. A program in memory is not affected by statements written and executed in the immediate mode.

NEVER USE A NUMBER TO BEGIN A STATEMENT

Suppose you want to assign the value 62 to the variable name X, and you are using the immediate mode. If you enter the statement 62=X, the computer thinks that you just entered line 62 of a program. The statement on that line is =X, which is not a valid statement in BASIC. If you have a program in memory, that line 62 will be added to the program, which can cause big trouble.

Later, when you run the program, it will stop and tell you that there is a syntax error at line 62.

If you are writing line 120 of a program and enter a line that says 120 62=X, the computer thinks you never give up. It stores =X as line 12062 of a program. The space between 120 and 62 is ignored by the computer.

Either of those assignment statements would work fine if written X=62. The variable name must always be first.

Variable names must begin with a letter. If you try to begin one with a number, that number may get mixed up with the line number, and you get the feeling of being in big trouble again.

Write this on the wall above your computer: *Never use a number to begin a statement.*

RULES FOR VARIABLE NAMES

When you write a long program, you will probably use lots of variable names. You choose the variable names that you want to use. There are two things to keep in mind: Variable names must follow the rules of BASIC. They should be chosen so they explain themselves and thereby help to explain the program.

Rules—Variable names can have one or two characters. If one character, it must be a letter. If there are two characters, the first character must be a letter. The second can be a letter or a number. You can use the numbers 0 through 9. These are valid variable names: A, XY, B2, B3.

In computer jargon, a combination of letters of the alphabet and numbers is called *alphanumeric*. Variable names are alphanumeric.

Variable names can have a third character, called a type-designation symbol. These symbols are not alphanumeric characters because they are not letters or numbers. They are % and \$. I mention them here so you won't be surprised later, when I tell you what they do.

How A Program Makes A Decision

When typing a variable name, you can use more than two alphanumeric characters, but the computer will recognize and use only the first two. If there is a type-designation symbol at the end of a long variable name, the computer will recognize it also.

If you use two similar variable names that are longer than two characters, you must be sure that the first two characters are different. The long variable names VIRTUE and VICE mean different things to you but they mean the same thing to the computer. Both are interpreted as the two-character variable name VI. Enter

```
EX=12345
PRINT EX
PRINT EXACTLY
PRINT EXTRA
EXTRA=678
PRINT EX
PRINT EXTRA
PRINT EXACTLY
```

A very important rule is that variable names cannot be the same as BASIC words, and they cannot contain BASIC words. It's easy to be trapped by that one. For example, enter

```
VENDOR=12
```

The computer won't let you use that variable name because it contains a BASIC word. Sometimes it is very difficult to see a BASIC word that is included in a variable name. Here is a procedure that will give you a clue when you don't see the problem. Enter

```
V=12
VE=12
VEN=12
VEND=12
```

The computer rejected VEND. That tells you that VEND includes a BASIC word ending with the letter D. The word is END.

BASIC words that cannot be used as variable names, or included in variable names, are called *reserved words*. All of the words listed in Appendix B are reserved words.

Choosing Helpful Variable Names—When you are writing a program, it is usually easy to remember what the variable names stand for and what they do in the program. Later, when you return to that program to fix or change it, it may be difficult to remember what the names stand for and what they do.

The only reason to use two-character variable names is to make the program lines shorter. Short lines take less space in memory. If your program will fit in the available memory, with a few thousand bytes left over for data storage, then there is no urgent reason to make the program shorter by using short variable names. Short program lines will execute a little faster than long lines, but the difference is usually not very much.

If you use DATE instead of just D or DA, COMPANY instead of C or CO, your programs will be easier to read, modify and fix.

DON'T CHANGE PROGRAM LINES BY ACCIDENT

For most of this book, you will enter programs and test them to see what they do. I will often suggest that you make changes to some of the program lines and then run the program again.

To make changes, you may move the cursor up into the program on the screen so you can change a program statement. This can cause two problems.

RULES FOR VARIABLE NAMES	
LENGTH	Any length, but only the first two characters plus a type-designation symbol are recognized and used by the computer.
CHARACTERS	Must be letters or numbers.
FIRST CHARACTER	Must be a letter. You can use A1 but not 1A.

One is that you forget to press RETURN with the cursor still in the program line that was changed. If you don't press RETURN, the changed line is not entered into the program in memory. It is changed on the screen, and looks just fine, but it is not changed in memory until you press RETURN.

The second problem is that you are in a hurry to run the program to see what the change did. You may leave the cursor in a program line, type RUN and press RETURN. If you do that, the program line is entered with the word RUN right in the middle of the line. The program won't run correctly. An error message will result.

Before running a program, always be sure to move the cursor down into a clear area of the screen first.

THE SOCIABLE COWBOY

You should still have the latest version of the cowboy program in memory. List it to be sure. If it isn't there, load it or go back to the last use of that program in this chapter and enter it again.

Run the program. All it does is print HOWDY YOUALL. This demonstration will put a decision point in the program, based on the number of people approaching on the dusty trail. Change line 30 and add a new line 35 so the program looks like this:

```
10 PRINT CHR$(147):PRINT:PRINT:PRINT
20 PRINT "HOWDY"
30 NUM=2
35 IF NUM=2 THEN GOTO 50
40 PRINT "PARDNER."
50 PRINT "YOUALL."
60 PRINT:PRINT:PRINT
```

List it and check lines 30 and 35 to be sure they are typed correctly. Sometimes you don't see a typing error when you enter a program line, but you do see it when you list the program.

In line 30, NUM is a variable name. It means number of people in the party. You could use NUMPEOPLE or any other legal variable name. The computer will use NU for that variable name. Line 30 establishes the number of people so the program can use that information to make a decision.

Line 35 is an IF-THEN statement used to make the decision. The computer will test the IF condition. If true, the THEN part of the statement will be executed. If not true, the next numbered line will be executed.

Obviously, the IF condition is true because line 30 made it so. Run the program. The result is a correct greeting for two people.

Keep Testing—It's a good idea to test small segments of a program as you write them. They are fresh in your mind and easier to fix at that time, if they don't work correctly.

When testing, you should test with variables in the expected range, test at the expected limits and test beyond the limits. Often, you will find defects in your logic.

How A Program Makes A Decision

You have tested when NUM is 2 and it works OK. Let's test NUM=1. To do that in this simple program, you have to change line 30. Make it read

```
30 NUM=1
```

Run the program. The cowboy gives both greetings. List the program to see what went wrong. The IF-THEN statement at line 35 does not execute. Line 40 executes, followed immediately by line 50.

The program should end immediately after line 40 executes. Here's one way to do that. Enter

```
45 END
```

Run it. The program almost works. The cowboy seems to say HOWDY PARDNER. READY. List it again. The purpose of line 60 is to move the prompt down on the screen so it doesn't seem to be part of the display. In this version, when NUM=1, line 60 doesn't execute.

One way to fix that is to change line 45 so it moves the prompt down before ending the program. Enter

```
45 PRINT:PRINT:PRINT:END
```

List it and mentally check program flow. When NUM=1, line 35 does not execute. Line 40 executes. Line 45 executes. It spaces down three lines and ends the program.

When NUM=2, line 35 executes and jumps to line 50. Line 50 executes. Line 60 executes. It spaces down three lines and the program ends because there is no more program.

The BASIC prompt symbol is moved down three lines no matter which way the program ends.

Run it with NUM=1. Change line 30 to make NUM=2. Run it again. It seems to work OK for 1 or 2 people.

Another Way—Usually, there is more than one way to write a program. Typically, you write it one way and then see a better way to do it. Please list the program.

Notice that lines 45 and 60 both do the same thing. They print three blank lines and end the program. It may be better to end the program at line 60 no matter what path the program takes to get there. Change line 40 to read like this:

```
40 PRINT "PARDNER.":GOTO 60
```

Then delete line 45 because it isn't needed. Now the program looks like this:

```
10 PRINT CHR$(147):PRINT:PRINT:PRINT
20 PRINT "HOWDY"
30 NUM=2
35 IF NUM=2 THEN GOTO 50
40 PRINT "PARDNER.":GOTO 60
50 PRINT "YOUALL."
60 PRINT:PRINT:PRINT
```

List the program and mentally check the program flow. When NUM=1, the sequence of lines that execute is 10, 20, 30, 40, 60. Run it to verify that.

Test by changing line 30 to make NUM=2. The line sequence should then be 10, 20, 30, 35, 50, 60. Run it to verify that. This version of the program seems to work OK with both of those values for NUM.

Test Unexpected Values—In this program, you have control over the value of NUM because you establish it in line 30.

In a real program, variable values are often calculated and depend on other values in the program. Sometimes they are typed in from the keyboard. Real programs are full of surprises. What if NUM is less than one? What if it is zero or a negative number?

List the program. The value of NUM is tested at only one place—line 35. If NUM has *any value smaller than 2*, line 35 will not execute. This program will give the same greeting to a party of 1, a party of 0, or even a party of -3 people!

Verify that by changing NUM to zero and running the program. This cowboy certainly would not greet zero people.

Try negative values for NUM if you wish, such as -3. To do that you need room on the program line for the - symbol. To make room, put the cursor where you plan to insert a character and then press SHIFT-INST DEL.

When you have finished testing, set NUM to zero again.

We have to fix the program. The first question is, what should the fix do? Let's decide that zero people or negative people is an illegal number. If that happens, we will just end the program by jumping to line 60. Enter a new line 34, so the program looks like this:

```
10 PRINT CHR$(147):PRINT:PRINT:PRINT
20 PRINT "HOWDY"
30 NUM=0
34 IF NUM<1 THEN GOTO 60
35 IF NUM=2 THEN GOTO 50
40 PRINT "PARDNER.":GOTO 60
50 PRINT "YOUALL."
60 PRINT:PRINT:PRINT
```

Be sure line 30 sets NUM to zero and run the program. It doesn't work very well. The cowboy says HOWDY and the program ends. We agreed earlier that this fellow would not speak to zero people. Maybe he is speaking to his horse.

All of this fussing around with a little program may seem ridiculous. When you start writing your own programs, you will discover that what is happening here is the real world. It's unusual to get a program right the first time you try. This is not a well-written program, but I think it's a good *demonstration* program because it shows a lot of things without a lot of typing.

We are testing NUM at lines 34 and 35, and we now have a problem with a HOWDY printed at line 20—before any tests are made. When a program has problems as bad as this, it is sometimes better just to start over. But, maybe we can fix it.

Desperate situations call for desperate solutions. When NUM is less than 1, let's clear the screen and then end the program. Change line 34 so the program looks like this:

```
10 PRINT CHR$(147):PRINT:PRINT:PRINT
20 PRINT "HOWDY"
30 NUM=0
34 IF NUM<1 THEN PRINT CHR$(147):END
35 IF NUM=2 THEN GOTO 50
40 PRINT "PARDNER.":GOTO 60
50 PRINT "YOUALL."
60 PRINT:PRINT:PRINT
```

Run it that way. The desperate remedy worked as planned, but there is still a serious problem. By looking at the program, you can see that it will fail when NUM is any number greater than 2.

Change line 30 so NUM is 3. Mentally follow the program flow. Lines 34 and 35 will not execute. The cowboy will say HOWDY PARDNER. to a group of three people. In fact, he will say that to an army or a whole tribe of Indians. Run it to verify that.

This fix is easy. The cowboy should give the same greeting to any party with more than one person. Change line 35 to read

```
35 IF NUM>1 THEN GOTO 50
```

Run it. I think the program is now bulletproof as far as the number of people is concerned. Verify that by trying a variety of numbers for NUM. Don't use commas when typing the numbers. Keep the numbers below 999999999, or you will encounter a difficulty that I haven't discussed yet.

How A Program Makes A Decision

When you have finished, list the program again and look it over. It is no longer a simple program, and it does not illustrate good programming. It does illustrate a variety of programming methods, and it gave you some experience with relational operators and program logic—both good and bad.

Depending on program flow, it can clear the screen at two places in the program. It may end at line 34 or at line 60.

The variable NUM is tested at two places, using relational operators. There are three possible results of running the program. It may end with a clear screen, it may print HOWDY PARDNER., or it may print HOWDY YOUALL.

The program responds to three different conditions: NUM<1, NUM=1 and NUM>1.

EXPLICIT AND IMPLICIT TESTS

Explicit tests are written explicitly in the program. You can see the program lines that perform explicit tests.

Implicit means *implied*. Implicit tests are invisible in the program, but are there anyway. In this program, the test for NUM<1 is on line 34. It is stated explicitly. The test for NUM>1 is explicitly stated on line 35.

Where is the test for NUM=1? It's implied by lines 34 and 35. The test for NUM=1 is that neither line 34 or line 35 executes. The program is written to provide the correct result for NUM=1 when neither line 34 nor line 35 executes.

When there are three possible conditions to be tested, you can test all three if you want to. But it is necessary to test only two of them. If NUM is not less than 1, and it is not greater than 1, then it must be 1.

Implied tests are common in computer programs. Please save the cowboy program or keep it in memory.

REM

When you are writing a program, everything about it is clear in your mind. You think that you'll never forget the clever things you did. You will.

REM means *remark*. You can put remarks in your program to explain how it works when you or another programmer look at it at some later time.

Because remarks are information about the program, rather than program statements, they should not be executed. The BASIC word REM is used to tell the computer that a remark follows. When the computer sees REM in a program line, it stops executing that line immediately and proceeds to the next numbered line.

You can use a program line just for a remark, with nothing else on it. Or, you can put program statements and a remark on the same line. If you write a line that has a statement, then a remark, then another statement, the last statement will never be executed. The computer stops executing a program line when it finds the word REM.

If a program line has one or more statements and a remark, the remark must be the last thing on the line. Remarks make a program longer and take space in memory. They slow down execution a little bit.

Don't use a remark to explain what a program line does. That is obvious from the program. Use remarks to state the overall purpose of a program, to explain the logic, to state the meanings of variable names, and to explain other things that are not obvious.

There is one place in the cowboy program where a remark may be useful. List the program if it is not on the screen. Add this remark to line 30:

```
30 NUM=3: REM NUMBER OF PEOPLE
```

You may have NUM set to some value other than 3. If so, it doesn't matter. When a remark follows a program statement, a colon is used to separate them. If it is the only thing on the line, the word REM follows the line number, without a colon.

If this were a real program, you should put a remark at the beginning to state its title and purpose, the name of the author, the date it was last worked on, and any other useful information

that will be helpful later. You could use line 5 for that. Keep the cowboy program in memory or save it.

TYPE-DESIGNATION SYMBOLS

You have been using NUM as a variable name for the number of people. At line 30, you have been assigning numeric values to that variable name. Because it receives numeric values, it is called a *numeric* variable name.

You can also assign a name to a string variable. Then, when you want to print that string, you just ask the computer to print the string variable name. If the program prints a string more than once, using a variable name to represent the string shortens program lines. This is because you don't have to type the entire string each time you want to use it. You just type the name of the string, which can be as short as two or three characters.

I mentioned earlier that both you and the computer must know if a variable is a string or a numeric. When strings and numerics are typed directly on program lines, quotation marks are used to enclose strings. Therefore the type of variable is obvious.

When a variable is represented by its variable name, the variable name itself must tell the computer which type of variable it represents. This is done by type-designation symbols used at the end of variable names. There are three type-designation symbols: \$, % and no symbol at all.

TYPE-DESIGNATION SYMBOLS	
\$	Literal string, such as "HELLO"
%	Integer or whole numbers, such as 6 or 179
No symbol	Decimal number, such as 10.3 or 12345.678 (Also called <i>floating-point</i> number)

String Variable Names—When a variable name is used to represent a string, it must have the symbol \$ at the end of the variable name. If not, the computer assumes that the name represents a numeric variable. It will attempt to evaluate the variable mathematically, and your program won't work correctly. Enter

```
GREETING$="HELLO"
```

That is an assignment statement. It assigns the string variable "HELLO" to the string variable name GREETING\$. The symbol \$ is pronounced *string*. The string variable name GREETING\$ is pronounced *greeting string*. Enter

```
PRINT GREETING$
```

The string variable "HELLO" was placed in memory under the string variable name GREETING\$ by the preceding assignment statement. It can be called from memory and displayed by a PRINT statement using that variable name. Enter

```
PRINT GR$
```

Because the computer actually uses only the first two characters of a variable name, plus the type-designation symbol at the end, printing GR\$ has the same result as printing the long variable name GREETING\$. Enter

```
PRINT GR
```

The variable name GR is not the same as the variable name GR\$. Because no value has been assigned to GR, that statement displays a zero.

How A Program Makes A Decision

Integer Variable Names—The word *integer* means a whole number. The type-designation symbol for integer variable names is %. When used as a type-designation symbol, % is pronounced *integer*. NUM% is pronounced *num integer*.

The variable NUM% will accept only integers, such as 11, 3 or 1230.

Numbers with a decimal fraction are called decimal numbers. 11.7 is a decimal number. It has two parts. The whole-number part is 11. The decimal fraction is 0.7.

If a decimal number happens to have a decimal fraction of zero, it is the same as an integer. The number 10.0 has the same value as the number 10.

If you assign a decimal number to an integer variable, it changes the decimal number to an integer. For positive decimal numbers, it ignores the decimal part. Converting the decimal number 123.45 into an integer gives 123 as the result.

For negative decimal numbers, the integer value is the next smaller whole number. The number -11 is the next smaller whole number below -10.4.

For a technical reason, the largest integer number the computer will accept is 32767. The smallest is -32768. If you try to use numbers outside those limits, the computer displays an error message. Try these statements:

```
NUM%=10
PRINT NUM%
PRINT NU%
NUM%=10.4
PRINT NUM%
NUM%=-10.4
PRINT NUM%
```

Decimal-Number Variable Names—A decimal-number variable name will accept decimal numbers. It will also accept integers. It considers an integer to be a decimal number with a decimal part that is zero.

The type-designation symbol for decimal-number variable names is no symbol at all. X\$ is a string variable name. X% is an integer variable name. X is a decimal-number variable name.

Very large decimal numbers, such as 9999999999999.999, are displayed in a different way from ordinary decimal numbers. The method of displaying very large and very small decimal numbers is called *floating-point*. For this reason, Commodore literature uses the term *floating-point variable name* to mean the same thing as decimal-number variable name.

There is a limit to the largest and smallest decimal numbers that the computer can use. Floating-point notation and the limits for large and small numbers are discussed in the chapter on mathematics. The following statements demonstrate the fundamental facts about decimal-number variable names. Enter

```
NUM=10.4
PRINT NUM
PRINT NU
NUM=11
PRINT NUM
NUM=-12.12
PRINT NUM
```

USING TYPE-DESIGNATION SYMBOLS

A type-designation symbol is a permanent part of each variable name. Every variable name has a type-designation symbol. Even if you don't use one, you use one, because no symbol *is* a symbol. If a variable name is typed without a symbol, it is automatically a decimal-number variable name.

Type-designation symbols are both necessary and helpful.

String Type Designation—The string symbol, \$, is necessary to tell the computer that the variable is a literal string. Otherwise, the computer will try to evaluate it as a number.

Integer Type Designation—The integer symbol, %, is used for two reasons. Integer numbers use less space in memory. The more important reason is that your programs should make sense.

If a number should be an integer, using the integer type designation causes the computer to accept only an integer.

Please list the cowboy program. At line 30, the decimal-variable name NUM is used. If you write line 30 to say NUM=10.3, the computer will not object. But, 10.3 people is an absurdity. To prevent that program from dealing in absurdities, change the variable name in line 30 to NUM%.

When you have done that, list the program and look at it carefully. It won't work. The problem is that NUM% is established at line 30 but NUM is tested at lines 34 and 35. NUM is not the same as NUM%. NUM will be zero because the computer has no value for NUM. Line 30 established a value for NUM%.

The fix is to change NUM to NUM% everywhere in the program. Please do that. Now the program will work. It will work even if you set NUM% equal to 10.3 people. But if you ask the computer to display the value of NUM%, it will tell you that there are only 10 people because NUM% accepts only the integer part of a decimal number.

When your program is handling quantities that should be integers, use integer type designations. If you write a program that does a calculation and then reports that there are 1000.3 aircraft flying over the ocean, hardly anybody will believe it.

Decimal-Number Type Designation—The decimal-number, or floating-point, type designation is used when your program should keep track of decimals.

If you write a program to balance your checkbook, or figure the amount of a customer's bill, it should be accurate to the nearest penny. If your program gives a customer a bill for \$11 when he spent only \$10.53, he won't like it. Also, he won't pay it.

When dealing with money, or any quantities that should be decimal numbers, use decimal-number variable names.

LOGICAL OPERATORS

A comparison of two values is done by a *relational operator* such as >. There is another type of operator that you haven't met before. It's called a *logical operator*. In Commodore BASIC, there are three logical operators: AND, OR and NOT. They are BASIC words.

The logical operator NOT is used for a purpose that I will discuss in a later chapter.

AND and OR can be combined with relational operators to make fancier comparisons in IF-THEN statements. When used in comparisons, their meanings in BASIC are the same as in English. Enter these examples:

```
X=3
```

```
Y=9
```

```
IF X=3 AND Y=9 THEN PRINT "YES"
```

That IF-THEN statement has two tests. Because of the AND operator, it will execute only if both tests are true. Enter the following statements. Remember that X is 3 and Y is 9. Compare the results to the tests in each statement.

```
IF X>2 AND Y=9 THEN PRINT "YES"
```

```
IF X>1 AND Y>5 THEN PRINT "YES"
```

```
IF X>0 AND Y<11 THEN PRINT "YES"
```

```
IF X=3 AND Y<5 THEN PRINT "YES"
```

The last statement does not execute because Y is not less than 5. One of the tests is true and the other false. With an AND operator, both tests must be true.

How A Program Makes A Decision

The OR operator requires either one OR the other to be true. Remember that X is 3 and Y is 9.
Enter

```
IF X=3 OR Y=9 THEN PRINT "YES"
```

That statement executes because both tests are true. One true test is enough.

```
IF X=3 OR Y=5 THEN PRINT "YES"
```

That statement executes because one of the tests is true.

```
IF X=1 OR Y=0 THEN PRINT "YES"
```

That statement does not execute because neither of the tests is true.

Relational and logical operators give you a lot of control over what happens in a program. Sometimes it takes careful planning and program testing to get these operators to do what you want them to.

REVIEW

I lured you into this chapter by promising to discuss decision-making. The chapter did that and a lot more. Please look over the chapter again to be sure you understand it and will remember what it says. There is a lot to know and remember about writing programs.

When you have a small vocabulary of BASIC words, it becomes easier and easier to add to it. The last version of the cowboy program does a lot of things, using only a few BASIC words. The more you know, the more you can do.

The accompanying table lists BASIC words and symbols used in this chapter. Please look up the BASIC words in Appendix B of this book. You can turn off your computer now, if you wish. Say *adios* to the cowboy.

BASIC WORDS AND SYMBOLS USED IN THIS CHAPTER	
AND	
GOTO	∇
IF	∇
LET	∇
OR	
REM	∇
THEN	∇

4 Loops

A loop occurs when a program jumps back to a smaller line number and repeats part of the program. In the last chapter, you saw that endless loops aren't very helpful. In my opinion, loops that work correctly are the most useful part of computer programming. This chapter will show you some good loops.

MORE ABOUT RUN AND GOTO

Here are two facts about RUN and GOTO statements that you will need to know later in this chapter. A RUN statement clears everything out of the computer memory except the program itself. The memory cleared is RAM, the random-access memory that holds programs and data such as variable values.

When you execute a program by a RUN statement, the program begins operation with nothing in memory except the program itself. To demonstrate that, enter and run this one-line program:

```
NEW
10 PRINT X
```

When you run it, it prints zero for the value of X because no value has been assigned to X. Now enter the following statements from the immediate mode:

```
X=4
PRINT X
```

The computer has the value 4 in memory with the variable name X. Run the program again. It still prints zero for the value of X. Entering a RUN statement clears memory, except for the program itself. To verify that, enter

```
PRINT X
```

The value of X is now zero even though you assigned the value of 4 to X before you ran the program. List the program. The program is still in memory, but data such as the value of X is gone. Executing a RUN statement did that.

Another way to run a program is with a GOTO statement. The program will begin execution at the line number specified by the GOTO statement. A GOTO statement does not clear memory. Enter

```
X=7
GOTO 10
```

The program runs from line 10 and prints 7 as the value of X. Memory was not cleared by the GOTO statement.

Using a GOTO statement to run a program from any desired line number is useful when troubleshooting or testing a program. It allows you to put values into memory that the program can use and then see what the program does with them.

COUNTERS

The way to prevent a loop from running endlessly is to provide some way to control the number of times that it operates. One way is to count the number of loop operations and stop when the desired number has been reached. Then, the program moves on and does something else. Here's an example for you to enter:

```
NEW
10 PRINT CHR$(147):REM CLEAR SCREEN
20 C=1:REM SET COUNTER
30 PRINT C:REM PERFORM AN OPERATION
40 C=C+1:REM INCREMENT COUNTER
50 STOP
60 GOTO 30:REM LOOP BACK AND DO AGAIN
```

List it and check your typing. I put in some remarks to explain the program.

Initializing a Variable—The number of operations of this loop is counted by a counter named C. When you are using a counter, you should set the starting value, so you know that the count is correct and what the count means. Usually, you will start with the counter set to zero or one.

In line 20 of this program, the counter is set to start with a value of 1. That is called *initializing a variable*, which means setting it to an initial value.

Useful Operation—A loop performs some useful operation and can do it more than once. Line 30 performs a useful operation by displaying the value of the loop counter. That isn't much, but it is useful because it shows how loop counters work.

Increment the Counter—The word *increment* means to increase or add to. In programming, it usually means adding a fixed amount, such as 1.

At line 40, the loop counter is incremented by 1 each time the loop operates. Line 40 counts loop operations by incrementing the counter.

Stop—The stop at line 50 is just to stop the loop once during each operation. It would not be used in a practical loop in a real program.

Jump Back and Do It Again—The essential part of a loop is a line that sends the program back to the beginning of the loop. Line 60 does that.

Run the program. Line 30 displays the value of the loop counter. The program stops at line 50 and tells you so. Start it up again by entering CONT. Line 60 jumps back to line 30 to cycle through the loop again. At this point, the loop counter has been incremented. Line 30 displays the count, which is 2. Enter CONT and run the loop again.

LOOP BOUNDARIES

Every loop has a beginning and an end—a top and a bottom. Please list the program in memory. The bottom of a loop is easy to identify because it sends the program back to the top of the loop. The bottom of the loop is line 60.

When you have identified the bottom of the loop, the top is easy to find because that's where the loop jumps to operate another time. The top of this loop is line 30.

All lines between the top and bottom of a loop are executed on each pass through the loop—unless there is something on a program line that says *don't execute this line*.

For example, if a program line that is inside a loop is an IF-THEN statement, it may or may not execute. If the IF condition is true, it will execute.

When you initialize a counter, it should be done before the beginning of the loop. Initializing is done only once.

HOW FAST CAN A LOOP RUN?

It depends on two things: the speed of the computer and the amount of data processing done inside the loop. Some computers operate faster than others, which will make loops run faster.

This loop doesn't do much data processing inside the loop. It does two operations: It prints C and then increments C.

To give you an idea of how fast a simple loop will run on a Commodore 64, delete line 50. This converts the program into another endless loop. Run the program. You can see how fast the value of the counter changes. To stop it, press RUN STOP.

SETTING THE NUMBER OF OPERATIONS

When a loop is controlled by a counter, one way to stop looping and move on is to test the value of the counter at each pass through the loop. When the loop has completed the desired number of operations, the program jumps out of the loop. To see that happen, enter these two lines. They will become part of the program.

```
50 IF C=10 THEN GOTO 100
100 PRINT "END OF LOOP":PRINT:PRINT:PRINT
```

List the program. Line 50 is now a *decision point*, sometimes called a *branch point*. It tests the value of C, using an IF-THEN statement. The loop will run just as it did before until C is 10. When C is 10, line 50 will execute and jump out of the loop to line 100, which ends the program.

Run it. The loop prints the numbers 1 to 9 and then stops. You may have expected it to print 1 to 10. List it again.

THE ORDER OF EVENTS

Three important events in a loop are the useful operation, incrementing the counter, and the decision to stop looping. In this loop, those things happen at lines 30, 40 and 50. These events can be in a different order. If so, the loop operates differently.

Consider program flow when C=9. Line 30 prints the number 9. Line 40 increments it so its value becomes 10. Line 50 tests the value of C, finds that it is equal to 10 and jumps immediately to line 100. Even though the loop counter reaches a value of 10, this loop operates only 9 times.

Run it again. You can see that it operated only 9 times. It printed the numbers 1 to 9 and stopped. To check the value of C, enter

```
PRINT C
```

Suppose you want the loop to operate 10 times. List the program. Change line 50 to jump out of the loop when the counter reaches 11. Make it look like this:

```
50 IF C=11 THEN GOTO 100
```

Run it that way. It operates 10 times. Check the value of C. It is 11.

Another way is to change the order of events in the loop. As written, it increments C before it tests C. If it tested C before incrementing, operation would be different.

Reverse lines 40 and 50. Overtyping 50 on line number 40 and pressing RETURN. The cursor moves down to the old line 50. Then overtyping 40 on old line number 50 and pressing RETURN. Using the down-arrow cursor key, move the cursor to a blank line at the bottom of the screen. List the program. Those lines are reversed.

Run it. Now, it operates 11 times, which is one too many. Change line 40 to jump out of the loop when the count is 10. Run it. It operates 10 times. List it again.

When the loop counter is incremented *before* the branch point, the number of loop operations is the same as the value of the counter—provided the counter was initialized to 1.

How many times would this loop operate if the counter were initialized to 0 instead of 1? Try it.

COUNTING BACKWARD

Occasionally, you will need a loop that can count backward. Here's one that does that. Please enter

```
NEW
10 PRINT CHR$(147)
20 C=33
30 PRINT C
40 IF C=27 THEN GOTO 90
50 C=C-1:GOTO 30
90 PRINT:PRINT:PRINT
```

List it and check your typing. Line 20 initializes the counter. Line 30 represents a useful operation. Line 40 is the branch point. Line 50 *decrements* the counter and jumps back to the top of the loop. Decrement is the opposite of increment.

How many times will this loop operate? Run it to verify your conclusion.

I usually use the order of events shown in this loop: Perform the useful operation. Test the counter to jump out of the loop when the count reaches the desired value. Increment or decrement the counter and jump back to the top of the loop.

USING OTHER INCREMENTS

You have incremented a loop counter by +1 and -1. Any increment value can be used. To count by twos, increment by 2. You can increment by 1.5 or -1.3, or whatever you need to use.

If you do that, be sure the program line that tests the counter is looking for a value that the counter will actually reach. For example, if the counter is always an even number, a line that says IF C=5 will never execute. If the line says IF C>5, it will stop loop operation when C is 6.

EXITS

In addition to a top and a bottom, a loop has one or more exits—unless it is an endless loop with no way out. Please list the program in memory. The only exit from this loop is at line 40. Each time line 50 executes, it sends the program back to 30.

FOR-NEXT LOOPS

This is a more elegant loop that uses a counter inside the computer instead of a counter in the program itself. It uses the BASIC words FOR and NEXT. Please enter

```
NEW
10 PRINT CHR$(147)
20 FOR C=1 TO 5
30 PRINT C
40 NEXT C
50 PRINT "END OF FOR-NEXT LOOP"
60 PRINT:PRINT:PRINT
```

Line 20 sets up the loop operation. It begins by naming the counter. This program uses the variable name C, but any *decimal-number variable name* can be used. An integer variable, such as C%, is not allowed by Commodore BASIC. Using it will produce a syntax-error message.

The FOR statement automatically initializes the counter to its starting value. If the statement says FOR C=1 TO 5, the starting value of C will automatically be 1.

The FOR statement sets the range of values that the counter can have. In this example, the counter will start at 1. The last operation will occur when the counter is 5. This loop will operate 5 times.

The FOR statement sets up the *specifications* for the loop operation. The computer stores those

specifications in memory. The loop counter is maintained in memory and operated by the BASIC interpreter.

Line 30 represents the useful operation provided by this loop.

Line 40 controls loop operation by referring to the loop specifications in memory. Each time the program reaches line 40, the computer increments the counter. Then it tests the value of the counter against the range of values established by the FOR statement.

If the counter value is *not greater* than the maximum value specified in the FOR statement, the loop continues to operate. The NEXT statement sends the program back to the statement *immediately following the FOR statement*. That statement can be on the same program line or on the next numbered program line.

In this example, the NEXT statement will increment the counter and send the program back to the top of the loop until the counter reaches 6. When that happens, the counter value will be greater than the maximum value specified by the FOR statement. The loop will stop operating.

Instead of jumping back to the top of the loop, the statement *immediately following the NEXT statement* is executed. That statement can be on the same program line as the NEXT statement, or it can be on the following line.

The FOR statement at line 20 is not part of the loop. It is executed only once to set up the loop specifications in computer memory. Line 30 is the top of the loop. Line 40 is the bottom of the loop. In this example, there isn't much loop.

Run it and then list it again. It prints the values of the loop counter from 1 to 5, as expected. Then it leaves the loop, moving from line 40 to line 50. At line 50, it prints a message. Line 60 spaces the prompt down and the program ends.

Check the value of the loop counter by entering

PRINT C

BASIC allows you to omit the name of the loop counter in a NEXT statement. This program will run the same way if line 40 says NEXT instead of NEXT C. Try it.

With uncomplicated programs, you can save a little space in memory by omitting the name of the counter. If the program is complicated, with several FOR-NEXT statements, the computer still doesn't require that the counter be named after each NEXT statement, but you may prefer to do it anyway. It makes programs easier to read and troubleshoot.

EXITS

A FOR-NEXT loop may have more than one exit. As written, the program now in memory has only one way out. It must finish the number of executions specified by the FOR statement and then leave the loop through line 40. When a FOR-NEXT loop does that, we say it has *run to completion*. Leaving the loop through the NEXT statement after the loop has run to completion is called a *normal termination* of the loop.

FOR-NEXT TRICKS

A FOR-NEXT loop is very easy to set up and use. It is more versatile than the preceding demonstration shows.

Starting at any Counter Value—You don't have to start counting at 1. The FOR statement starts counting wherever you tell it to. Try this:

```
20 FOR C=12 TO 17
```

Counting in Steps—You don't have to count in steps of 1. The computer will automatically use steps of 1 unless you tell it otherwise. Set the step value like this:

```
20 FOR C=10 TO 70 STEP 10
```

Run it that way. Notice that STEP is not a separate statement. It is part of the FOR statement.

The step value does not have to be an integer. It can be 1.3 or 0.27 or any other number with a decimal fraction. In Commodore BASIC, the loop counter in a FOR-NEXT loop must be a decimal-number variable, such as C.

If the STEP value is not 1, the value of the counter during the last operation of the loop may be less than the maximum specified value. The counter value after the last operation must always be greater than the maximum value specified. To demonstrate that, enter

```
20 FOR C=1 TO 5 STEP .7
```

Run the program. The last loop operation occurs when the counter is 4.5. When the program ends, display the value of the counter by entering

```
PRINT C
```

It is 5.2, which is $4.5 + .7$.

Default Values—Some statements have a “built-in” value used automatically unless told otherwise. This is called the *default value*. In a FOR statement, the default STEP value is +1 unless you supply some other value.

Counting Backward—To count backward, use a negative STEP value. In the FOR statement that sets up the loop specifications, the maximum value of the counter is stated first, followed by the minimum value.

The loop will run until the counter value is *less than the minimum value* specified by the FOR statement. Try running the loop with the following FOR statements. Display the value of the counter after each complete execution of the loop by entering PRINT C

```
20 FOR C=100 TO 10 STEP -10
```

```
20 FOR C=100 TO 10 STEP -11.5
```

A FOR-NEXT LOOP WILL ALWAYS OPERATE AT LEAST ONCE

The computer will always execute a FOR statement and proceed through the loop to the NEXT statement—unless the FOR statement or one of the statements in the loop has a syntax error.

If the specifications set up by the FOR statement are impossible to follow, the computer isn’t aware of the problem until the NEXT statement executes. Therefore, a FOR-NEXT loop will always operate once. Enter

```
20 FOR C=100 TO 10 STEP +5
```

Obviously, something is wrong with those specifications. The computer can’t count from 100 down to 10 by adding +5 to the starting value of the counter.

Run it. It executes one time. At the first execution of the NEXT statement, the computer notices that something is wrong and stops the loop. What is wrong is that the counter value is incremented to 105, which is outside the specified range of the counter.

USING VARIABLES TO CONTROL A LOOP

So far, you have typed numbers into a FOR statement to set limits for the counter. You can use variables, such as X and Y, for the limits. To do that, the computer must have values for X and Y already in memory when loop operation begins.

The program can *calculate* values for X and Y and then run the loop from X to Y. To simulate that, let’s give values to X and Y by assignment statements and then run the loop. Enter

```
15 X=5: Y=12
```

```
20 FOR C=X TO Y
```

Run it. It operates the same as if the numbers 5 and 12 had been typed into the FOR statement.

Plugging in Values from Immediate Mode—You can assign values to variable names from the immediate mode and run the program without losing the values—if you use a GOTO statement to run the program. This is a good way to test a program. Enter

```
X=15
```

```
Y=25
```

```
GOTO 20
```

Choose the starting line number carefully. What would happen if the GOTO statement ran the program from line 15 instead of line 20? Line 15 would set different values for X and Y in memory. The loop would run from 5 to 12 again. Try it.

If the program has never run before, the FOR statement has not executed, and the loop specifications are not in memory. If you set values for X and Y in the immediate mode and then start at a line number greater than 20, a program error will occur and execution will stop. The error message will say that the program found a NEXT statement without first executing a FOR statement.

If a program is running, and has executed the FOR statement, you can stop it and change the value of the counter in the immediate mode. Then you can start it again by a GOTO statement that enters the loop behind the FOR statement. The program will run normally, resuming with the new value of the counter that was set from the immediate mode.

FOR AND NEXT STATEMENTS MUST BE IN MATCHED PAIRS

Every FOR statement must be matched with a NEXT statement. If not, the program won't work correctly. Delete line 40 and run the program.

List it. The program enters the loop and prints the first value of the counter. Because it doesn't find a NEXT statement, there is nothing to send it back to the top of the loop. From line 30, it executes lines 50 and 60.

There is no error message for that problem. The program just doesn't work.

If the program finds a NEXT statement without first executing a matching FOR statement, it stops and displays an error message.

If you use the name of the loop counter in a NEXT statement, it must match the name of the counter in the FOR statement. To meet a new error message, enter

```
40 NEXT Z
```

Run it. List it. There is no FOR statement using Z as the counter. This is a NEXT WITHOUT FOR error. Notice that the loop operated one time even with an error in the NEXT statement.

If the program doesn't have a FOR statement, the same error message results. Delete line 20 and run it again. Notice that it printed zero at line 30. Without a FOR statement, the value of the counter is not established.

A LOOP ON ONE PROGRAM LINE

Simple loops can be written on one program line. Enter

```
NEW
```

```
10 FOR I=1 TO 10: PRINT I: NEXT
```

Run it. It works the same as if it were on three lines. Don't put a loop on a single line if the loop has an IF-THEN statement in it, followed by the NEXT statement. If the IF-THEN statement doesn't execute, the computer won't see the NEXT statement and the program won't work correctly.

Loops are very powerful. Many programs require repeated operations, which are best done in a loop. In this program, 10 operations are done by only one line. It could do 1000 operations, or a million!

STANDARD LOOP-COUNTER NAMES

It is often a good idea to use a name for the loop counter that helps explain the program. If it is counting apples, then APPLES might be a good name for the counter.

In mathematics, the word *iteration* is used to mean a repeated operation. Some programmers use the letter I as a loop counter, thinking that I is the number of iterations.

Complicated programs often use several loops with different counter names. After using I for the name of a loop counter, the letters J and K are often used for following loops. Because these letters are in alphabetical order, they may indicate the order of the loops in the program.

By habit and custom, the letters I, J and K are commonly used as loop counters. It is usually

better to use counter names that mean something and help explain the program, rather than using I because other programmers use it.

A LOOP THAT DOES NOTHING

Sometimes, doing nothing is worthwhile. Enter

```
NEW
10 PRINT "START"
20 FOR DL=1 TO 500: NEXT
30 PRINT "FINISH"
```

Run the program. The loop on line 20 doesn't do anything except run. Its purpose is to create a delay, perhaps to slow down a display so it can be read more easily. In a delay loop, I use DL for the counter. DL means delay. You can change the amount of delay by changing the maximum value of the counter.

LOOPS IN THE IMMEDIATE MODE

You can run a loop in the immediate mode. But you have to enter all of the statements as one line. This requires typing them with colons separating the statements, and then pressing RETURN. Try this:

```
FOR I=1 TO 10: PRINT I+2: NEXT
```

This loop does some arithmetic as it runs. On each pass, it adds 2 to the counter value and prints the result.

TERMINATING LOOP OPERATION

Often you will use a loop to do something and then stop loop operation when the desired action has been accomplished. For example, you can use a loop to search through a list of names, looking for a specified name. When that name has been found, there is no reason to search further, so you stop the loop.

Stopping a loop when it has done its job makes programs run faster—sometimes significantly faster. It is done by a comparison or test. Please enter

```
NEW
10 PRINT CHR$(147)
20 FOR I=1 TO 15
30 PRINT I
40 IF I=7 THEN GOTO 100
50 NEXT
100 PRINT:PRINT:PRINT
```

This loop now has two exits, one at line 40 and the other through line 50. If line 40 executes, the program jumps to line 100 and loop operation stops immediately. Of course, line 40 will execute in this program because it is just testing the value of the counter, and the counter will reach a value of 7.

If this loop were looking for the name Jones in a mailing list, line 40 would be asking *is this name Jones?* If Jones is found, the loop stops operating. If not, the loop runs to a normal termination.

I imagine that FOR-NEXT loops have a front door, a back door and one or more side doors. The front door is the top of the loop. It is the entrance.

If the loop runs to a normal termination, the exit is through the back door—the NEXT statement. Some people say that the program *flows through* the NEXT statement when the loop operation ends.

If operation is terminated before the specified number of executions has been made, the exit is through the side door—line 40 in this example. That is not a normal termination of the loop.

To see a side-door exit, run the program. When I becomes 7, the program jumps out of the loop to line 100.

To see a normal termination, change line 40 like this:

```
40 IF I=19 THEN GOTO 100
```

Run it. Line 40 doesn't execute because the value of I never reaches 19. The loop makes the specified number of executions and exits through line 50.

It is usually not a good idea to jump out of a FOR-NEXT loop without running it to a normal termination. When that happens, the loop specifications remain in memory—at a special location. Eventually, this location will fill up with residue from unfinished loop specifications. An error message results. It says **OUT OF MEMORY**. Actually, only the part of memory that stores loop specifications is full.

A program can jump out of FOR-NEXT loops a few times without bringing the loops to a normal termination, and it will run OK. With a very simple program, the limit is nine abnormal terminations. In a complicated program, the limit may be less.

That precaution applies only to FOR-NEXT loops. If the program supplies the loop counter, you can jump out of the loop any time you wish.

BRINGING A FOR-NEXT LOOP TO NORMAL TERMINATION

One way to bring a loop to a normal termination is to let it run the number of times specified in the FOR-NEXT statement. You can still use the loop to search for a name or a value and make a note that it was found.

In the following example, the loop is looking for a value of J. List the program in memory and change it to look like this:

```
10 PRINT CHR$(147)
15 FLAG=0
20 FOR I=1 TO 15
30 PRINT I
35 J=I*3
40 IF J=9 THEN FLAG=1
50 NEXT
60 IF FLAG=1 THEN PRINT "FOUND J=9"
100 PRINT:PRINT:PRINT
```

Line 15 initializes a FLAG that will be tested later. Line 35 is used just to provide something to "look for." As the loop runs, line 40 tests each value of J, looking for a 9. When J is 9, line 40 sets the FLAG to 1. The loop doesn't stop. It runs to completion. The fact that the FLAG was set is a way of making note that J reached a value of 9.

When the loop has run to completion, line 60 tests the FLAG to see if J=9 was found. Run the program. Check the value of I by displaying it. It is 16. The specifications for that loop are no longer held in memory because they aren't needed any more.

If you are running that loop just to find J=9, or the name JONES in a mailing list, there is no point in continuing to run the loop after it is found. Here is a way to stop the loop but still bring it to a normal termination. Change line 40 like this:

```
40 IF J=9 THEN FLAG=1: I=15
```

List the program. When J=9, line 40 does two things. It sets a flag to make a note of that. Then, it sets the loop counter to its maximum value. Then, line 50 tests the value of I to see if the loop should run again. It increments I and compares its value to the loop specification. I will be 16. The specification says to stop at 15. Loop operation is ended.

Run the program. The loop ran only three times until it found J=9. Print the value of I. It is 16. The loop ran to a normal termination.

GOSUB

Sometimes, when you have a loop running in a program, you will want to leave the loop temporarily to do something else. When the “something else” has been done, you want to return to the loop and resume loop operation where it left off. You can do that with a *subroutine*.

GOSUB is similar to GOTO except that GOTO jumps to another location in the program and the program runs from there. GOSUB will jump to another location, execute a subroutine, and return to the place it jumped from.

A subroutine is usually executed several times in the program. By jumping to it, and then returning to the main program, you can use the subroutine often but type it only once.

This demonstration shows the program structure for a subroutine. Enter

```
NEW
10 PRINT CHR$(147)
20 FOR I=1 TO 10
30 GOSUB 200
40 PRINT"RETURNED"
50 NEXT
60 END
190 REM SUBR TO PRINT I
200 PRINT"THE VALUE OF I IS" I
210 RETURN
```

Subroutines are called by a GOSUB statement that uses the first line number of the subroutine. The program jumps immediately to that line number. The subroutine will execute and then return to the main program at the *next statement following* the GOSUB that called it.

In this program the subroutine is called at line 30. The next statement that can be executed is line 40. As the program runs, line 30 calls the subroutine, the subroutine executes and returns to line 40.

Subroutines have three parts: a beginning line number that serves as the address for the subroutine, one or more program statements that do something, and a RETURN statement.

This subroutine begins at line 200. The statement that does something is also on line 200. All it does is print the value of the loop counter. The RETURN statement is at line 210. The RETURN statement is always the last statement in the subroutine. It jumps back to the main program.

Run the program. It doesn't do much, but it gives you the idea of a subroutine.

The GOSUB statement must jump to a line number that exists in the program. It would run the same way if the GOSUB statement jumped to line 190 instead of 200. Some programmers put remarks into a program while they are writing it, to identify segments, and then take the remarks out later. If a GOSUB jumps to a remark, and that line is deleted, the program fails. It is best to call a subroutine using the first “working” line number, rather than a line that has only a remark.

Every subroutine must have a RETURN statement. Otherwise, it won't return. The program will keep running just as though the GOSUB had been a GOTO.

Although this demonstration shows a subroutine being called repeatedly from a loop, you can call a subroutine from anywhere in a program. It doesn't have to be called from a loop. I called it from this loop to demonstrate two things. A subroutine can execute many times, but is written into the program only once. A subroutine is a way to leave a loop temporarily and then return to the loop. You can also use a GOTO to leave the loop and another GOTO to jump back into the loop.

NESTED LOOPS

A *nested* loop is one loop running inside another loop. This routine is a loop. When you get it

running, we will nest another loop inside it. Notice that there is no line 40 in this program. Please enter

```
NEW
10 PRINT CHR$(147)
20 FOR I=1 TO 100
30 PRINT I
50 NEXT I
```

Run the program. The numbers are displayed too fast to read easily. In addition, the display scrolls. There are two ways to improve readability. One is to slow down program execution so the numbers don't move so fast on the screen. The other is to display the data one screen load at a time. That method will be demonstrated later in this chapter.

To slow it down, put a delay loop at line 40. The program then looks like this:

```
10 PRINT CHR$(147)
20 FOR I=1 TO 100
30 PRINT I
40 FOR DL=1 TO 200: NEXT DL
50 NEXT I
```

When this program runs, lines 10, 20 and 30 will execute. At line 40, a delay loop is encountered. It is a complete loop, all on one line. The program waits while the loop at line 40 runs to completion. The amount of delay is determined by how high the delay loop counts. The name of the delay-loop counter, DL, is shown in the NEXT statement for that loop.

After the delay loop has run to completion, line 50 executes. The name of the loop counter is shown on that line. It is clear which NEXT statement works with which FOR statement. Line 50 is the NEXT statement for the I loop.

That sequence of events continues until the I loop has operated the specified number of times. When it has operated 100 times, the program ends. On each pass through the I loop, there is a pause while the DL loop runs to completion.

The I loop operates 100 times. The DL loop also operates 100 times. Each time the DL loop runs, it counts to 200.

Run the program. As you can see, the delay loop slows it down. List it again.

The top of the I loop is line 30 and the bottom of the I loop is line 50. The entire DL loop is on line 40. Therefore, the DL loop is nested inside the I loop. The DL loop is the *inner* loop. The I loop is the *outer* loop.

Nested loops are like the mileage counter in the speedometer of an automobile. The mileage counter is called an *odometer*. Starting at the right, each dial on the odometer must make a full revolution before the dial on its left can advance to the next number on that dial. Those are mechanical nested loops. The dial on the right is the innermost loop.

A common mistake is to get the NEXT statements out of order in programs using nested loops. To show the nested-loop structure more clearly, I will rewrite this program to put the delay loop on two lines. Don't enter this version.

```
10 PRINT CHR$(147)
20 FOR I=1 TO 100
30 PRINT I
40 FOR DL=1 TO 200
45 NEXT DL
50 NEXT I
```

Loops

Considering only the FOR and NEXT statements, this is the structure of that program:

```
FOR I
FOR DL
NEXT DL
NEXT I
```

In nested loops, the NEXT statements are always in reverse order, compared to the FOR statements. That causes the inner loop to run to completion each time, before the outer loop can cycle again. Suppose there are three nested loops with A, B and C as the counters. The structure is:

```
FOR A
FOR B
FOR C
NEXT C
NEXT B
NEXT A
```

In this arrangement, the A loop waits for the B loop to run to its full count before the A loop can complete one operation. The B loop waits for the C loop to run to its full count before the B loop can complete one operation. The C loop doesn't wait for anything. It runs to completion each time the program reaches it.

Single loops are powerful. Nested loops are even more powerful. With Commodore BASIC, you can nest as many as 10 loops.

I used a delay loop to illustrate nested loops because that is very easy to understand. To give you a better idea of how they are used, here is a plan for a more complicated program. It isn't the program, it's just a plan. When you can plan a nested-loop operation, writing it is easy.

Suppose you have five mailing lists and you want to find the name J. J. Jones, no matter which list it is on. The mailing lists are for five fraternal organizations: Elks, Lions, Mice, Yaks and Zebras. Elks are on list number 1, Zebras are on list number 5. This is the plan. Don't enter it.

```
10 FOR LIST=1 to 5
20 GET SELECTED LIST
30 FOR ITEM=1 TO END OF LIST
40 IF ITEM="J. J. JONES" THEN GOTO 100
50 NEXT ITEM
60 NEXT LIST
100 DISPLAY DATA FROM LIST (such as Jones' address)
```

Line 10 sets up a loop to get five lists, one at a time. The first list to be searched is list number 1. Line 20 finds the selected list on a disk or cassette and brings it into memory so it can be searched.

Then line 30 sets up a loop to examine all names on the list selected by line 10. Line 40 tests each item on the list to see if it is J. J. Jones. If so, the program displays data about Jones and ends by jumping out of the loop to line 100. If not, the program continues.

Line 50 jumps back to line 40 to examine the next item on the list. If all items on list 1 have been examined without finding Jones, the program proceeds from line 50 to line 60 instead of looping back.

Line 60 gets the next list by incrementing the counter for that loop and jumping back to line 20. That will get list 2. The inner loop then searches the second list. This continues until Jones is found or all five lists have been searched.

SCROLL CONTROL

You should still have the program in memory that displays the numbers 1 to 100 and scrolls

This is the program:

```
10 PRINT CHR$(147)
20 FOR I=1 TO 100
30 PRINT I
40 FOR DL=1 TO 200: NEXT DL
50 NEXT I
```

Run it to be sure you remember what it does. We put a delay loop at line 40. This demonstrated a way to slow down a scroll and also demonstrated a simple nested loop.

Instead of allowing a display to scroll, it is often better to break up a long list into smaller segments that will fit on the screen without scrolling. Displaying 20 items at a time usually works well. Each display is called a *screen load*. Each screen load is like a page in a book. To look over the entire list of items, examine the screen loads in sequence. Normally, lists of items are displayed by a loop. To get segments of 20 items at a time, monitor the loop counter and stop the loop each time it displays 20 items. Then start it again and display the next 20 items. That requires stopping the loop when the count is 20, 40, 60 and so forth. To do that, you need a way to test the counter to see if its value is a multiple of 20.

Test Method—The test is to see if the value of the counter divides evenly by 20. Dividing evenly means that there is no remainder in the result. Because the computer does decimal arithmetic, the remainder will be a decimal fraction. In the number 1.77, the .77 is a decimal fraction. Enter

```
PRINT 21/20
PRINT 40/20
```

Only multiples of 20 divide evenly by 20. A mathematical procedure is called an *algorithm*. Here is an algorithm to test a number to see if it is a multiple of 20. Please enter each of these steps.

```
COUNT=99
```

COUNT represents the value of the loop counter. $N = \text{COUNT}/20$

That step divides the count by 20. The result, N, will be tested to see if it has a decimal fraction. Obviously, it does. $N\% = N$

The type-designation symbol % causes a variable name to accept only whole numbers. The decimal portion is discarded, if there is a decimal portion. When $N\%$ is set equal to N, $N\%$ becomes the whole-number portion of N.

The program wants to know if N has a decimal fraction. If not, the count was evenly divisible by 20. To find out, $N\%$ is compared to N. IF $N - N\% = 0$ THEN . . .

If the difference between N and $N\%$ is zero, they have the same value. If they are the same value, then N does not have a decimal fraction because $N\%$ *cannot* have a decimal fraction.

Using the Test—We will use the program in memory to demonstrate scroll control. List it. When you are displaying a screen load at a time, you should print the screen loads as quickly as possible, without unnecessary delays. That means you don't want the existing line 40 in the program. Enter

```
35 I%=I/20
40 IF I/20-I%=0 THEN STOP
```

List it and check your typing. Line 35 divides the loop counter, I, by 20 and assigns that value to the variable name I%. I% will accept only the whole-number part.

Line 40 applies the test. It divides I by 20 and then subtracts I%. If the result is zero, I is evenly divisible by 20. That means the program has printed 20 items on the screen, so the program is stopped by a STOP statement. The items printed are just values of the loop counter. In a real-world program, they would be items from a list of some kind.

Run it. It displays 20 numbers and stops. Start the program again by entering CONT. It displays the next 20 numbers. Continue until all 100 numbers have been displayed.

Stopping and restarting a program using STOP and CONT is crude and not good programming. I used that method because I have not yet shown you how to stop and start programs more elegantly.

PRINTING A LIST

Here is more information about the PRINT statement. You have used it so far to print strings, such as PRINT "HELLO", and to print numerics, such as PRINT X.

You have also used a semicolon at the end of a PRINT statement to suppress a carriage return. Suppressing the carriage return causes whatever is printed next to appear on the same screen line, rather than the next line. Enter

```
PRINT "A"  
PRINT "B"  
PRINT "A": PRINT "B"
```

The last item is just two statements on the same line. It does the same thing as the preceding two PRINT statements. Enter

```
PRINT "A";: PRINT "B"
```

That is two statements on the same line but the first statement uses a semicolon to suppress the carriage return. Both A and B print on the same screen line. There is no space between them. Enter

```
PRINT "A ";: PRINT "B"
```

That prints a space between A and B. The space is included in the quotation marks with the A. It is part of that string. Instead of doing it that way, you could put a space ahead of B, inside those quotation marks. Enter

```
PRINT "A" "B"
```

That statement prints a list of items. The list of items to be printed is called a *print list*. No semicolon is needed between the items. When you print a list, they follow each other on the screen automatically. Notice that the space between the two sets of quotation marks is ignored by the computer. Enter

```
PRINT "A" " B"
```

That puts a space between them. It is part of the string expression that includes B. Enter

```
PRINT "A" " " "B"
```

That also puts a space between A and B. The space is itself enclosed in quotation marks. It is a literal string that's nothing but a space. That list has three items to be printed. Each is a string enclosed in quotation marks. Enter

```
PRINT "A" 7
```

Print lists can include both strings and numerics. Enter

```
PRINT "A" 7 8
```

The computer reads 7 8 as the number 78. To separate the 7 and the 8, you must provide the separation. Enter

```
PRINT "A" 7 " " 8
```

That prints three spaces between the numbers. Numbers print with a leading and trailing space. You are seeing the trailing space behind the 7, one space that was printed as a string in quotation marks, and the leading space for the 8.

TABBING OVER

The display screen has four *print zones* that you can use to display lists in vertical columns. To *tab* means to move the cursor to the next column. A comma in a PRINT statement causes an automatic tab to the next print zone on the screen. Enter

```
PRINT 6,7,8,9
```

If you don't like that spacing, or need more columns, you can set up your own tabs, using TAB statements. Enter

```
PRINT TAB(5) "A"
```

The A is printed in the sixth space from the left. TAB(5) means to skip over 5 spaces and print on the next space. Enter

```
PRINT TAB(5) "A" TAB(10) "B"
```

Each TAB statement in a print list counts from the left margin of the screen and ignores any other TAB statements in that program line. The screen has 25 horizontal rows and 40 vertical columns. The first column at the left is column 0. TAB(5) prints in column 5. TAB(10) prints in column 10, if counting begins at column 0.

NESTED-LOOP DEMONSTRATION

This is a nested-loop program that demonstrates nested-loop programs and TAB statements.

It will be very easy to enter if you duplicate lines by overtyping line numbers. For example, to make line 40, duplicate line 20. Then list the program and change I to J in line 40. To make line 50, start by duplicating line 30. Then change it so it is correct for line 50.

```
NEW
10 PRINT CHR$(147)
20 FOR I=1 TO 2
30 PRINT TAB(3)"PASS "I" THROUGH I LOOP"
40 FOR J=1 TO 2
50 PRINT TAB(6)"PASS "J" THROUGH J LOOP"
60 FOR K=1 TO 2
70 PRINT TAB(9)"PASS "K" THROUGH K LOOP"
80 NEXT K
90 NEXT J
100 NEXT I
```

Before you run it, let's discuss a couple of things. Lines 30, 50 and 70 print lists of items. Each list has four items: a TAB statement, a string, a numeric variable and another string. The numeric variables are the loop counters for the I, J and K loops.

The TAB statements are used to indent whatever is printed by each of the three loops. The indent is different for each loop. On the screen, this produces a graphic indication of the loops in operation. Run it.

REVIEW

Loops are probably the most important program routines. You will use them often and in a great variety of ways. This chapter shows you how they are set up and how they work. As you review this chapter, set up and run some loops for practice. In the accompanying table is a list of BASIC words used in this chapter. Please look them up in Appendix B of this book.

BASIC WORDS USED IN THIS CHAPTER			
FOR	GOSUB	GOTO	NEW
NEXT	PRINT	RUN	STEP
TAB			

5

Entering And Editing Program Lines

You already know a lot about entering and editing program lines because you have been doing it. This chapter is a complete discussion of entering and editing lines. Some of it will be a review.

This information applies both to program lines and statements that are typed and entered in the immediate mode. The word *keystroke* is used in a general way to mean anything that you type. Some keystrokes produce visible characters on the screen, some don't.

ENTERING A LINE

When you type something, the printable keystrokes are displayed on the screen so you can see what was typed. The letter A is a printable keystroke. Other keystrokes, such as SHIFT and cursor keys, may affect what is on the screen, but they don't print any additional characters on the screen.

When you have typed something, pressing the RETURN key enters what you typed into the computer for processing. If you are writing a program, what's actually entered is a program line, which is a group of up to 80 keystrokes. A program line can occupy one or two screen rows because screen rows hold only 40 characters.

To enter a program line, you must be sure the cursor is in that program line on the screen when you press RETURN. If a program line occupies only one row on the screen, the cursor can be anywhere on that row. If a program line occupies two rows, the cursor can be anywhere on either row.

When you type and enter a line, the computer will receive for processing exactly the same line that you see on the screen. Because you understand BASIC, you can predict what the computer will do when it executes that line.

IMMEDIATE MODE

When you turn on the computer, it's in the immediate mode. Whatever you type and enter is received by the BASIC interpreter and executed immediately. If it isn't a valid statement, an error message results.

The immediate mode has other names including *direct mode*, *command mode* and *calculator mode*. Commodore uses immediate mode and direct mode.

PROGRAM MODE

In this mode, you can write or run BASIC programs. To write a program line, begin with a line number. This automatically puts the computer in the program mode. When you press RETURN to enter a program line, the computer automatically resumes operating in the immediate mode.

A RUN statement puts the computer in the program mode and causes it to execute whatever BASIC program is in memory. When a program is interrupted, or ends, the computer returns to the immediate mode.

KEYBOARD MODES

The keyboard is always set to one of two modes: upper-case-and-graphics or upper-and-lower-case. One way to switch modes is to press Commodore-SHIFT.

Switching between these two modes changes the way the computer and display respond to a keystroke. If the keyboard is set for the upper-case-and-graphics mode, pressing key A produces an upper-case A on the screen.

Switching to the upper-and-lower-case mode and then pressing key A displays a lower-case a on the screen. It also changes every character on the screen to the character set used in the upper-and-lower-case mode, even if the characters were originally typed in the upper-case-and-graphics mode.

The screen can display only one character set at a time.

PRINTING STRINGS

In a PRINT statement, characters enclosed in quotation marks are displayed exactly as typed. One or more characters enclosed in quotation marks comprise a literal string. The characters that can be displayed are letters, numbers, punctuation marks, spaces and symbols.

Strings represented by a string variable name are also displayed by a PRINT statement. The statement PRINT NAME\$ will display the string represented by NAME\$.

PRINTING NUMERICS

In a PRINT statement, numeric expressions are not enclosed in quotation marks. The computer will evaluate the expression and print the result.

Numeric expressions that are represented by a numeric variable name are also displayed by a PRINT statement. The statement PRINT NUM will display the value represented by NUM.

PRINTING A LIST OF ITEMS

A single PRINT statement can be used to print a list of items. The items can be strings and numerics, intermingled as desired. Each item on the print list is displayed immediately after the preceding item, with no spaces between them. The items may themselves print spaces, as part of the items. Enter

```
PRINT "A " "B C" "D" "123 45"
```

PRINTING SPACES

Spaces enclosed in quotation marks are printed as spaces on the screen. Spaces not in quotation marks are ignored. Enter

```
PRINT "ABC"
```

```
PRINT "A B C"
```

```
PRINT "A" "B" "C"
```

Positive numbers have a blank space in front of the number. Negative numbers use that space for a minus sign. Both positive and negative numbers have a space at the end of the number. Enter

```
PRINT 67
```

You can see the leading space.

```
PRINT "A" 67 "B"
```

You can see the leading and trailing spaces.

```
PRINT "A" -67 "B"
```

That produces an error message. The computer thinks `-67` is an incomplete numeric expression to be evaluated. Try this

```
PRINT "A" 0-67 "B"
```

That works. You can see that the minus sign occupies the space in front of the number.

Sometimes, printing a string variable and a number or numeric variable causes a problem. Enter

```
A=7
PRINT A
PRINT A 8
B$="B"
PRINT B$
PRINT B$ A
PRINT A B$
```

The computer interprets `A8` and `AB$` as variable names that have not been given values. If you encounter a problem in printing lists of mixed variables, you can always do it with more than one `PRINT` statement. Enter

```
PRINT A; :PRINT B$
```

THE CURSOR

The cursor always designates the location where the next character will be displayed. After printing a character, the cursor automatically moves to the next following space on the screen and waits for the next character to be printed.

When a program is running, the cursor is normally not visible. But it works the same way whether you can see it or not. When creating screen displays, it will be helpful to track the location of the cursor mentally.

TAB()

This BASIC word moves the cursor to the right. The statement `PRINT TAB(n)` moves the cursor `n` columns, counting from the left border of the screen.

As you know, the screen has 40 columns, numbered from 0 to 39. `TAB(10)` causes the next character to print in column 10 on the screen. Enter

```
PRINT "A" TAB(10) "B"
```

Move the cursor along the characters on the screen and count spaces. The letter `A` is printed in column 0. `B` is printed in column 10.

Move the cursor down to a clear area on the screen and fully to the left. Enter

```
PRINT TAB(40) "C"
```

That skips an entire row on the screen and prints `C` at the first column of the next row. Try printing `TAB(80)`.

`TAB` statements are used to display data in columns on the screen. When the cursor is moved by a `TAB` statement, it does not erase any characters that are in its path.

SPC()

This is a BASIC word that moves the cursor to the right in a way similar to a `TAB` statement—with one important difference. `SPC(n)` moves the cursor `n` columns to the right of *wherever* the cursor was just before the `SPC()` statement was executed. Enter

```
PRINT "A" TAB(10) "B" SPC(4) "C"
```

The letter A is in column 0. B is in column 10. After B is printed, the cursor is in column 11. SPC(4) moves the cursor 4 times—to columns 12, 13, 14 and 15. It stops at column 15. The location of the cursor always establishes where the next character is printed. C is in column 15.

Another property of SPC() is that it will suppress a carriage return on the screen if it ends a PRINT statement. After it executes, it has the same effect as ending a PRINT statement with a semicolon. Enter

```
PRINT "A": PRINT "B"  
PRINT "A" SPC(5): PRINT "B"
```

USING COMMAS TO PRODUCE AUTOMATIC TABS

Commas in a PRINT statement cause automatic tabs to specified columns on the screen. For this purpose, the screen is organized into four *print zones*. The zones begin at columns 0, 10, 20 and 30. Each is 10 columns wide.

A comma in a PRINT statement or list of items to be printed causes an automatic tab to the beginning of the following print zone. If you are printing a table of data on the screen, and four print zones are enough, then using commas to produce automatic tabs is very convenient. Enter

```
PRINT "A","B","C","D"  
PRINT "A", , , "D"  
PRINT "A","B","C","D","E"
```

After you have used the four print zones, another comma tabs to the first print zone in the following row.

An automatic tab produced by a comma depends on the location of the cursor when the comma takes effect, not the location of the last printed character. If the cursor is at column 9, a comma will move it to column 10. If it is at column 10, a comma will move it to column 20. Enter

```
PRINT TAB(10)"A","B"
```

After printing A at column 10, the cursor is at column 11. The comma moves it to column 20. Enter

```
PRINT TAB(9)"A","B"
```

After printing A at column 9, the cursor is at column 10. The comma moves it to column 20. Enter

```
PRINT TAB(8)"A","B"
```

After printing A at column 8, the cursor is at column 9. The comma moves it to column 10. All of that was probably obvious. Enter

```
PRINT TAB(8)1,2
```

Numbers automatically print with a following space. After printing 1 at column 8, a space is automatically printed in column 9 and the cursor ends up at column 10. The comma moves it to column 20.

GRAPHICS SYMBOLS

Most of this book uses printed type to show program lines. In this chapter, many of the lines use graphics symbols. They appear on the display screen and also in printouts of programs. So you can see *exactly* what they look like, program lines using graphics symbols are reproduced from a printout made on the Commodore printer.

GRAPHICS SYMBOLS AND GRAPHICS CODES

When there is only one way to do something, the choice is easy. Do it that way. The Commodore 64 has two ways to do several kinds of things. One of them is displaying graphics symbols using a PRINT statement.

In the immediate mode, pressing SHIFT or the Commodore key with a letter key produces one of the graphics symbols shown on key fronts.

In a PRINT statement, those same symbols may be typed inside quotation marks to be printed. When the line executes, the symbol in the PRINT statement appears on the screen. Type the following characters without pressing RETURN:

```
PRINT"
```

To complete that statement, press SHIFT-Q. Then type quotation marks to end the statement. The result is:

```
PRINT"␣"
```

Pressing SHIFT-Q put a circle inside the quotation marks. Press RETURN. The same symbol is displayed. That can be done in the immediate mode or in a program line.

The other way is to use a number code to represent the circle, instead of the symbol. The number code for that character is 209. Enter

```
PRINT CHR$(209)
```

The same graphics character is displayed. Using number codes is discussed in the next two chapters. I want you to know about them now, so you will know that there is a set of graphics code numbers that can be used interchangeably with the graphics symbols in PRINT statements. I prefer to use the number codes, rather than the graphics symbols. I suggest that you read this chapter and the next two chapters before deciding which method you prefer.

Because the symbol is only one character, you can put about 70 graphics symbols in a single program line. If you use CHR\$(209), you can put only 7 or 8 of them in a program line. However, if you need a lot of graphics symbols, the best way to do it is with a loop. That way, you enter the symbol only once, whether it is a graphics symbol or the equivalent number code.

Using graphic symbols or number codes makes no difference in the amount of space required to store the program in memory. Both are stored the same way.

Graphics symbols are easier to use when writing a program because you don't have to look up anything. The symbols are shown on key fronts. If you see a graphics symbol on the screen, or in a printout of a program on paper, it is sometimes difficult to distinguish one from another similar symbol. Code numbers are unambiguous.

You can use one or the other, or both. You can intermix them in the same program line.

This chapter discusses the graphics symbols. Number codes are discussed in the next two chapters.

CONTROL SYMBOLS AND CONTROL CODES

Control relates to keystrokes that control something such as cursor location or screen colors. There are also two ways to write statements that control actions of the computer. You can use control symbols or control codes.

As you know, pressing SHIFT-CLR HOME clears the screen and homes the cursor. Enter this statement using SHIFT-CLR HOME to make the reversed-heart graphics character.

```
PRINT"␣"
```

That statement clears the screen. If you want to actually display a reversed heart, these keystrokes will do that: Use CTRL-RVS ON to set reversed characters. Then press SHIFT-S. If you put that symbol in a PRINT statement, it will display a reversed heart.

What the computer recognizes is the keystrokes made, not the resulting display.

Here is the other way to clear the screen and home the cursor.

```
PRINT CHR$(147)
```

You have been using that method in demonstration programs.

In my opinion, using code numbers for control functions is less ambiguous than using symbols. I won't force that opinion on you. Control symbols are discussed in this chapter. Control codes are discussed in the next two chapters. You can use them interchangeably, even on the same program line.

PRINTING GRAPHICS SYMBOLS

Graphics symbols may be typed inside quotation marks in a PRINT statement. The graphics symbol will be displayed. In the upper-case-and-graphics keyboard mode, a diamond symbol is produced by pressing SHIFT-Z. Enter and run

```
NEW
10 PRINT "◆"
```

Typing a series of graphics symbols causes that number of symbols to be displayed in sequence. Enter and run

```
10 PRINT "◆◆◆◆◆◆◆◆◆◆"
```

Graphics symbols can be intermixed with other characters from the keyboard in any combination. Using SHIFT-S to produce the graphics symbol, enter and run

```
NEW
10 PRINT "I ◆ BANANAS"
```

LOOKING AHEAD

The procedures in the rest of this chapter may seem a little complicated. I suggest that you read through it without trying to get every detail firmly in your mind on the first pass.

Enter the demonstrations. They are short and give you the general ideas quickly and in the simplest way.

When you have finished the chapter, you will have the overall ideas and understand the purposes of these procedures. Then, the details will seem less complicated and easier to understand. At the end of the chapter, I will suggest that you go through this material again, more slowly, and practice each procedure until you are mentally comfortable using it.

THE QUOTE MODE

When you type an opening quotation mark, meaning the first of a pair, the computer and display respond differently to some keystrokes. This is called the *quote mode* because it happens only when you are typing after an opening quotation mark. Its purpose is to allow you to control the screen display by including symbols inside quotation marks.

Table 5-1 on page 62 shows things that you can do in the quote mode by typing symbols inside quotation marks. It shows which keys to press and the resulting symbols. Please enter the following lines to demonstrate using the quote mode. They will have the same result in the immediate mode or as program lines.

The first line in each of the following demonstrations tells you what to type. Type what follows the word (Type). Letters and symbols marked on the keys are shown as marked on the keys. That is, if you see A, press A.

Other keystrokes are shown inside bracket symbols. For example, [SHIFT-CLR HOME] means hold down the SHIFT key while pressing the CLR HOME key. Spaces are used to make the typing instructions easier to read. The lines will work the same with or without the spaces.

The second line shows what will appear on the screen when you have followed the typing instructions. After you have typed the line, press RETURN to see what it does when executed.

```
(Type) PRINT "[SHIFT-CLR HOME]"
PRINT "␣"
```

You have used that before. It clears the screen.

```
(Type) PRINT "A [CRSR DOWN] [CRSR DOWN] B"
PRINT "A␣␣B"
```

**TABLE 5-1
CONTROL SYMBOLS IN THE QUOTE MODE**

To Do This	Use These Keystrokes	Symbol
HOME THE CURSOR	CLR-HOME	"H"
MOVE CURSOR UP	SHIFT-CRSR UP	"J"
MOVE CURSOR DOWN	CRSR DOWN	"Q"
MOVE CURSOR LEFT	SHIFT-CRSR LEFT	"I"
MOVE CURSOR RIGHT	CRSR RIGHT	"P"
SET BLACK	CTRL-1	"B"
SET WHITE	CTRL-2	"W"
SET RED	CTRL-3	"R"
SET CYAN	CTRL-4	"C"
SET PURPLE	CTRL-5	"P"
SET GREEN	CTRL-6	"G"
SET BLUE	CTRL-7	"B"
SET YELLOW	CTRL-8	"Y"
SET ORANGE	COMMODORE-1	"O"
SET BROWN	COMMODORE-2	"B"
SET LIGHT RED	COMMODORE-3	"R"
SET GRAY 1	COMMODORE-4	"G"
SET GRAY 2	COMMODORE-5	"G"
SET LIGHT GREEN	COMMODORE-6	"G"
SET LIGHT BLUE	COMMODORE-7	"B"
SET GRAY 3	COMMODORE-8	"G"
REVERSE ON	CTRL-9	"R"
REVERSE OFF	CTRL-0	"O"
CLEAR THE SCREEN	SHIFT-CLR HOME	"C"

The two Q symbols represent the cursor-down keystroke. The letter A is printed, then the cursor is moved down two rows, then the letter B is printed.

```
(Type) PRINT "A [CTRL-2] B [Commodore-7] C"
PRINT "AQBCC"
```

That changes character colors. It prints A in the normal light blue. Then it changes character color to white and prints B. Then it changes character color back to light blue and prints C.

```
(Type) PRINT "A [CTRL-9] B"
PRINT "ARB"
```

That causes a reversed character. It prints A. Then it enables the RVS ON function and prints B.

The RVS ON setting lasts only to the end of the line, or until an RVS OFF setting is made if that happens before the end of the line. Type something to verify that. To make a second line with reversed characters, you must set RVS ON again at the beginning of the second line. Enter

```
(Type) PRINT "A [CTRL-2] B"
PRINT "ARB"
```

That prints A in the normal light-blue character color. Then it changes character color to white and prints B.

Character colors remain set until changed to another color or restored to normal by pressing RUN STOP-RESTORE. Type something to verify that the character color remains set to white. Then restore normal character color.

```
(Type) PRINT "A [SHIFT-INST DEL] [SHIFT-INST DEL] B"
PRINT "AIBB"
```

SHIFT-INST DEL is the insert keystroke. It produces a symbol in the quote mode, but the symbol doesn't do anything when the string is printed.

```
(Type) PRINT "ABCDE [INST DEL] [INST DEL] F"
PRINT "ABCF"
```

INST DEL is the delete keystroke. When you enter a line, it deletes preceding characters. It's demonstrated in this chapter.

CONTROL SYMBOLS

In the preceding demonstrations, you used special symbols produced by cursor keys and other keys. When enclosed in quotation marks, these symbols control cursor position, character color and a few other things. The symbols themselves don't print—they control something.

In this book, these special symbols are called *control symbols*.

CONTROLLING HOW NUMERICS PRINT

You have been using control symbols inside quotation marks to affect the display of characters inside the quotation marks.

Numerics cannot be inside quotation marks, but you can use the same methods to control the display of numerics. To use the control symbols that must be in quotation marks, put them in quotation marks. After the second quotation mark, type the numerics that you want to display. Enter this example

```
(Type) PRINT "[CTRL-2]" 2+2 "[Commodore-7]" 9/3
PRINT " " 2+2 " " 9/3
```

That line selects white for the character color, then evaluates the numeric expression 2+2 and displays the answer in white. Then it selects the normal light-blue character color, divides 9 by 3 and prints the answer in light blue.

COMBINING GRAPHICS SYMBOLS, CHARACTERS AND CONTROL SYMBOLS

The benefit of the quote mode is that you can combine letters, numbers, punctuation, graphics symbols and control symbols. This provides good control over what appears on the screen, where it appears and what it looks like.

In Chapter 1, you made a box by typing graphics symbols in the immediate mode. The following program lines will draw the top and one side of a box.

After reading the discussion of these program lines, finish the program to make a complete box. Then print your name inside the box. The main problem will be selecting graphics symbols that work with the ones I used. Line 20 has six cursor-down keystrokes and 10 cursor-right keystrokes. Line 30 uses Commodore-Y to make the graphics symbol. Line 40 uses Commodore-H. Enter these lines:

```
NEW
10 PRINT " ":REM CLR SCREEN
20 PRINT " " ;:REM START
30 FOR I=1 TO 10: PRINT " " ;:NEXT:REM TOP
40 FOR I=1 TO 6: PRINT TAB(20)" " :NEXT:REM RT SIDE
```

Line 10 clears the screen and then produces a carriage return. Line 20 prints two groups of control symbols to move the cursor to the starting position to display the box. It moves the cursor down six rows on the screen and then moves it 10 columns to the right. The semicolon holds the cursor at the following column on that row, waiting for the next thing to display.

Line 30 prints 10 Commodore-Y keystrokes to draw the top of the box. That can be done by printing 10 identical control symbols, similar to the method used in line 20. It can also be done by a loop operating 10 times. For this line, the method used doesn't matter much. Printing 10 graphics symbols would make a shorter program line. But, using a loop is a more general way to do it.

The semicolon in line 30 is executed 10 times. Each graphics symbol follows the preceding

symbol on the same row. When the last symbol is printed, the cursor is at the following column on that row.

Line 40 starts with the cursor at that location and makes the right side of the box. It uses a loop to print six graphics symbols, produced by Commodore-H. The TAB statement causes each symbol to appear in column 20. There is no semicolon on that line, so each PRINT statement moves the cursor down one row on the screen after it executes.

Tabbing over and moving the cursor down one row causes each of the vertical bars produced by Commodore-H to connect to the one above. You can't see where the cursor ends up after printing the sixth vertical bar because the computer returns to the immediate mode and displays the prompt. The prompt is always at the left side of the screen.

Using a loop at line 40 is better than repeatedly printing a graphics symbol to make the right border of the box. That's because each graphics symbol must be tabbed over. If a loop were not used, you would have to type TAB(20) six times to make the right border.

Run the program. Now, finish this program to draw the rest of the box and put your name in it. I suggest the following steps: Move the cursor back to the starting position—the top-left corner of the box. Homing the cursor and then printing repeated control symbols will do that. The line will be similar to line 20.






Then draw the left side of the box in a way similar to line 40. Both the TAB statement and the graphics symbol will be different. Look over the graphics symbols printed on the keys and pick one that should work. You may have to try more than one.

Then locate the cursor and draw the bottom of the box. When that is done, use a string of control symbols to move the cursor inside the box. Then print your name.

A program that does those things is at the end of this chapter. Do your program first and then look to see how I did it. Your way may be better.

CONTROL SYMBOLS THAT REQUIRE SPECIAL ENTRY

There are some control symbols that the computer will recognize and respond to if they appear inside quotation marks, but you cannot type them in the quote mode. These symbols are shown in Table 5-2.

TABLE 5-2 SPECIALLY ENTERED CONTROL SYMBOLS		
To Do This	Use These Keystrokes	Symbols
CARRIAGE RETURN	REVERSED SHIFT-M	
UPPER & LOWER CASE	REVERSED N	"  "
UPPER CASE & GRAPHICS	REVERSED SHIFT-N	"  "
DISABLE CASE-SWITCH KEYS	REVERSED H	"  "
ENABLE CASE-SWITCH KEYS	REVERSED I	"  "

To use these control symbols, leave a blank space where you want to use the symbol, type the ending quotation mark and finish the line. Enter it by pressing RETURN.

These special control symbols are typed as reversed characters. The reason you cannot type them in the quote mode is that setting reversed characters in the quote mode causes a reversed R to appear in the line and reverses following characters. These symbols are *entered* as reversed characters, not ordinary characters following a reversed R symbol.

To do that, move the cursor to a blank row on the screen and press CTRL-9 to set the keyboard for reversed characters. Then move the cursor back into the line, place it at the blank space and type the desired symbol. Then press RETURN to enter that line again with the added symbol.

One of the control symbols that you can enter by this method is a carriage return. If you place a carriage-return symbol in a string, the computer will stop printing the string at the carriage return, move down to the beginning of the next row and resume printing the string. To demonstrate that, enter

```
NEW
10 PRINT "NOW IS THE TIME FOR ALL GOOD MEN TO COME"
```

That will wrap around and use part of the next screen row. When you pressed RETURN to enter that line, the cursor moved automatically to a clear row on the screen. Press CTRL-9 to set the keyboard for reversed characters.

Put a carriage return between TIME and FOR. Put it in the space that is already there. Move the cursor to the space between TIME and FOR. Press SHIFT-M. A reversed diagonal symbol is displayed in that space. That is the control symbol for a carriage return. Press RETURN to enter the line. Pressing RETURN ends the reversed-character setting automatically.

Run the program. It breaks the first row at TIME and resumes with FOR on the following row. Of course, you can do the same thing with two PRINT statements.

Commodore literature refers to the carriage return produced by this method as SHIFT-RETURN. Pressing SHIFT-RETURN on the keyboard has the same effect as pressing RETURN and does not produce the result just demonstrated.

As you can see in Table 5-2, there are two control symbols that allow you to select the character set from a program line. This does the same thing as pressing Commodore-SHIFT.

Pressing Commodore-SHIFT works in the immediate mode or the program mode. Another person, when running your program, could switch character sets and cause the display to change.

To prevent that, there is a control symbol to disable the Commodore-SHIFT keystroke. There is another control symbol to turn it on again.

EDITING LINES

You can edit any line on the screen by moving the cursor into that line in the immediate mode. This capability is referred to as a *screen editor*. You edit one line at a time, but it can be anywhere on the screen.

If a program line that you want to edit is not on the screen, list that part of the program and then edit the line.

CHANGING ONE OR MORE CHARACTERS

If one or more characters in a line is incorrect, but the total number of characters is correct, you can fix the errors just by typing correct characters on top of the incorrect characters. When you have done that, press RETURN while the cursor is still in that program line.

ADDING CHARACTERS TO THE END OF A LINE

To add characters to the end of an existing line, move the cursor to the end of the line. Type whatever you want to add and press RETURN.

DELETING AN ENTIRE LINE

Type the line number with nothing behind it. Press RETURN.

DELETING PART OF A LINE

Place the cursor to the right of the last character to be deleted. Press unshifted INST DEL as many times as necessary. Then press RETURN.

The delete keystroke works as just described in all modes except the insert mode, which is discussed later in this chapter.

PUTTING A NEW LINE IN A PROGRAM

Enter the new line using a line number that has not already been used in the program. The new line will be automatically placed in numerical order in the program.

REPLACING A PROGRAM LINE

Enter the replacement line using the same line number as the line to be replaced. The new line will take the place of the old line with the same number.

INSERT MODE

This mode is primarily intended for editing but it can also be used to add control symbols to lines in a way similar to the quote mode.

Editing in the Insert Mode—To insert characters in a line, move the cursor to the first character to the right of the insert location and press SHIFT-INST DEL, which is the *insert keystroke*. That puts the computer in the insert mode.

Each insert keystroke opens up one space in the line by moving characters to the right. Then, fill the open spaces with characters. Typing the same number of keystrokes as were opened up by insert keystrokes ends the insert mode. This leaves the cursor in the line so other editing changes can be made, if desired. Enter

```
NEW
10 PRINT "ABCDE"
```

Move the cursor over the D and open up three spaces. Fill them with XXX, then press RETURN. List the line and then run it.

If you open up more spaces than you need, you can just press RETURN, which will enter the edited line and end the insert mode. However, the spaces will remain in the line. If they are part of a PRINT statement, they will print as spaces. Enter

```
10 PRINT "ABCDE"
```

Move the cursor over the D and open up three spaces. Fill them with XX and a blank space, then press RETURN. List the line and then run it.

If too many spaces were opened up, they can be removed by the delete keystroke. Until all spaces that were opened by insert keystrokes have been filled, the delete keystroke does not work normally. It displays a reversed T—the control symbol for delete. When all spaces have been filled, the insert mode is ended and the delete keystroke resumes normal operation. Making additional delete keystrokes will delete the reversed T symbols placed in the line. Enter

```
10 PRINT "ABCDE"
```

Move the cursor over the D and open up three spaces. Fill them with XX and a blank space. Make two delete keystrokes by pressing unshifted INST DEL twice, then press RETURN. List the line and then run it.

What happens if reversed T symbols are left in the line is discussed in the following section.

Using the Insert Mode to Add Control Symbols to a Line—In the insert mode, the keys that move the cursor, set character color, set reverse on and off, and clear the screen do not operate normally. Instead, they cause control symbols to appear in the line. With one exception, these symbols and their effects are the same as in the quote mode. They are listed in Tables 5-1 and 5-2.

If you move the cursor back into a line between quotation marks and make the insert keystroke, that does not return to the quote mode. The computer is in the insert mode. With one exception, the same control symbols can be produced in the insert mode as in the quote mode. When the line is executed, those symbols do the same thing no matter how they were placed in the line.

The exception is the INST DEL key. In the quote mode, insert keystrokes display a symbol, but the symbol doesn't do anything. The symbols remain in the program, but have no effect. In the quote mode, delete keystrokes work normally—they delete characters to the left.

In the insert mode, insert keystrokes open up spaces. Delete keystrokes produce a reversed T

control symbol. If the control symbols are left in the line, they operate in the same way as delete keystrokes when the line is executed.

If you put three delete control symbols in a line, they will delete three characters to the left when the line executes. The deletion also occurs when you list the program. Enter

```
10 PRINT "ABCDE"
```

Then place the cursor over the D and make two insert keystrokes. Then make two delete keystrokes and press RETURN. The result should show two reversed T control symbols between the C and the D. When they execute, they will delete the B and the C. List it and run it.

SPACES IN LINES

Some versions of BASIC require spaces to separate BASIC words from other characters in the line. Commodore BASIC does not require spaces. Omitting unnecessary spaces makes program lines shorter, so they require less memory, but also makes them more difficult to read and understand. Enter

```
10FORI=1TO10:PRINTI:NEXT
```

Run it. The BASIC interpreter recognizes the words FOR, TO, PRINT and NEXT, even though there are no spaces in the line.

That's the reason variable names cannot contain BASIC words. If you use AUTOMOBILE as a variable name, the computer recognizes TO as a BASIC word between AU and MOBILE. It doesn't find a valid BASIC statement using TO, so it declares a syntax error.

If your program will fit in the available memory, with enough memory left over for data needed to run the program, then there is no compelling reason to omit spaces in program lines. The test is whether or not the program runs. If you get an OUT OF MEMORY error message, the program may be too long.

Making a program shorter by removing spaces and remarks will cause it to run a little faster. In my opinion, it is usually not worth the time and effort. Try it and form your own opinion.

If you store programs on tape in a cassette recorder, make the program as compact as possible. There can be a worthwhile reduction in the amount of time needed to put the program on tape and load it back again into the computer.

ABBREVIATIONS FOR BASIC WORDS

You can type many of the BASIC words with fewer keystrokes by using standard abbreviations. The abbreviation is the first one or two characters of the word, followed by the next character typed with a SHIFT key held down.

For example, NEXT can be entered as N SHIFT-E. An abbreviation that doesn't follow that rule is the question-mark symbol. Enter

```
NEW  
10 ? "ABCDE"
```

List it and run it. That abbreviation enters PRINT in the program line. Abbreviations don't save space in a program listing because the complete BASIC word is used in the listing.

Using abbreviations does not reduce the amount of space that a program occupies in memory because BASIC words are compressed when read into memory anyway. They are represented by numbers, called *tokens*. There is a token for each BASIC word and symbol.

There are two reasons to use abbreviations. If you are entering a long program line, and it won't fit on two screen rows using BASIC words, you may be able to get it on two rows by using abbreviations.

When listed on the screen, it will expand to more than two rows because complete BASIC words appear in the listing. That is OK if you just want to read the line. To edit the line, you must type it over again using abbreviations, so it fits on two rows. Otherwise, the line will not be accepted when you press RETURN to enter it.

The other reason to use abbreviations is if you can type program lines faster that way. This is

Entering And Editing Program Lines

another example of having two ways to do something. I find it easier to type the entire BASIC word than to try to remember the abbreviation and press SHIFT while entering the last character. You may prefer to use the abbreviations.

A list of Commodore BASIC abbreviations is in Appendix C.

REVIEW

Please look over this chapter again to review the information. The most important part is the editing methods. They may seem complicated, but after you have used them a while, they become easy. Type some program lines, run them, change them and run them again. Do that until you can change lines easily and correctly.

Practice using control symbols in both the quote mode and the insert mode. Do that until you are sure you understand how they work, but don't spend a lot of time on it. Don't try to memorize the control symbols and their effects. The things that can be done with control symbols can also be done with code numbers. This is discussed in the next two chapters.

When I have discussed number codes in the next two chapters, I will suggest that you use them instead of control symbols. If I make that suggestion after you have memorized the control symbols, you may decide to continue using control symbols. Wait until you see how to use number codes before deciding which to memorize and use.

Whether you use symbols or number codes in program lines, you will occasionally have a problem making changes to lines that have been entered. The problem is that you will get into the quote mode or insert mode without intending to. Control symbols will appear in the program line. Press RETURN. Then move the cursor back into the line and delete the undesired symbols.

There is only one new BASIC word in this chapter. It is SPC(. Look it up in Appendix B.

A PROGRAM TO DRAW A BOX AND PRINT A NAME IN IT

```
10 PRINT"┌":REM CLR SCREEN
15 REM 6 CRSR DOWN & 10 CRSR RIGHT
20 PRINT"XXXXXXXXXXXXXXXXX":REM START
25 REM 10 COMMODORE-Y SYMBOLS
30 FOR I=1 TO 10:PRINT"┐":NEXT:REM TOP
35 REM 6 COMMODORE-H FOR RT SIDE
40 FOR I=1 TO 6:PRINT TAB(20)"|":NEXT
45 REM HOME, 7 CRSR DN, 9 CRSR RT
50 PRINT"XXXXXXXXXXXXXXXXX";
55 REM 6 COMMODORE-N TO MAKE LEFT SIDE
60 FOR I=1 TO 6:PRINT TAB(10)"|":NEXT
65 REM CRSR UP ONE ROW, ONE SPACE TO RT
70 PRINT TAB(11)"└";
75 REM PRINT 9 COMMODORE-P FOR BOTTOM
80 FOR I=1 TO 9:PRINT"_":NEXT
85 REM 3 UP AND 7 LEFT TO PRINT NAME
90 PRINT"ITXXXXXXXXNAME"
```

6 ASCII Codes

In preceding chapters I demonstrated using the keyboard to display graphics symbols, set character color, home the cursor, and similar things. The keystrokes resulted in graphics symbols or control symbols.

All of the things that can be done using symbols can also be done using number codes. There are two kinds of number codes. One is *ASCII codes*. The other is *screen-display codes*.

This chapter discusses ASCII codes. The next chapter discusses screen-display codes.

ASCII CODES

ASCII, pronounced *askey*, is an abbreviation of *American Standard Code for Information Interchange*. It is a standard method of communication over telephone lines between such things as teletype machines and computers.

There are 128 ASCII codes, using code numbers 0 to 127. Among the 128 ASCII code numbers, codes 0 to 31 are used for control codes. Codes 32 to 127 are used for characters.

Character codes represent visible characters: letters, numbers and punctuation marks.

Control codes are intended to control the mechanical or electrical operation of the distant machine. These are operations such as moving the cursor or advancing the paper.

ASCII codes are built into most computers. Your computer can send ASCII codes to communicate with a distant computer or receive ASCII codes and respond to or interpret each code number.

When your computer receives ASCII codes, it doesn't matter how far away the sender of these codes is. The sender can be you, at the keyboard of your computer. Because your computer is built to understand ASCII codes, you can "speak" to it in ASCII. Most computers don't often communicate with distant computers, so the most common use of ASCII codes is to communicate with a "local" computer—meaning the one you are using.

Computer manufacturers use some of the code numbers for a different purpose than standard ASCII. The resulting code is not standard ASCII, but it is usually called ASCII anyway.

The Commodore 64, and most other computers, can use 256 code numbers—from 0 to 255. Because standard ASCII uses only the first 128 numbers, this provides an additional 128 code numbers—128 to 255. Computer makers use the additional 128 codes for whatever they wish, such as graphics characters.

Commodore ASCII is similar to but not the same as standard ASCII. Commodore ASCII is shown in Table 6-1. Codes 0-31 are used for control codes. Most are not standard ASCII. Codes 32-90 are used for upper-case letters, the numbers 0-9 and punctuation marks. In this range of code numbers, standard ASCII and Commodore ASCII are the same.

In standard ASCII, codes 91-127 are used for lower-case letters and some additional symbols. Commodore ASCII uses these codes for other things. There are no lower-case letters in Commodore ASCII. The computer can display lower-case letters by a method discussed later.

TABLE 6-1
EFFECT OF ASCII CODES IN UPPER-CASE AND GRAPHICS MODE

Code	Display	Effect If Not Displayed	Code	Display	Effect If Not Displayed
0		NO EFFECT	67	C	
1		NO EFFECT	68	D	
2		NO EFFECT	69	E	
3		NO EFFECT	70	F	
4		NO EFFECT	71	G	
5		WHITE CHARACTERS	72	H	
6		NO EFFECT	73	I	
7		NO EFFECT	74	J	
8		DISABLE COMMODORE-SHIFT	75	K	
9		ENABLE COMMODORE-SHIFT	76	L	
10		NO EFFECT	77	M	
11		NO EFFECT	78	N	
12		NO EFFECT	79	O	
13		CARRIAGE RETURN	80	P	
14		SET UPPER & LOWER CASE	81	Q	
15		NO EFFECT	82	R	
16		NO EFFECT	83	S	
17		CURSOR DOWN	84	T	
18		REVERSE ON	85	U	
19		HOME CURSOR	86	V	
20		BACKSPACE & DELETE	87	W	
21		NO EFFECT	88	X	
22		NO EFFECT	89	Y	
23		NO EFFECT	90	Z	
24		NO EFFECT	91	[
25		NO EFFECT	92]	
26		NO EFFECT	93	^	
27		NO EFFECT	94	_	
28		RED CHARACTERS	95	~	
29		CURSOR RIGHT	96		
30		GREEN CHARACTERS	97		
31		BLUE CHARACTERS	98		
32		SPACE	99		
33	!		100		
34	"		101		
35	#		102		
36	\$		103		
37	%		104		
38	&		105		
39	'		106		
40	(107		
41)		108		
42	*		109		
43	+		110		
44	,		111		
45	-		112		
46	.		113		
47	/		114		
48	0		115		
49	1		116		
50	2		117		
51	3		118		
52	4		119		
53	5		120		
54	6		121		
55	7		122		
56	8		123		
57	9		124		
58	:		125		
59	;		126		
60	<		127		
61	=		128		NO EFFECT
62	>		129		ORANGE CHARACTERS
63	?		130		NO EFFECT
64	@		131		NO EFFECT
65	A		132		NO EFFECT
66	B		133		CODE FOR FUNCTION KEY 1

TABLE 6-1 (continued)

EFFECT OF ASCII CODES IN UPPER-CASE AND GRAPHICS MODE

Code	Display	Effect If Not Displayed	Code	Display	Effect If Not Displayed
134		CODE FOR FUNCTION KEY 3	165		
135		CODE FOR FUNCTION KEY 5	166	#	
136		CODE FOR FUNCTION KEY 7	167		
137		CODE FOR FUNCTION KEY 2	168	*	
138		CODE FOR FUNCTION KEY 4	169	▼	
139		CODE FOR FUNCTION KEY 6	170		
140		CODE FOR FUNCTION KEY 8	171		
141		CARRIAGE RETURN ON SCREEN	172		
142		SET UPPER CASE & GRAPHICS	173		
143		NO EFFECT	174		
144		BLACK CHARACTERS	175		
145		CURSOR UP	176		
146		REVERSE OFF	177		
147		CLEAR SCREEN, HOME CURSOR	178		
148		INSERT	179		
149		BROWN CHARACTERS	180		
150		LIGHT RED CHARACTERS	181		
151		GRAY 1 CHARACTERS	182		
152		GRAY 2 CHARACTERS	183		
153		LIGHT GREEN CHARACTERS	184		
154		LIGHT BLUE CHARACTERS	185		
155		GRAY 3 CHARACTERS	186		
156		PURPLE CHARACTERS	187		
157		CURSOR LEFT	188		
158		YELLOW CHARACTERS	189		
159		CYAN CHARACTERS	190		
160		SPACE	191		
161					
162			192 TO 223		SAME AS 96 TO 127
163			224 TO 254		SAME AS 160 TO 190
164			255		SAME AS 126

TABLE 6-2

EFFECT OF ASCII CODES IN UPPER-AND-LOWER-CASE MODE

Code	Display	Effect If Not Displayed	Code	Display	Effect If Not Displayed
0		no effect	32		space
1		no effect	33		
2		no effect	34	"	
3		no effect	35	#	
4		no effect	36	\$	
5		white characters	37	%	
6		no effect	38	&	
7		no effect	39	'	
8		disable commodore-shift	40	(
9		enable commodore-shift	41)	
10		no effect	42	*	
11		no effect	43	+	
12		no effect	44	,	
13		carriage return	45	-	
14		set upper & lower case	46	.	
15		no effect	47	/	
16		no effect	48	0	
17		cursor down	49	1	
18		reverse on	50	2	
19		home cursor	51	3	
20		backspace & delete	52	4	
21		no effect	53	5	
22		no effect	54	6	
23		no effect	55	7	
24		no effect	56	8	
25		no effect	57	9	
26		no effect	58	:	
27		no effect	59	;	
28		red characters	60	<	
29		cursor right	61	=	
30		green characters	62	>	
31		blue characters	63	?	

TABLE 6-2 (continued)
EFFECT OF ASCII CODES IN UPPER-AND-LOWER-CASE MODE

Code	Display	Effect If Not Displayed	Code	Display	Effect If Not Displayed
64	@		131		no effect
65	a		132		no effect
66	b		133		code for function key 1
67	c		134		code for function key 3
68	d		135		code for function key 5
69	e		136		code for function key 7
70	f		137		code for function key 2
71	g		138		code for function key 4
72	h		139		code for function key 6
73	i		140		code for function key 8
74	j		141		carriage return on screen
75	k		142		set upper case & graphics
76	l		143		no effect
77	m		144		black characters
78	n		145		cursor up
79	o		146		reverse off
80	p		147		clear screen, home cursor
81	q		148		insert
82	r		149		brown characters
83	s		150		light red characters
84	t		151		gray 1 characters
85	u		152		gray 2 characters
86	v		153		light green characters
87	w		154		light blue characters
88	x		155		gray 3 characters
89	y		156		purple characters
90	z		157		cursor left
91	[158		yellow characters
92	\		159		cyan characters
93]		160		space
94	^		161		
95	_		162		
96	`		163		
97	a		164		
98	b		165		
99	c		166		
100	d		167		
101	e		168		
102	f		169		
103	g		170		
104	h		171		
105	i		172		
106	j		173		
107	k		174		
108	l		175		
109	m		176		
110	n		177		
111	o		178		
112	p		179		
113	q		180		
114	r		181		
115	s		182		
116	t		183		
117	u		184		
118	v		185		
119	w		186		
120	x		187		
121	y		188		
122	z		189		
123	[190		
124	\		191		
125]				
126	^		192 to 223		same as 96 to 127
127	_		224 to 254		same as 160 to 190
128		no effect	255		same as 126
129		orange characters			
130		no effect			

HOW CHARACTERS ARE DISPLAYED

When you type characters at the keyboard, an ASCII code number is produced inside the computer for each character you type. This code number is sent to the display to cause a character to appear on the screen. Each code number is used to select one character from a set of available characters.

Even though there is only one set of ASCII code numbers, there are two character sets that can be displayed. One is called the *upper-case-and-graphics set*. The other is the *upper-and-lower-case set*. As you know, you can switch back and forth between these two sets of characters by pressing Commodore-SHIFT.

Because the two character sets are different, the same ASCII code number may produce two different characters on the screen—depending on which character set is in use. For example, in the upper-case-and-graphics mode, ASCII code 65 produces a capital letter A. In the upper-and-lower-case mode, code 65 produces a lower-case letter a. Code 97 produces an upper-case A.

Commodore ASCII uses the second group of 128 codes for several purposes. Some are additional control codes. Some are additional graphics symbols. Some are duplicates of other code numbers.

COMMODORE ASCII CODES

For the rest of this book, ASCII will mean Commodore ASCII—the codes that are actually used by your computer.

Table 6-1 shows the effect of ASCII codes 0-255 when the upper-case-and-graphics mode has been selected. Table 6-2 shows the effect of codes 0-255 when the upper-and-lower-case mode has been selected.

By comparing these tables, you will see that more graphics characters are available in the upper-case-and-graphics mode because lower-case letters are not available in that mode. The upper-and-lower-case mode has fewer graphics characters because both upper- and lower-case letters are available.

In this book, and in most programming, the upper-case-and-graphics mode is used unless the program requires upper- and lower-case letters to be displayed on the screen.

CHR\$()

Table 6-1 shows that the ASCII code for letter A is 65. To send that code number from the keyboard to the computer, you must indicate that it is an ASCII code, not the number 65. That is done by the BASIC word CHR\$(). CHR\$ is pronounced *character string*.

The ASCII code number is placed inside the parentheses. For example, CHR\$(65) tells the computer that it has received ASCII code number 65. It doesn't tell the computer what to do with that code.

The BASIC word PRINT is used with ASCII codes to tell the computer to do whatever the code represents. If the code represents a printable character, it will be displayed on the screen. If the code represents some other action, such as a carriage return, that will be done. Earlier, I said that the word PRINT really means *do it*. This is another example of that meaning.

With the computer in the upper-case and graphics mode, enter

```
PRINT CHR$(65)
```

The computer interprets that statement to mean *print the character represented by ASCII code 65*. In the upper-case-and-graphics mode, that displays a capital letter A on the screen.

Press Commodore-SHIFT to change character sets. The letter A changes to a lower-case a. All characters on the screen change to the upper-and-lower-case mode.

Switch back to upper-case-and-graphics mode. Enter

```
PRINT CHR$(65)CHR$(65)
```

It prints a list of two items. Both are the letter A. You are displaying the letter without actually pressing its key. Enter

```
PRINT CHR$(65)CHR$(13)CHR$(65)
```

The computer printed an A, performed a carriage return, and then printed another A. Table 6-1 shows that CHR\$(13) is a carriage return. ASCII code 13 is one of the control codes.

WHY ARE THERE SO MANY WAYS TO DISPLAY CHARACTERS?

You can display a character by pressing that key on the keyboard, in the immediate mode. That is done so you can see what you typed and correct it if necessary before entering it.

You can display one or more characters by a PRINT statement that includes the characters to be displayed, such as PRINT "A" or PRINT 123. That's an essential part of programming.

If a distant computer tells your computer to PRINT CHR\$(65), it will display an A. If you tell your computer to do that, it will make the same response.

You can display graphics symbols, such as a circle, by a PRINT statement that prints that graphics symbol or a PRINT statement that prints the ASCII code number for that symbol, such as CHR\$(209).

You can produce a carriage return by pressing the RETURN key on the keyboard, by printing a control symbol or printing CHR\$(13).

You can display one or more characters by assigning the characters to a variable name and then printing the variable name. For example: X=7: PRINT X. This method is also an essential part of programming.

All of these methods exist for logical reasons. You can write better and more powerful programs by knowing and using these methods. Use the best method for whatever you want the program to do. If it is not clear that one is best, use the method you prefer.

WHAT ASCII CODES DO

Because ASCII codes are executed by a PRINT statement, you can see what they do by displaying them on the screen.

Most of the things that can be done with control symbols and graphics symbols can also be done with ASCII codes. Those that can't be done with ASCII codes can be done with the other type of number codes discussed in the next chapter.

It is interesting to see what happens when each ASCII code is printed, and it helps you learn to use them. This program will get you started. Enter

```
NEW
5 REM DISPLAY ASCII CODES
10 PRINT CHR$(147)
20 FOR C=0 TO 255
30 PRINT C TAB(8) CHR$(C) TAB(18) "DONE"
40 FOR DL=1 TO 400:NEXT
50 PRINT CHR$(154) CHR$(142):REM RESTORE
60 NEXT
```

Line 20 sets up a loop to run from 0 to 255. The counter is C. The value of the counter will be used to print codes 0 to 255.

Line 30 begins by printing the value of C, so you can see which ASCII code number is being displayed. Then it tabs over to column 8 and prints CHR\$(C). That will cause the computer to do whatever CHR\$(C) represents, for each value of C. Then, line 30 tabs over to column 18 and prints the word DONE.

Printing DONE has two purposes. It shows you that whatever that code number does has happened. That is useful when a code number doesn't do anything. If the code does something, it may affect how the word is displayed. If so, you can see the effect of that code number.

Some of the code numbers change character color. Code 14 selects upper and lower case. Those actions will affect how the word **DONE** appears on the screen. Line 50 then resets normal character color by printing `CHR$(154)`. Look at Table 6-1 to verify that. Then it selects upper-case and graphics characters, to cancel the effect of code 14. It quickly cancels the lower-case characters produced by code 14. You have to watch carefully to see them at all.

When you run this program, you will see every control action that can be produced by ASCII codes. You will see every character that can be displayed on the screen using the upper-case-and-graphics set of characters.

Everything that will be demonstrated in this chapter also applies to the equivalent control and graphics symbols discussed in Chapter 5, when they are used in `PRINT` statements. An ASCII code and its equivalent symbol do not produce two different actions by the computer. They are just two different ways of producing the same action.

This program uses a loop to print the ASCII code numbers in an orderly way. There is no way to use a loop to display the effects of the control *symbols* and graphics *symbols* in a `PRINT` statement, except by displaying the equivalent ASCII code numbers.

Run the program once or twice, for a first look at the codes and the results of printing them. You may wish to run it again while reading the following discussion of control codes.

THINGS YOU SHOULD KNOW ABOUT CONTROL CODES

There is nothing complicated about the character codes. Print one and you get the character that it represents. The control codes are also easy to use if you know what they do.

The following paragraphs discuss codes that produce interesting or unusual results. Observing these effects now will be helpful later, when you write a program and unexpected things happen on the screen. The discussion also applies to the equivalent control symbols.

Some ASCII character-code numbers produce different characters, depending on the character set in use. The control-code numbers produce the same result with either character set. There is only one set of ASCII code numbers. The display shows two different sets of characters simply by responding differently to the same set of ASCII codes.

`CHR$(8)` disables the Commodore-SHIFT keystroke. Enter

`PRINT CHR$(8)`

Then press Commodore-SHIFT. Nothing happens. The Commodore-SHIFT keystroke has been disabled.

`CHR$(9)` enables the Commodore-SHIFT keystroke. Enter

`PRINT CHR$(9)`

Try Commodore-SHIFT again. It has been enabled. You can use these statements in a program to control whether or not the *user* of the program can change character sets.

`CHR$(13)` produces a carriage return. It caused the word **DONE** to be moved down one row when you ran the program.

`CHR$(14)` switches to upper-and-lower-case set. You saw that briefly because the program switched back to upper case immediately. Enter

`PRINT CHR$(14)`

Then type something. To restore upper case only, enter

`PRINT CHR$(142)`

You can use those statements in a program to control screen displays.

`CHR$(17)` moves the cursor down one row on the screen. Enter

`PRINT "A" CHR$(17)CHR$(17)CHR$(17)"B"`

After the letter **A** was displayed, the cursor was in column 2 on the screen. Three cursor-down actions moved it down three rows. It remained in column 2 and printed the letter **B**. Cursor-down just moves the cursor down.

ASCII code numbers used as `CHR$()` in a `PRINT` statement are not enclosed in quotation marks. The `$` symbol identifies `CHR$()` as a string. If `CHR$(65)` is placed inside quotation marks, the letters `CHR$(65)` will be displayed instead of the character represented by ASCII code 65.

`CHR$(18)` sets reversed characters. Reversed characters remain in effect only to the end of a program line or until a `RETURN` is executed. Enter

```
PRINT CHR$(18) "ABCD"  
PRINT "ABCD"
```

Reversed characters are very useful in creating interesting and effective screen displays. They're also easy to do.

`CHR$(19)` homes the cursor. In a program, it does the same thing as pressing the `CLR HOME` key in the immediate mode.

`CHR$(20)` backspaces and deletes. It does the same thing as pressing the `INST DEL` key in the immediate mode.

Enter

```
PRINT "ABCD"CHR$(20)CHR$(20)"EF"
```

Deleting in a program line is sometimes useful. For example, if you ask the computer user to enter a data item and he enters an invalid reply, you can delete it on the screen and ask him to do it again.

`CHR$(29)` moves the cursor to the right. All four of the cursor movements can be programmed. This is useful in positioning the cursor where you want it when creating a screen display. You can display a standard form to be filled in by the user. As each item is entered, move the cursor to the location where the next item should be entered.

`CHR$(34)` prints a quotation mark. The demonstration program was a little misleading because it showed graphics symbols between the quotation mark that was displayed by `CHR$(34)` and the word `DONE`. List the program to see why that happened.

Line 30 prints a list of items. The quotation mark was printed in column 10 because of the preceding `TAB` statement. The quotation mark put the computer in the quote mode. Then, another `TAB` statement moved the cursor to column 18. That required seven cursor-right movements. Because the computer was in the quote mode, cursor-right movements were indicated by a graphics symbol. In a program, you wouldn't do it that way, unless you intended to display cursor-right symbols.

The quote mode is ended by printing a second quotation mark or by a `RETURN` keystroke at the end of a program line.

`CHR$(34)` is used in program lines to display a quotation mark when nothing else will do. Enter

```
PRINT CHR$(34)"HELLO"CHR$(34)
```

Both of those `CHR$(34)`s put the computer into the quote mode, but each quote mode is ended immediately. `CHR$(34)` is the only way you can put quotation marks on the screen by a `PRINT` statement. Try it this way:

```
PRINT " "HELLO" "
```

The computer prints what it finds inside the first pair of quotation marks, which is nothing at all. Then it evaluates `HELLO` as a numeric expression and prints that value, which is zero. Then it prints what it finds inside the second pair of quotation marks.

Codes 133 to 140 are associated with the function keys. This is discussed later.

There are more control codes in Table 6-1. I think their uses will be obvious. You may want to experiment with some of them to be sure you know how to use them.

WHEN TO USE ASCII CODES

There is usually no reason to use ASCII codes to represent letters and punctuation symbols. It is easier to type the letters and punctuation directly from the keyboard, rather than type `CHR$(65)` for `A`, and so forth. But there are a few useful tricks, such as using `CHR$(34)` for quotation marks.

The choice between using ASCII control codes and control symbols was discussed in the preceding chapter. As a *method*, using number codes is more general than using symbols. I suggest that you use control codes. But, you can program equally well either way.

IMPORTANT CONTROL CODES

To create and manage a screen display, you must control two things: what prints and where it prints. Sometimes, controlling where it prints is the hardest part.

The ASCII codes give you a full set of commands that move the cursor. They are used to put the cursor where you want a character to print. Here are the codes and what they do:

ASCII CODE WHAT IT DOES

CHR\$(147)	Clears screen and homes cursor.
CHR\$(19)	Homes cursor without clearing screen.
CHR\$(13)	Moves cursor down one row and fully left.
CHR\$(17)	Moves cursor down one row.
CHR\$(29)	Moves cursor right one column.
CHR\$(145)	Moves cursor up one row.
CHR\$(157)	Moves cursor left one column.

KEYBOARD BUFFER

A *buffer* is a space in memory that is used as a temporary parking place for data. There are several buffers in the computer.

Characters typed at the keyboard pass through a keyboard buffer on their way to the computer. The purpose of this buffer is to hold keystrokes until the computer is ready to receive them.

A fast typist can type short bursts faster than the computer can process them. If that happens, the keystrokes wait in the buffer until the computer can accept them. Keystrokes are not lost. When the typist pauses or slows down, the computer empties the buffer and catches up.

When the computer is busy doing something else, such as running a loop, it doesn't pay attention to the keyboard. Keystrokes are held in the keyboard buffer until the computer is ready to receive them. To demonstrate that, enter

```
NEW
```

```
10 FOR I=1 TO 200:PRINT I:NEXT
```

That loop just counts to 200 and displays the count. Run it. While it is running, type your first name. Your name waits in the keyboard buffer until the loop finishes running. Then the characters are displayed in the same order that they were typed. Characters come out of the keyboard buffer in the same order they were typed into it.

Run it again. While the loop is running, type

```
12345678901234567890
```

When the loop stops, only the first 10 keystrokes are displayed. That's because the keyboard buffer holds only 10 keystrokes. If you type more than that, additional keystrokes are lost.

GET

This is a BASIC word that accepts only a single keystroke. It is used to examine individual keystrokes, to see what they are so the program can decide what to do with them. It is also used to stop program execution until the operator presses a key.

A GET statement must provide a variable name for the keystroke it's waiting for. The form is GET *variable name*. The name can be a numeric or string variable name. I suggest using only string variables. It is easy to convert a string to a number, using VAL(), if that's what the program needs.

This discussion shows how to use GET with string variables. The statement GET A\$ tells the computer to get the next keystroke and give it the name A\$.

When that statement is executed, the computer will look in the keyboard buffer. If a keystroke is there, it accepts the keystroke. If not, it accepts what it found, which is nothing.

Because the variable name used in this GET statement is a string variable name, A\$, the computer treats whatever it gets as a string. A string with nothing in it is called a *null string*, or *null*.

As you know, strings are typed inside quotation marks. A null string is "". Nothing is between the quotation marks.

The statement GET A\$, will *always* get something from the keyboard buffer, even if it is a null.

To stop program execution requires two statements, which are usually on the same line. I will refer to the two statements, used together, as a GET routine. An example is:

10 GET A\$: IF A\$="" GOTO 10. The first statement gets whatever is in the buffer. The second statement says, *if you got nothing, go back to the beginning of this line and do it again*.

It will continue doing that until the GET statement finds something in the buffer that is not a null. The way to put something in the buffer is to press a key.

The GET routine is a loop on a single line. It will wait indefinitely until a key is pressed at the keyboard—if the keyboard buffer was empty when the statement first executed. If a keystroke is waiting in the buffer, the GET statement grabs it, and the program will not stop.

A GET statement does not display the keystroke that it gets from the keyboard buffer. It stores it in memory, using the variable name supplied by the GET statement. To display that character, a separate PRINT statement is required.

ASC()

ASC is a BASIC word used to find the ASCII code number for a keystroke. The keystroke must be inside the parentheses. Enter

```
PRINT ASC("A")
```

The computer tells you that 65 is the ASCII code number for A. ASC() works only with strings. Notice that the letter A in the preceding PRINT statement is enclosed in quotation marks to indicate that it is a string. Enter

```
V$="C":PRINT V$:PRINT ASC(V$)
```

Those statements assigned the variable name V\$ to the character C. It printed V\$ and then printed ASC(V\$). The general form of the statement is PRINT ASC(*string*), in which *string* means anything that the computer accepts as a string. It can be characters typed inside quotation marks or a string variable name.

If the string has more than one character, ASC() gets the ASCII code number for the first character in that string and ignores remaining characters. Enter

```
PRINT ASC("APPLE")
FRUIT$="APPLE"
PRINT ASC(FRUIT$)
```

GETTING ASCII CODE NUMBERS FROM THE COMPUTER

When you are programming, you will sometimes need to know the ASCII code for a keystroke. If an ASCII table, such as Table 6-1, isn't handy, you can just ask the computer. The following program does that. It also shows you some things about the keyboard that have not been demonstrated yet. Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT"MAKE A KEYSTROKE.":PRINT
30 GET K$:IF K$="" GOTO 30
40 PRINT"ASCII CODE IS " ASC(K$)
50 PRINT
60 GOTO 20
```

Line 10 clears the screen. Line 20 displays an instruction. The second statement on line 20 prints a blank row on the screen to improve the display.

Line 30 uses a GET routine to wait for a keystroke. The keyboard buffer will be empty when line 30 executes, even if it was full before you ran the program. Executing a RUN statement from the immediate mode clears everything out of memory except the program itself, and then runs the program.

Therefore, line 30 will get the *next* keystroke, not one that was stored in the keyboard buffer. A good way to think about line 30 is that it *receives* the next keystroke and puts it in memory with the name KS\$. This line uses an abbreviated form of IF-THEN-GOTO. Look up IF in appendix B.

Line 40 prints a list of two items. The first item is information. It tells the user what will be displayed. The second item is ASC(KS\$) — the ASCII code for KS\$.

Line 50 prints a blank row on the screen. Line 60 jumps back to line 20 to do it all over again. This program is an endless loop. To stop execution, press RUN STOP.

Run the program and press key A. Refer to Table 6-1 while pressing character keys on the keyboard. The ASCII code numbers displayed by the program should be the same as those in the table. Stop running the program when you want to. Keep the program in memory.

SELECTING CHARACTER COLOR BY ASCII CODES

Table 6-1 shows the ASCII codes that select character colors, but they are scattered all over the table. If you use those codes when programming, a separate table that shows only those code numbers will be more convenient. Table 6-3 is for that purpose, but it is incomplete. The center column must be filled in.

The program in memory will help you do that. Run it again and select black as the character color by pressing CTRL-1. The program tells you that the ASCII code number for that keystroke is 144. That number is already entered in Table 6-3.

TABLE 6-3 SELECTING CHARACTER COLORS		
Color	ASCII Code	Keystroke
Black	144	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
Gray 1		Commodore-4
Gray 2		Commodore-5
Light Green		Commodore-6
Light Blue		Commodore-7
Gray 3		Commodore-8

Why Aren't Characters Black?—An important and useful property of the GET routine that you are running is that it “captures” control codes without executing them.

Please break out of the program and list it. In a program, control codes and control symbols are executed by a PRINT statement. At line 30, the keystroke was made that selects black as the character color. But, nowhere in the program is that keystroke used in a PRINT statement.

In line 40, what is printed is ASC(KS\$), not KS\$. The result is that the program “knows” what keystroke was made but that keystroke has no effect.

Finish the Table—Run the program again. Make the remaining keystrokes that select character colors and write them in the center column of Table 6-3. Don't break out of the program when you have finished.

FUNCTION KEYS

At the right of the keyboard are four function keys. The program in memory should still be running. Press the key labeled *f1*. It produces ASCII code 133. When unshifted, the function keys produce the odd code numbers, 133, 135 and so forth. Hold down the SHIFT key to produce the even numbers.

Fill in Table 6-4 and then we will use the function keys in a program.

TABLE 6-4 ASCII CODES FOR FUNCTION KEYS	
Function Key	Code
f1	
f3	
f5	
f7	
f2	
f4	
f6	
f8	

USING FUNCTION KEYS

Function keys don't do anything except produce an ASCII code number. To use them in a program, define what they do and tell the user that by a display on the screen. Then, when the user presses one of the keys, detect the keystroke with a GET routine and perform the defined action. Here is an example:

Break out of the program you are running. List it. Endless loops are OK for demonstration routines, but not for practical programs that somebody else may run.

This version uses function keys to give the user a choice of making another keystroke or ending the program. Change the program starting at line 50.

```
10 PRINT CHR$(147)
20 PRINT "MAKE A KEYSTROKE.":PRINT
30 GET KS$:IF KS$="" GOTO 30
40 PRINT "ASCII CODE IS " ASC(KS$)
```

```
50 PRINT:PRINT"PRESS F1 TO CONTINUE"  
   CHR$(13) TAB(6) "OR F7 TO END"  
60 GET REPEAT$:IF REPEAT$="" GOTO 60  
70 IF REPEAT$=CHR$(13) GOTO 10  
80 IF REPEAT$=CHR$(136) THEN END  
90 GOTO 60:REM ERROR TRAP
```

Line 50 displays instructions. Line 60 gets a keystroke named REPEAT\$. Lines 70 and 80 test REPEAT\$. If it is CHR\$(133), key f 1 was pressed. The program jumps back to line 10. Line 10 clears the screen and then the program repeats.

If REPEAT\$ is CHR\$(136), key f 7 was pressed. Line 80 ends the program.

Line 90 is a trap for keyboard errors. If neither line 70 nor line 80 execute, the user did not press f 1 or f 7. Because the instructions allow only f 1 and f 7, any other key is not a valid response. If the program reaches line 90, it jumps back to line 60 to get another keystroke. In effect, it just ignores invalid keystrokes. A better way is to tell the user that there was a keyboard error and then jump back to get another keystroke.

Run it and press some character keys. It runs the same as before, except that you must press f 1 to loop back, and it clears the screen each time. As a programmer, you probably prefer the endless loop because it runs faster and you don't have to bother with keystrokes to loop back. If you are writing a program for someone else to use, don't let it get into an endless loop.

While the program is running, press f 1 as the requested keystroke. It displays the code for f 1 and then displays the instructions to repeat. Press f 1 again to loop back.

This time, press f 7 as the requested keystroke. It displays the code for f 7 and does not end the program. Press f 7 again. The program ends. List it.

This program displays the code for any keystroke first. At that point, the program doesn't attach any special significance to f 1 or f 7. After the keystroke code has been displayed and the instructions to repeat are displayed, then f 1 and f 7 mean repeat or end.

Run it again. For the requested keystroke, press CTRL. Nothing happens. There are some keys that a GET routine does not recognize.

Run the program and press keys to find all of the keystrokes that a GET routine does not respond to.

DEFINING NEW FUNCTIONS FOR ANY KEY

You can use this technique with any key. Suppose you write a program that does some calculations. The user enters data as requested by the program. Then, he sees instructions that say something like, TO CALCULATE TOTAL, PRESS T and some other choices.

Use a GET routine to detect which key was pressed. To detect the keystroke T, you can test either for CHR\$(84) or for "T" because both have the same meaning to the computer.

PLANNING A PROGRAM

The first step in writing a program is to make a plan. Think about what the program should do, and in what order. Then write down small steps, in English, that will accomplish that.

Suppose you want to write a simple game entitled *Guess The Secret Number*. You already know how to play it. Maybe this plan will work:

- 1) Clear the screen.
- 2) Display instructions for first player.
- 3) Get secret number from first player as NUM\$.
- 4) Display instructions for second player.
- 5) Get guess from second player as GES\$.
- 6) Test guess by comparing GES\$ to NUM\$.
- 7) If equal, go to step 9.
- 8) If not equal, go to step 5.
- 9) Ask if players want to play again.

10) If no, clear screen and end program.

11) If yes, loop back to step 1.

When you have made a plan, start writing the program lines by typing them at the keyboard. Test the program as you write each segment by running it and observing what it does. It may help to display the values of variables, such as NUM\$ and GES\$ while you are testing. You can usually do that in the immediate mode. If the program doesn't work correctly, fix it before going any further.

PRACTICE

For programming experience, write the Secret Number program just described. You can do it with the BASIC words and programming methods that have been discussed.

Stick with it until it works. The only way to learn to program is to program. The more difficulty you have, the more you learn when you conquer the difficulty.

My version of that program is at the end of this chapter. Don't look at it until you get your program working well.

REVIEW

Look over this chapter and repeat the demonstrations. The new BASIC words are ASC(), CHR\$() and GET. Look them up in Appendix B.

GUESS THE SECRET NUMBER

This program does a couple of things that I didn't ask you to do, but it follows the same basic plan.

NEW

```
5 REM GUESS THE SECRET NUMBER GAME
10 PRINT CHR$(147)
20 PRINT SPC(8)"GUESS THE SECRET NUMBER"
30 PRINT:PRINT
40 PRINT"PLAYER #1, TYPE A NUMBER FROM 0
   TO 9"
50 GET NUM$:IF NUM$="" GOTO 50
60 PRINT
70 IF ASC(NUM$)<48 OR ASC(NUM$)>57 THEN
PRINT "NO, NO. TYPE A NUMBER.":GOTO 50
80 PRINT:PRINT"THANK YOU!"
90 FOR DL=1 TO 600:NEXT
100 PRINT CHR$(147):PRINT:PRINT
110 TRY%=0: REM COUNTS GUESSES
120 PRINT"PLAYER #2, GUESS THE SECRET NU
MBER.":PRINT
130 PRINT"BY PRESSING A NUMBER KEY FROM
0 TO 9."
140 GET GES$:IF GES$="" GOTO 140
150 TRY%=TRY%+1:PRINT:PRINT
160 PRINT"THAT WAS GUESS NUMBER"TRY%
170 PRINT"AND IT WAS ";
180 FOR DL=1 TO 1000:NEXT
190 IF GES$=NUM$ GOTO 300
200 PRINT"- - - WRONG!"
210 PRINT"TRY AGAIN.":GOTO 140
```

(Program continued on next page.)

```
300 PRINT"- - - RIGHT!"
310 PRINT:PRINT:PRINT
320 PRINT"PLAY AGAIN?  YES/NO?"
330 PRINT"PLEASE PRESS Y OR N."
340 GET REPLY$:IF REPLY$=" " GOTO 340
350 IF REPLY$="N" THEN PRINT CHR$(147):
    END
360 GOTO 10
```

Most of this should be obvious. The spaces in line 20 center the title on the screen. PRINT statements that don't print anything are to produce blank rows on the screen to make the display look better. Line 50 gets the number as NUM\$.

Line 70 prevents the program from accepting anything except a number from 0 to 9 as NUM\$. It tests ASC(NUM\$). ASCII codes for numbers are 48 to 57. If the ASCII code for NUM\$ is less than 48 or greater than 57, what was typed by player #1 cannot be a number. The program displays a message and jumps back to line 50 to get a valid input.

When a number has been typed, line 80 thanks player number 1. Line 90 is a delay loop. Line 100 clears the screen so new instructions can be displayed.

This program counts the number of guesses, using TRY% as the counter. Line 100 initializes the counter by setting it to zero.

Line 140 gets a guess from player number 2. Line 150 adds 1 to TRY%, to count the guesses. Line 160 displays the number of guesses.

Line 170 begins a message that tells whether or not the guess was correct. Line 180 is a delay loop to build a little tension in the player's mind while he waits to find out if he guessed correctly or not.

Line 190 tests the guess. If it is correct, the program jumps ahead to line 300.

If the guess is not correct, line 200 executes. It says that the guess was wrong. Line 210 asks for another guess and jumps back to line 140 to get it.

If the guess was correct, the program jumps from line 190 to line 300. Line 300 says that the guess was correct. Line 320 asks if the players want to play again. Line 340 gets the reply. If the reply is negative, the program ends.

If the reply is affirmative, line 360 executes and jumps back to line 10 to do it again. Notice that the program forces the first player to enter a number from 0 to 9. It allows the second player to press any key.

Play *Guess The Secret Number* until the thrill is gone.

7

Screen-Display Codes

On the inside, computers use only numbers. ASCII codes are used to store printable characters in memory and also to provide some control functions such as carriage returns and cursor control. ASCII codes were discussed in the preceding chapter.

An ASCII character code used in a PRINT statement causes a character to appear on the screen. But, the ASCII code number is not printed directly. Instead, it is used to select a character from one of two character sets. The ASCII code number may produce two different characters, depending on which character set is in use.

The characters in the character sets are each represented by a number from 0 to 255. Because these codes represent the actual characters displayed, these numbers are called *screen-display codes*.

To display a character on the screen, an ASCII code number is produced by a keystroke or from a statement in a program line. Then, the ASCII code is translated into a screen-display code. The character represented by that screen-display code number is selected from one of two character sets and displayed.

In this chapter I will show you how to control screen background and border colors. You already know how to set character color. Then I'll discuss the set of code numbers actually used by the computer to control the display—the screen-display codes.

INPUT-OUTPUT DEVICES

This book uses the word *computer* in two ways. It is used in a general sense to mean all of the equipment that you have assembled together: the electronics inside the keyboard housing, the keyboard itself, the display and whatever else you may have connected, such as disk drives, a cassette recorder or a printer.

A more precise definition is that the computer is a group of electronic circuits, called *chips*, mounted inside the same housing as the keyboard. Devices such as the keyboard, display, printer, cassette recorder and disk drive are not part of the computer. They are *connected* to the computer. They are *input-output devices*.

Input means that data flows to the computer. *Data* is anything that conveys information—letters, numbers and symbols. The computer receives data from input devices, processes it as instructed, and sends it to output devices.

The keyboard is an input device. It sends keystrokes to the computer, but cannot receive data from the computer.

The display is an output device. The computer sends number codes, called *screen-display codes*, to the display. This causes characters and symbols to be displayed on the screen. It will help you understand the rest of this chapter if you consider the display as an output device that just displays whatever the computer tells it to.

A printer is an output device. It receives data from the computer and prints it on paper.

Some devices are both input and output. These include the cassette recorder and disk drives. They can receive data from the computer, store it, and play it back into the computer when requested.

Plug-in game and program cartridges are input devices. They input to the computer but do not receive data from the computer.

DISPLAY CONTROLLER

Some of the electronic chips inside the keyboard housing are used to operate the display. I will refer to this electronic system as the *display controller*. The computer tells the display controller what to display and the controller makes it happen.

WHAT THE DISPLAY DOES

In the program mode, the display does what the program “tells it to.”

In the immediate mode, the display shows what you typed. As each key is pressed, that character flows through the keyboard buffer into the computer. It goes to the display controller, which puts it on the screen. The screen editor allows you to change characters on the screen or add to whatever is there. Putting characters on the screen does not cause the computer to do anything.

When you have a statement typed and corrected, if necessary, you press the RETURN key. The characters in that program line on the screen are sent to the BASIC interpreter. If the line was preceded by a line number, it is placed in memory as part of a program. The computer then waits in the immediate mode for you to type something else.

If what was input from the keyboard does not begin with a line number, the BASIC interpreter executes it immediately. If it is not a valid statement, an error message results.

When you run a program in memory, each line in the program is delivered to the BASIC interpreter for execution, just as if the line had been typed from the keyboard—with one difference. The program line is not displayed on the screen. Only the *result* of program execution is displayed.

Another function of the display is to list the program in memory. The LIST command causes the computer to get program lines from memory, in numerical order, and display them on the screen. The program is not executed, it is just displayed. Then, you can edit or change any line on the screen.

COMPUTER MEMORY

Strictly speaking, memory is not part of the computer itself. It is another input-output device. As you know, there are two kinds of memory, random-access and read-only.

Random-access memory, RAM, is an input-output device. The computer can send data to memory or receive data from memory. Data in RAM can be changed by sending in different data to replace it. When this book refers to *memory*, it is referring to RAM.

Read-only memory, ROM, is an input device. Data is placed permanently in this type of memory when the memory unit is manufactured. Data can be sent from ROM into the computer, but not from the computer into ROM.

If you purchase a game or program cartridge, and plug it into the Cartridge Slot on the back of the Commodore 64 keyboard, you have added ROM memory to the computer. The game or program is stored in a ROM chip in the cartridge. The computer has more total memory when the cartridge is plugged in than it did before. When you use the computer to run the game or program in the cartridge, it runs a program stored in the cartridge.

MEMORY LOCATIONS

The computer operates by *reading* various memory locations to get instructions—such as a

BASIC program in RAM memory. While operating, it may store and retrieve data in memory. To do these things, it must know how to find locations in memory.

Each memory location has a number that serves as its address. Each location can store a single keystroke or character, represented by a number code.

The amount of memory is expressed in units called *K*. The symbol *K* represents the number 1024. If a computer has 64K of memory, the actual number of locations is 65,536. The beginning address in memory is location zero and the last address is 65535. Memory-location numbers are written without commas.

The Commodore 64 has 64K of random-access memory, plus additional read-only memory—such as the ROM unit that holds the BASIC interpreter. BASIC can work with only 64K of memory at one time. That's because the highest address number that BASIC can use is 65535.

When you turn on the Commodore 64, it automatically selects part of available RAM and parts of ROM to arrange a total of 64K of memory to be used by BASIC. Part of the 64K is RAM, to store a BASIC program and data. The other part is ROM needed to run a BASIC program—the BASIC interpreter in ROM, and other things. The result is that everything needed to write or run BASIC programs is available in a range of *addresses* beginning with zero and ending at 65535.

USES FOR COMMAS

Commas have different meanings, depending on how they are used.

Commas in Strings—If a comma is included in a string—inside quotation marks—it will be displayed as part of the string. If that string is given a string variable name, the comma is still part of the string. Enter

```
PRINT "A,B,C"
XY$="A,B,C":PRINT XY$
PRINT "THERE ARE 12,345 CARS"
```

In the last statement, the characters 12,345 are treated as part of a string, not as a number.

Automatic Tabs—If a comma is included in a print statement, *but is not part of a string*, it causes automatic tabs. Enter

```
PRINT "A","B","C"
PRINT 1,2,3,4
PRINT "A","B,C"
```

Commas as Data Separators—When commas are not part of a string, and not included in a PRINT statement to produce automatic tabs, they act as data separators. Data separators are usually called *delimiters*. They set *de limit* for *de data*.

When the computer sees a comma in data, it stops reading and assumes that it has a complete data item. If there is more data behind the comma, and the program is not written to read more data, an error message results. Enter

```
X=12 , 345
```

The computer doesn't know what to do with the numbers following the comma, so it displays an error message.

If the statement is written so that the computer resumes accepting data after a comma, what it reads is taken as the *next* data item.

POKE

This chapter demonstrates several things that you can do by putting numbers into specified locations in memory. This is called *poking* numbers into memory.

POKE is a BASIC word that tells the computer to put a number into a specified memory location. The form is

POKE *location, number*

In a POKE statement, a comma separates the memory location from the number you are placing there. For example, POKE 2050,65 puts the number 65 into memory location 2050.

POKE 2050,65 is a complete statement that can be executed.

I remember that a POKE statement must have two numbers by thinking that you must specify the memory location and then what to put there—using a comma as a data separator.

PEEK()

The BASIC word PEEK is the opposite of POKE. It gets the number stored at a memory location so you can see what it is. The form is

PEEK(*n*)

in which *n* is an address in memory. PEEK(2050) is an example.

PEEK(2050) is not a statement. It is handled as a numeric variable name. Its value is the number at the specified location in memory. You can PRINT PEEK(2050), or do arithmetic with it. You can assign the value of PEEK(2050) to another variable name. Enter

```
POKE 2050,65
PRINT PEEK(2050)
PRINT CHR$(PEEK(2050))
V=PEEK(2050)
PRINT V
PRINT CHR$(V)
X=2*PEEK(2050)
PRINT X
```

Numbers stored in a single memory location are in the range of 0 to 255. One location can store one character. Characters that can be displayed, such as A, are stored as their ASCII code numbers. I remember that a PEEK() statement requires a number in () by thinking of () as a memory location. I imagine that I am *peeking* into a ().

INPUT

The following demonstration uses INPUT statements to receive data from the keyboard. As you know, GET receives only a single keystroke and does not require pressing RETURN to enter the keystroke. GET does not display the character typed.

INPUT receives a sequence of keystrokes, ending with a RETURN keystroke. It displays the keystrokes typed. An INPUT statement must supply a variable name for the data that it receives from the keyboard. The format is INPUT *variable name*.

When that statement is executed, the program stops and waits for a keyboard input. It displays a question mark as a prompt to the user. It also displays the cursor. As a keyboard input is typed, the cursor moves on the screen to show where the next character will be displayed.

INPUT will accept keystrokes until the RETURN key is pressed. That signals the computer to accept the sequence of keystrokes that were typed and place it in memory using the variable name supplied by the INPUT statement.

The variable name supplied in the INPUT statement determines the type of data that can be accepted. If a numeric variable is supplied, only numerics can be input. If a string variable name is supplied, strings may be input.

There is more information about INPUT statements in the following chapter.

HOW TO CONTROL BORDER AND BACKGROUND COLORS

Here is one way to use PEEK and POKE statements. There are memory locations that control border and background colors on the display. Location 53280 controls the border and location 53281 controls the background.

You will gradually learn and remember several important memory locations in your computer, just as you know your phone number and street address. A good way to remember something is to

Screen-Display Codes

write it down in a notebook or on a piece of paper that you keep handy. Write this down:

BORDER COLOR LOCATION 53280

BACKGROUND LOCATION 53281

To select border and background colors, you must poke number codes into those two locations. The codes are shown in Table 7-1. There are 16 possible colors, numbered from 0 to 15.

Let's try a few. Enter

POKE 53280, 0

POKE 53281, 0

POKE 53280, 4

Restore normal screen colors by pressing RUN STOP-RESTORE.

TABLE 7-1 SCREEN-COLOR CODES	
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow
8	Orange
9	Brown
10	Light Red
11	Gray 1
12	Gray 2
13	Light Green
14	Light Blue
15	Gray 3

DISPLAYING SCREEN COLORS

The screen can show three areas of color: border, background and character color. Some combinations of character and background colors are easier to read than others. The overall appearance and visual effect of a display change when you change colors.

Here is a program to display all possible color combinations. When you run it, make notes of combinations you like. Then you can use them again later. The plan is to set character color from the keyboard at the beginning of the program. Then the program asks you to enter a number to set border color.

With character and border color set, the program uses a loop to display all 16 background colors. Then it invites you to set another character color and do it again. Enter.

NEW

5 REM DISPLAY COLOR COMBINATIONS

10 PRINT CHR\$(147):PRINT:PRINT

(Program continued on next page.)

```
20 PRINT " ENTER CHAR COLOR NAME";:
  INPUT CC$
30 PRINT:PRINT " ENTER ASCII CODE";:
  PRINT " FOR CHAR COLOR ";
40 INPUT CC:PRINT:PRINT
50 PRINT " CHOOSE BORDER COLOR 0 TO 15"
60 PRINT:PRINT " ENTER BORDER COLOR ";:
  INPUT BR:PRINT CHR$(147):REM CLEAR
```

This segment asks you to specify character color by entering the name of the color to be used and then entering the ASCII code for the desired color. In Chapter 6, you filled in Table 6-3, listing ASCII codes for the 16 available character colors. It is on page 79. The ASCII code is taken by an INPUT statement as the variable CC, meaning *character color*.

Then it asks you to select border color by entering one of the screen-color codes shown in Table 7-1. The input is taken as BR, meaning *border*.

When all keyboard inputs have been made, line 60 clears the screen. To test this segment, run it. Enter WHITE for character color and ASCII code 5 to select that color. Then enter screen-color code 4 to select a purple border. When the program stops, enter

```
PRINT CC
PRINT BR
```

Both numbers should be displayed in normal light blue because the program hasn't changed any colors yet. If the display looks OK and the codes are what you entered, the segment is probably entered correctly. Now enter

```
70 PRINT CHR$(CC): REM SET CHAR COLOR
80 PRINT:PRINT " CHARACTER COLOR IS "CC$
90 POKE 53280,BR: REM SET BORDER
100 PRINT:PRINT " BORDER ="BR
```

At line 70, the statement PRINT CHR\$(CC) sets character color using the ASCII code number specified earlier in the program as CC. Line 80 displays the character-color number on the screen, in the selected character color. The screen background is the normal blue.

At line 90, the statement POKE 53280,BR sets border color using the screen-color code specified earlier as BR. Line 100 displays the selected border-color number on the screen, in the selected character color. The screen background is still the normal blue color.

Notice that character color is set by *printing* an ASCII code. Border color is set by *poking* a screen-color code.

To test this segment, run the program. Enter WHITE for the color name and ASCII code 5 to select white characters. Enter screen-color code 4 to select a purple border.

Run it. The border should be purple. The characters should be white against a normal blue background because the background hasn't been changed yet. Press RUN STOP-RESTORE to change back to normal screen colors. Now enter

```
110 REM CHANGE BACKGROUND
120 FOR BK=0 TO 15:POKE 53281,BK
130 PRINT CHR$(19):FOR L=1 TO 5:PRINT:
  NEXT L: REM LOCATE CURSOR
140 PRINT " BACKGROUND ="BK
150 PRINT:PRINT:PRINT
160 PRINT " TO CONTINUE, PRESS SPACE BAR"
170 GET A$:IF A$="" GOTO 170
180 NEXT BK
```

Screen-Display Codes

Line 120 sets up a loop to run from 0 to 15, using BK as the counter. BK means *background*. The second statement on that line pokes the value BK into the memory location that sets background color. As this loop runs, all 16 background colors will be displayed with the selected character color and border color.

This segment will display the screen-color code number for each background color while that color is being used. The statements that produce this display must be inside the BK loop, so the display will change each time the background color changes.

Line 130 locates the cursor so the background-color number will be displayed at the same location on the screen, each time. The first statement on that line homes the cursor without clearing the screen. Then a loop using L as the counter moves the cursor down 5 rows on the screen. Line 140 prints a message stating the background color being used, which is BK.

The display should remain on the screen while you look at it and make notes of the color numbers, if you wish. Line 160 tells the user how to display the next combination of colors when he is ready. Line 170 is a GET routine that waits for the next keystroke.

The program says, press the space bar to continue, but nearly any key will do. Some programmers use PRESS ANY KEY when nearly any key will do. I prefer to give definite instructions. Also, there are some keystrokes that a GET routine does not recognize.

When the user presses the space bar, ASCII code 32 is produced. The GET routine grabs it and names it A\$. Nothing is done with A\$. The purpose of the GET routine is just to stop the program until the user presses a key.

To test this segment, run the program. Enter WHITE for the name of the character color and 5 to select it. Enter 0 for border color.

The display should show white characters against a black background, with a black border. Press the space bar to see the next color combination. You see white characters against a white background. Press the space bar again. The background changes to red.

If the program is doing that, it's OK. Finish running it by pressing the space bar until you have seen all 16 background colors with white characters and a black border. The program ends with white characters against a light-gray background.

The next program segment restores normal screen colors and asks the user if he wishes to run it again. Enter

```
190 REM DO AGAIN?
200 POKE 53280,14:POKE 53281,6:
    REM SET NORMAL BR AND BG
210 PRINT CHR$(154): REM NORMAL CHAR
220 PRINT CHR$(147):PRINT:PRINT
    " ALL BACKGROUND COLORS DISPLAYED"
230 PRINT" USING "CC$" CHARACTERS"
240 PRINT" AND BORDER "BR:PRINT:PRINT
250 PRINT:PRINT" DO AGAIN? Y/N?"
260 GET A$:IF A$="" GOTO 260
270 IF A$="Y" GOTO 10
```

Lines 200 and 210 set normal screen colors again. Then a message is displayed. Line 260 uses a GET routine to get a keystroke. If the user presses key Y, the program loops back to line 10 and repeats.

SELECTING COMBINATIONS OF COLORS

Run the program. When you have seen white characters with all background and border colors, select black characters and run the program again. If you have enough patience, you can see and make notes about all possible combinations.

There are 16 possible character colors, backgrounds and borders. That is a total of 16x16x16

combinations, which is 4,096 different combinations. If you display all of them with this program, it will take an hour or two just to look at them.

One of the problems of programming with 4,096 possible color combinations is simply choosing one. Some have limited usefulness, such as white characters on a white background. Some color combinations are more pleasing or more effective than others. Some make characters easy to read, some difficult.

I suggest that you write down a few color combinations you like. Then when you are programming, you can just refer to your notes to select colors that you know work well together. This is worth doing because it will save time later.

SCREEN-DISPLAY CODES

Each character on the screen is made by a pattern of dots. There are two character sets that can be displayed on the screen or printed on paper by a Commodore printer—upper-case-and-graphics and upper-and-lower-case. The character sets are referred to as Set 1 and Set 2. Set 1 is upper-case-and-graphics. Set 2 is upper-and-lower-case. They are shown in Tables 7-2 and 7-3. Character sets are stored in ROM memory.

What is stored is the dot pattern used to make each character. Each dot pattern is represented by a screen-display code. The code numbers are 0 to 255. Screen-display codes are not the same as ASCII codes. There are some things that you can do using screen-display codes that can't be done any other way.

SELECTING A CHARACTER SET

Each of the two character sets uses the same group of screen-display-code numbers—0 to 255. The display controller must first be told which character set to use. Then each code number represents a specific character in that set. The display can use only one character set at a time, which is why all characters on the screen must be from one set or the other.

There are three ways to switch character sets. From the keyboard, press Commodore-SHIFT. That works in the immediate mode or when a program is running, but it must always be done by fingers on the keys.

In Chapter 5, Table 5-2 shows control symbols that select one of the two character sets. These can be used in a program line in the immediate mode.

This chapter will show how to switch character sets using screen-display codes.

HOW A KEYSTROKE BECOMES A CHARACTER ON THE SCREEN

Translating a keystroke into a displayed character or some other action by the computer is done in a series of steps. In the first step, the computer notices which key is pressed.

The computer knows when you press special keys, such as Commodore, CTRL and SHIFT. It also knows when you press ordinary keys such as the letter A or the number 7.

When the computer receives keystrokes from the keyboard, it translates the keystrokes into screen-display code numbers that will cause the desired character to be displayed or cause the desired action of the computer.

To deliver a screen-display code to the display controller, the computer places the code number at a specified location in memory, in a *screen memory map*. It is a map of locations on the screen and what should be displayed at each location. The display controller looks at each location on the map to get the code and then displays the corresponding character at the corresponding location on the screen.

WAYS TO DISPLAY A CHARACTER

If you type the letter A on the keyboard, the chain of events just described happens. A keystroke is translated into a screen-display code; the display code is placed at the correct location in the screen memory map; and the desired character appears on the screen.

Another way is to poke a screen-display code directly into the screen-memory-map location so it will produce a character on the screen. By doing that, you can produce a character on the screen without ever typing that character. Instead, you type the display-code number for that character and poke it into memory.

TABLE 7-2
SCREEN-DISPLAY CODES USING UPPER-CASE-AND-GRAPHICS MODE
 (A reversed space is printed in front of reversed characters.)

Code	Display	Code	Display	Code	Display
0	@	57	9	114	T
1	B	58	:	115	
2	B	59	:	116	T
3	B	60	<	117	---
4	B	61	=	118	---
5	B	62	>	119	---
6	B	63	?	120	---
7	B	64	!	121	---
8	B	65	*	122	---
9	B	66	!	123	---
10	B	67	!	124	---
11	B	68	!	125	---
12	B	69	!	126	---
13	B	70	!	127	---
14	B	71	!	128	---
15	B	72	!	129	---
16	B	73	!	130	---
17	B	74	!	131	---
18	B	75	!	132	---
19	B	76	!	133	---
20	B	77	!	134	---
21	B	78	!	135	---
22	B	79	!	136	---
23	B	80	!	137	---
24	B	81	!	138	---
25	B	82	!	139	---
26	B	83	!	140	---
27	B	84	!	141	---
28	B	85	!	142	---
29	B	86	!	143	---
30	B	87	!	144	---
31	B	88	!	145	---
32	B	89	!	146	---
33	B	90	!	147	---
34	B	91	!	148	---
35	B	92	!	149	---
36	B	93	!	150	---
37	B	94	!	151	---
38	B	95	!	152	---
39	B	96	!	153	---
40	B	97	!	154	---
41	B	98	!	155	---
42	B	99	!	156	---
43	B	100	!	157	---
44	B	101	!	158	---
45	B	102	!	159	---
46	B	103	!	160	---
47	B	104	!	161	---
48	B	105	!	162	---
49	B	106	!	163	---
50	B	107	!	164	---
51	B	108	!	165	---
52	B	109	!	166	---
53	B	110	!	167	---
54	B	111	!	168	---
55	B	112	!	169	---
56	B	113	!	170	---

TABLE 7-2 (continued)

Code	Display	Code	Display	Code	Display
171		200		229	
172		201		230	
173		202		231	
174		203		232	
175		204		233	
176		205		234	
177		206		235	
178		207		236	
179		208		237	
180		209		238	
181		210		239	
182		211		240	
183		212		241	
184		213		242	
185		214		243	
186		215		244	
187		216		245	
188		217		246	
189		218		247	
190		219		248	
191		220		249	
192		221		250	
193		222		251	
194		223		252	
195		224		253	
196		225		254	
197		226		255	
198		227			
199		228			

TABLE 7-3
SCREEN-DISPLAY CODES USING UPPER-CASE-AND-LOWER-CASE MODE
(A reversed space is printed in front of reversed characters.)

Code	Display	Code	Display	Code	Display
0	@	27	[54	6
1	a	28	£	55	7
2	b	29]	56	8
3	c	30	↑	57	9
4	d	31	←	58	0
5	e	32		59	1
6	f	33	!	60	2
7	g	34	"	61	3
8	h	35	#	62	4
9	i	36	\$	63	5
10	j	37	%	64	6
11	k	38	&	65	7
12	l	39	'	66	8
13	m	40	(67	9
14	n	41)	68	0
15	o	42	*	69	1
16	p	43	+	70	2
17	q	44	,	71	3
18	r	45	-	72	4
19	s	46	.	73	5
20	t	47	/	74	6
21	u	48	0	75	7
22	v	49	1	76	8
23	w	50	2	77	9
24	x	51	3	78	0
25	y	52	4	79	1
26	z	53	5	80	2

TABLE 7-3 (continued)

Code	Display	Code	Display	Code	Display
81	Q	141	■	201	■
82	R	142	■	202	■
83	S	143	■	203	■
84	T	144	■	204	■
85	U	145	■	205	■
86	V	146	■	206	■
87	W	147	■	207	■
88	X	148	■	208	■
89	Y	149	■	209	■
90	Z	150	■	210	■
91	+	151	■	211	■
92	*	152	■	212	■
93	-	153	■	213	■
94	×	154	■	214	■
95	÷	155	■	215	■
96	■	156	■	216	■
97	■	157	■	217	■
98	■	158	■	218	■
99	■	159	■	219	■
100	■	160	■	220	■
101	■	161	■	221	■
102	■	162	■	222	■
103	■	163	■	223	■
104	■	164	■	224	■
105	■	165	■	225	■
106	■	166	■	226	■
107	■	167	■	227	■
108	■	168	■	228	■
109	■	169	■	229	■
110	■	170	■	230	■
111	■	171	■	231	■
112	■	172	■	232	■
113	■	173	■	233	■
114	■	174	■	234	■
115	■	175	■	235	■
116	■	176	■	236	■
117	■	177	■	237	■
118	■	178	■	238	■
119	■	179	■	239	■
120	■	180	■	240	■
121	■	181	■	241	■
122	■	182	■	242	■
123	■	183	■	243	■
124	■	184	■	244	■
125	■	185	■	245	■
126	■	186	■	246	■
127	■	187	■	247	■
128	■	188	■	248	■
129	■	189	■	249	■
130	■	190	■	250	■
131	■	191	■	251	■
132	■	192	■	252	■
133	■	193	■	253	■
134	■	194	■	254	■
135	■	195	■	255	■
136	■	196	■		
137	■	197	■		
138	■	198	■		
139	■	199	■		
140	■	200	■		

The display controller doesn't care how the code number got there. It displays the desired character. Offering more than one way to do something increases the versatility and usefulness of the computer.

SCREEN MEMORY MAP

The space in memory, where screen-display codes are placed, is organized exactly like the display space on the screen.

The locations on the screen where a character can be displayed are numbered, starting with screen location zero at the top-left corner. Moving to the right, along the top row, location numbers are 0, 1, 2, 3 and so forth, up to 39.

The second row has locations 40 to 79. The bottom row has locations 960 to 999. The screen can display 1000 characters simultaneously, at screen locations 0 to 999.

The screen memory map begins at location 1024 in memory and has space for 1000 screen-display codes, each representing one character on the screen. Location 1024 in memory corresponds to location 0 on the screen. Location 1025 in memory is location 1 on the screen, and so forth. Screen location numbers and the screen memory map are shown in Figure 7-4, next page.

If you poke a screen-display code into location 1024 in memory, that character will appear instantly at location 0 on the screen—provided the display controller also knows what color the character should be.

COLOR MEMORY MAP

The screen has two maps in memory. One is the character to be displayed at each location. The other is the color of each character at each location.

The color memory map starts at location 55296 in memory and has 1000 locations, corresponding to the 1000 locations on the screen. In memory, location 55296 holds the character-color code for location 0 on the screen. This map is shown in Figure 7-5, page 97.

To display a character, the screen-display code for that character must be in the screen memory map and the color code for that character must be at the corresponding location in the color memory map.

COLOR CODES

Numerical color codes used in the color memory map are the same as those used earlier in this chapter to set border and background colors. They are shown in Table 7-1.

PLACING SCREEN-DISPLAY CODES AND COLOR CODES IN THE MEMORY MAPS

These codes can be placed in the two memory maps in two ways:

By Ordinary Keystrokes—Select a character color at the keyboard, such as CTRL-1, and then type the desired character. When that has been done, all following keystrokes that display something cause the computer to perform two actions.

It puts the screen-display code for the character to be displayed in the screen memory map. It also puts the color code for that character in the color memory map.

To demonstrate that, set the keyboard to type upper-case-and-graphics characters. Press CTRL-1 to select black characters.

Move the cursor to location 0 at the top-left corner of the screen and press key A. Peek at both memory maps to see what is stored there. Move the cursor down to a clear area on the screen and enter

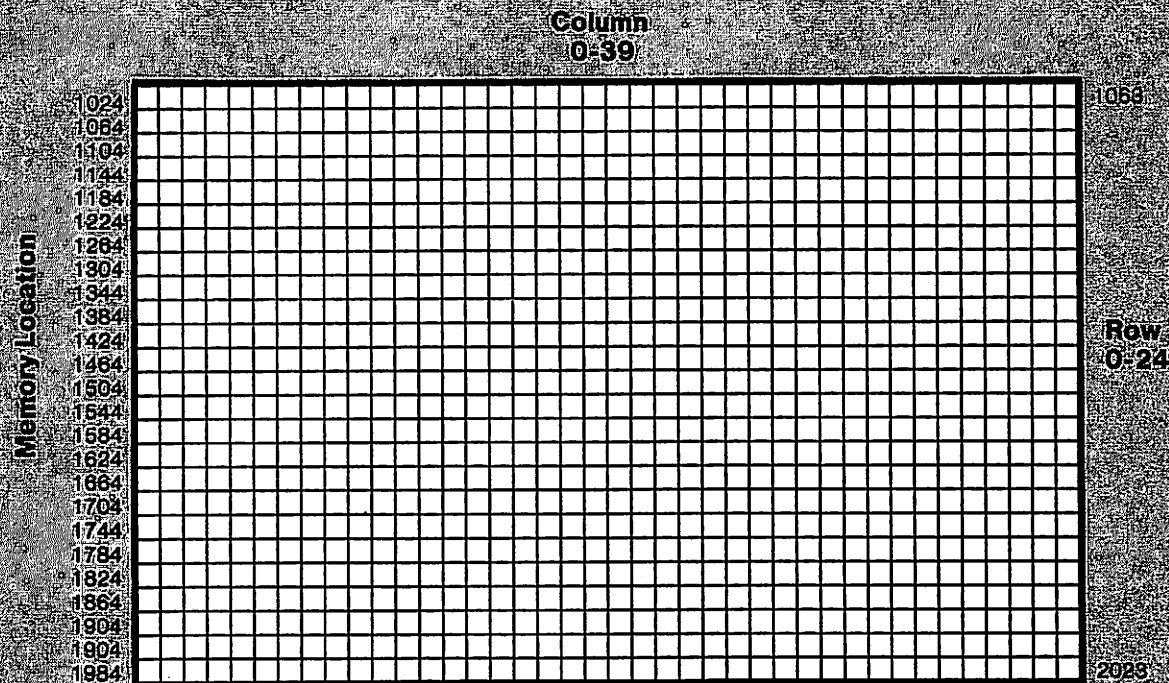
```
PRINT PEEK(1024)
```

That location in the screen memory map corresponds to the top-left corner of the screen. It holds the number 1. Table 7-2 shows that the number 1 is the screen-display code for the letter A in character set 1. Move the cursor down to a clear area on the screen and enter

```
PRINT PEEK(55296)
```

That's the corresponding location in the color memory map. It holds the number 0. Table 7-1 shows that the color code for black is 0.

FIGURE 7-4
SCREEN-LOCATION NUMBERS & SCREEN MEMORY MAP

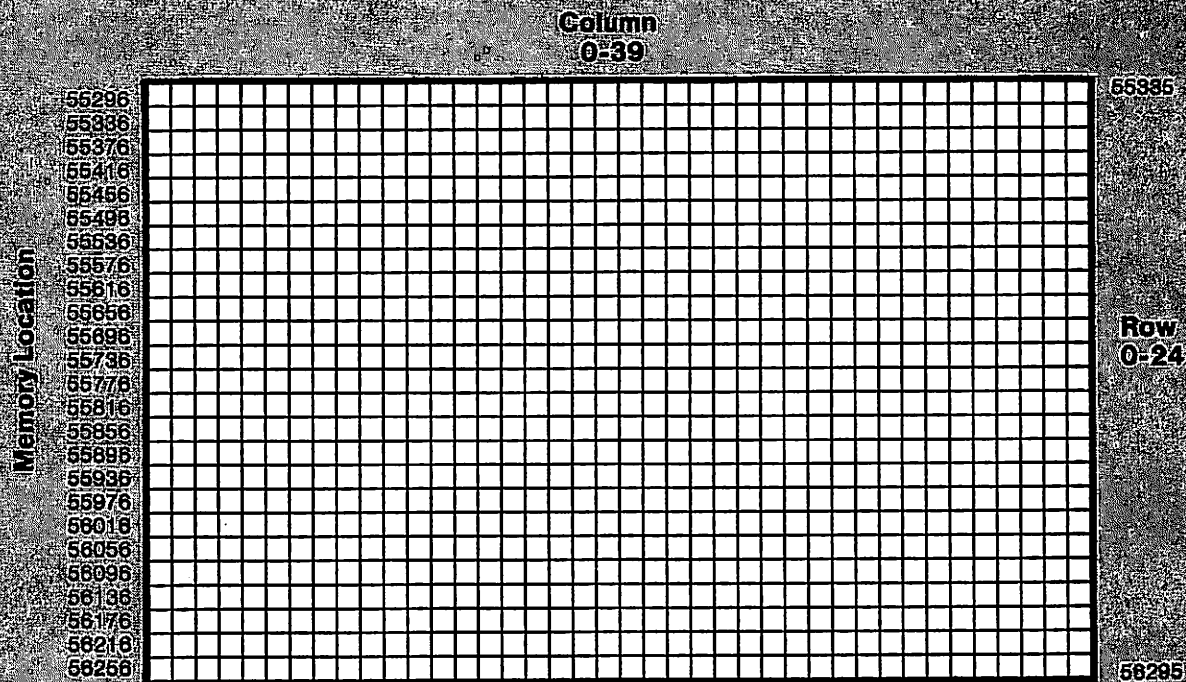


To place a character at any location on the screen, the computer puts a screen-display code in the corresponding location of this screen memory map in the computer memory. Location 1024 on this map in memory corresponds to location 0 on the screen — the top left corner.

There are 1000 print locations on the screen and 1000 locations in the screen memory map.

Characters can be displayed by poking screen-display codes into the appropriate locations in this screen memory map. You must also poke character color into the color memory map. Figure 7-5.

FIGURE 7-5 COLOR MEMORY MAP



After a screen-display code is placed in the screen memory map, the character color for that location on the screen is placed in this color memory map. Location 55296 on this map in memory corresponds to location 0 on the screen—the top left corner.

There are 1000 print locations on the screen and 1000 locations in the screen memory map.

If there is no character on the screen at a certain location, and you display a character by poking it into the screen memory map, you must also poke the character color into the corresponding location of this color memory map.

If a character is already being displayed, you can change the character by poking a different screen-display code into the screen memory map. If you do not poke a new color into the color memory map, the new character will be displayed using the same color as the character that it replaced.

Screen-Display Codes

Those codes, in both maps, were placed there by ordinary keystrokes—setting color by CTRL-1 and then pressing A.

Poking Codes—You can produce the same result by poking those two codes into the two maps. It takes one poke to put the screen-display code into the screen memory map and another poke to put the color code into the corresponding location in the color memory map.

HOW TO POKE CHARACTERS ONTO THE SCREEN

Let's display a white B at the top-left corner of the screen. The screen-display code for B, using character set 1, is the number 2. The color code for white is 1. With the cursor in a clear area near the bottom of the screen, enter

```
POKE 1024,2: POKE 55296,1
```

The first statement pokes the screen-display code for B into the screen memory map. The second statement pokes the color code for white into the corresponding location of the color memory map.

The result is a white B at the top-left corner of the screen. Peek at both memory maps to verify the numbers.

CHANGING A CHARACTER ALREADY DISPLAYED

If there is a valid number in both memory maps, a character is displayed. In that case, you can change the screen memory map to change the character, or the color memory map to change the color, or both.

Change the B at the top-left corner to a C. Enter

```
POKE 1024,3
```

Change the color of the C from white to black by entering

```
POKE 55296,0
```

DISPLAYING CHARACTER SETS

Tables 7-2 and 7-3 show all characters in the two character sets and the screen-display code numbers. It's interesting to see the characters on the screen, and it helps you learn to use screen-display codes. The next demonstration program will do that.

Screen location 500 should be near the middle of the screen because location 0 is at the top-left corner, and location 999 is at the bottom-right corner. The characters will be placed there, one at a time.

To do that, the screen-display code for each character will be poked into the screen memory map at location $1024 + 500$, which is 1524. The desired character color will be poked into the color memory map at location $55296 + 500$, which is 55796.

These two map locations are each 500 locations from the start of each map, so they represent the same screen location. Enter

```
NEW
10 PRINT CHR$(147)
20 FOR CHAR=0 TO 255
30 POKE 1524,CHAR
40 POKE 55796,14
50 FOR DL=1 TO 400:NEXT DL
60 NEXT CHAR
```

Line 10 sets up a loop to run from 0 to 255. The counter is CHAR, meaning character number. The computer will actually use CH for that variable name.

Line 20 pokes CHAR into memory location 1524. As CHAR changes from 0 to 255, each character in the selected character set will be displayed, provided the color is also specified.

Line 30 pokes a color code into the color memory map at the corresponding location. It pokes the number 14 into that location. Color 14 is light blue, the normal character color. All characters will be displayed in that color.

Line 40 is a delay loop to slow down program execution. Line 50 is the bottom of the outer loop.

Using Commodore-SHIFT, set the keyboard to character set 1, which is upper-case-and-graphics. Then run the program.

All characters in set 1 are displayed at location 500 on the screen. Change the keyboard to character set 2 and run it again. All characters in the upper-and-lower-case mode are displayed.

This program works, but just barely. Here are some things it doesn't do. It doesn't select the character set to be displayed from the program—you have to do that before running it. It doesn't show the code number for each character as it displays the character. The following sections show how to put those features into the program.

SELECTING A CHARACTER SET FROM A PROGRAM

This is done by poking. POKE 53272,21 selects upper case and graphics. POKE 53272,23 selects upper and lower case. I suggest that you put these numbers in your notes.

CHOOSING A CHARACTER SET

Instead of expecting you to choose a character set before running the program, this program should ask you to choose it while the program runs. It will get the reply, then proceed accordingly. Start a new program by entering:

```
NEW
10 PRINT CHR$(147)
20 PRINT"DISPLAY CHARACTER SET 1 OR 2?"
30 PRINT
40 PRINT"PRESS 1 OR 2."
50 GET REPLY$:IF REPLY$=" " GOTO 50
60 IF REPLY$="1" THEN POKE 53272,21
70 IF REPLY$="2" THEN POKE 53272,23
80 IF REPLY$<>"1" AND REPLY$<>"2"GOTO 50
```

Line 20 asks a question and line 40 tells the user how to reply. Line 50 uses a GET routine to wait for the reply keystroke. This keystroke is taken as the string variable REPLY\$. The computer will actually store it as RE\$.

At line 60, if REPLY\$ is 1, character set 1 is selected by the POKE statement on that line. At line 70, if REPLY\$ is 2, character set 2 is selected. These lines use an equals sign as a relational operator to test REPLY\$.

Line 80 is an *error trap*. Before running this part of the program to test it, read the following discussion.

ERROR TRAPS

Error traps are program lines that prevent program failure due to errors. Suppose the user presses key 3 by mistake. At line 50, REPLY\$ will become 3. Line 60 will not execute because REPLY\$ is not 1. Line 70 will not execute because REPLY\$ is not 2.

The user made an invalid selection from the keyboard because there are only two character sets. Line 80 prevents the program from continuing until a valid selection has been made.

If lines 60 and 70 do not execute, then line 80 will execute. It says *if REPLY\$ is not 1 and it is not 2, then jump back to line 50 and get another keystroke*. This error trap causes the program to ignore invalid keystrokes. Normally, the user will make another selection, usually without the typing error.

A better error trap results if the program tells the operator that a keyboard error was made. It

could display something like SORRY, WRONG NUMBER. PLEASE SELECT AGAIN. I didn't do that because I wanted to keep this program short.

What you have entered is not a complete program, but it is a complete *routine*. It gets a keyboard input and checks it to see if it is valid.

When writing programs, it's a good idea to test each routine immediately. If it doesn't work, it is easier to fix then, rather than later when you may have forgotten why you wrote it that way.

Test this routine by running the program. Respond to the question on the screen by pressing key 3. Then press other character keys. The program won't do anything until you press key 1 or 2. If it doesn't work that way, check your typing.

MORE IMPROVEMENTS

When a valid selection has been made, the program should show the character set and code number being displayed. Enter the rest of the program, starting at line 90.

```
10 PRINT CHR$(147)
20 PRINT"DISPLAY CHARACTER SET 1 OR 2?"
30 PRINT
40 PRINT"PRESS 1 OR 2."
50 GET REPLY$:IF REPLY$=" " GOTO 50
60 IF REPLY$="1" THEN POKE 53272,21
70 IF REPLY$="2" THEN POKE 53272,23
80 IF REPLY$<>"1" AND REPLY$<>"2"GOTO 50
90 FOR CHAR=0 TO 255
100 PRINT CHR$(147)
110 PRINT"SET "REPLY$,"CODE","CHARACTER"
120 PRINT
130 PRINT,CHAR
140 POKE 1024+144,CHAR
150 POKE 55296+144,14
160 PRINT
170 PRINT"PRESS SPACE BAR TO CONTINUE"
180 GET A$:IF A$=" " GOTO 180
190 NEXT CHAR
```

List the program and check your typing. When you list a program, you will often see a typing error that you didn't notice when you entered it. When the program is listed, there is a blank row between lines 80 and 90. That's because line 80 completely fills one row on the screen. That produces an automatic carriage return before displaying line 90.

The new lines do essentially what the earlier demonstration did, with some additions. Line 90 sets up a loop to run from 0 to 255. Line 100 clears the screen on each pass through the loop.

Line 110 prints headings on the screen to explain what is being displayed. This line prints a list of four items, using commas for automatic tabs. The first item printed is the word SET followed by a space. The next item is the character-set number, which was given the name REPLY\$ at line 50. Then a comma tabs over to the next print zone. The word CODE is printed. Another comma causes an automatic tab and the word CHAR is printed. Line 120 puts a blank row below the headings.

Notice that line 110 used a comma to tab the word CODE over to the second print zone on the screen. Line 130 prints the value of the loop counter, CHAR, using a comma between PRINT and CHAR. The comma tabs the number CHAR over so it is directly beneath its heading.

Lines 140 and 150 poke each screen-display code onto the screen by putting the correct numbers at the correct memory locations. The characters will be displayed in light blue.

The character produced should be below the word CHARACTER on the screen. When poking screen-display codes into the screen memory map, the location of each character on the screen is

determined by where the code is poked into the memory map. In this program, the codes are poked into location $1024 + 144$, which is 144 locations from the top-left corner. I chose that number so the character would be displayed beneath the heading, **CHARACTER**.

Later in this chapter, I will show you how to find memory-map locations for any desired screen location.

The color code, 14, is poked into the color memory map 144 locations from the beginning. It causes the character to be displayed in light blue.

After each screen-display code has been displayed, along with the character that it produces, the program stops at line 180 and waits for a keystroke. When the space bar or any other character key is pressed, the **NEXT** statement at line 190 jumps back to line 100 to run the loop again. Line 100 clears the screen again.

Run it and select a character set to be displayed. If it doesn't seem to work correctly, check your typing on the new lines that you just entered. Run it again to display the other character set.

List the program and look it over. It does a lot of things. It clears the screen, asks a question, gets a reply from the keyboard and makes a decision based on testing that reply. The decision is to display one of the two character sets.

It has an error trap to prevent keyboard error. It pokes characters onto the screen by poking the screen-display code into one memory map and the character color into the other map. It puts characters on the screen by two methods: **PRINT** statements and poking.

It uses a **GET** routine to get a keystroke that is used to make a decision. It uses another **GET** routine just to stop the program and wait for a keystroke to start it again.

Everything in this program is practical real-world programming. If you understand this program, you are doing fine. You won't have difficulty with the rest of this book, and you will learn to write programs.

ROWS AND COLUMNS

In the preceding program, I told you where to poke codes for a desired location on the screen. You should learn how to figure that out.

The way to find a location in the memory maps is to find it on the screen first. There are two ways to specify a place on the screen where a character can be displayed. One is the *location number*—from 0 to 999.

The other is to organize the screen into rows and columns mentally. Rows are horizontal. Rows on the screen are numbered from 0 to 24. Columns are vertical. Columns are numbered from 0 to 39.

To illustrate the idea of rows and columns, home the cursor. Then, move it down to row 4. It starts at row 0, so you move it to row 1, 2, 3, 4.

Then move the cursor to column 10, remembering that it is in column 0 when it is fully to the left on the screen. When you have done that, the cursor is at row 4, column 10.

When you are writing a program, it is much easier to visualize the screen as rows and columns, rather than a series of location numbers from 0 to 999. But when you are poking characters into the memory maps, you need to know the screen-location number, not rows and columns. The memory-map locations are calculated by adding the screen-location number to the starting map-location number. In the preceding demonstration, location 144 on the screen was used to display the character. The corresponding map locations were $1024 + 144$ and $55296 + 144$.

Following is an easy way to convert screen row and column numbers into screen-location numbers. Then, the screen-location numbers can be converted into memory-map locations.

To begin, run the program in memory. Select character set 1. When it stops the first time, break out of the program by pressing **RUN STOP**. The character being displayed is @. Move the cursor to the top-left corner of the screen. While counting rows, move it down to the same row as the @ symbol. It is on row 3. The top row is 0.

Then count columns while moving the cursor to the right until it is on top of the @ symbol. It is at column 24. This program displays characters at row 3, column 24.

This formula will find the screen-location number for any place on the screen specified by row and column:

$$\text{SCREEN LOCATION} = (40 * R) + C$$

R is the row number, and C is the column number.

For example, if the cursor is at row 3, column 24, calculate the screen-location number like this:

$$\text{SCREEN LOCATION} = (40 * 3) + 24$$

This is $120 + 24$, or 144. Therefore, location 144 is the same as row 3, column 24.

When planning a screen display, use row and column numbers. It is easier to visualize tables or graphics by that method. Write down the row and column numbers for each part of the display.

When writing program lines to poke screen-display codes into memory maps, the first step is to convert row and column numbers into screen-location numbers by the method just demonstrated.

When you have a screen-location number, calculate the corresponding map locations with these formulas:

$$\text{SCREEN MEMORY MAP LOCATION} = 1024 + \text{SCREEN LOCATION}$$

$$\text{COLOR MEMORY MAP LOCATION} = 55296 + \text{SCREEN LOCATION}$$

For example, if the screen-location number is 144, then

$$\begin{aligned} \text{SCREEN MEMORY MAP LOCATION} &= 1024 + 144 \\ &= 1168 \end{aligned}$$

$$\begin{aligned} \text{COLOR MEMORY MAP LOCATION} &= 55296 + 144 \\ &= 55340. \end{aligned}$$

You can have the computer do the arithmetic for you. Lines 140 and 150 in this demonstration program let the computer do it.

Here is another example. In the first demonstration program in this chapter, I suggested that screen location 500 should be near the center of the screen because it is about halfway between location 0 and location 999. It's row 12, column 20.

$$\begin{aligned} \text{SCREEN LOCATION} &= (40 * 12) + 20 \\ &= 500. \end{aligned}$$

CHARACTER SETS FOR PROGRAMMING

You can write programs using either character set. BASIC words are normally written in capital letters, but they can be typed in lower case. The computer will recognize and execute them.

It's a good idea to write a program using the same character set that you will use when executing the program. The demonstration program in this chapter works OK with either character set. The question and instructions were displayed in upper case with set 1 and in lower case with set 2, but they are readable either way.

That isn't always true. Here is an example: Select the upper-and-lower-case mode. Enter
print "Hello"

That works fine. Now select the other character set by pressing Commodore-SHIFT. The letter H in Hello is displayed as SHIFT-H in that set, which is how you entered it from the keyboard. In set 1, SHIFT-H is a graphics symbol. The display doesn't make sense.

When you write a program in one character set, it may be a good idea to make sure that the program is executed using the same character set. You know how to do that by a POKE statement in the program. Lines 60 and 70 of the program in memory do that.

WHEN TO USE POKES

Some of the things you can do with POKE statements can also be done in other ways. You can locate the cursor to print something on the screen using either control symbols or control codes, and you can select character color the same way.

The only way to control background and border colors is by poking numbers into the correct memory locations.

When something can be done in more than one way, take your choice. If there is only one way, then you need to know and use that method.

REVIEW

Look through this chapter again for review. The demonstration programs are the first complicated programs in this book. Be sure you understand them.

The new BASIC words in this chapter were INPUT, POKE and PEEK. Look them up in Appendix B.

IS IT NECESSARY TO POKE BOTH CHARACTER AND COLOR?

To poke a character onto a blank location on the screen, this chapter says that you must poke a screen-display code into the screen memory map and a color code into the color memory map.

For early Commodore 64 models, that is not true. You can display a character in light blue, at a blank location on the screen, just by poking the screen-display code into the screen memory map.

For current models, if a character already exists on the screen, you can poke a new character into the screen memory map at that location and the new character will replace the old. It will be the same color as the old character.

For current models, if a character does not already exist on the screen, you must poke both the screen-display code and the character color into memory. If you don't poke both, the character is not displayed.

Instructions in this chapter will work on all Commodore 64 models.

8

How To Input Data From The Keyboard

Most programs can't do anything useful without data from the outside world. I am using the word *data* to mean anything you want to put in—numbers, letters, words, poetry, astrological truths, even long and preachy book manuscripts.

Your computer can receive data over a phone line from a databank or another computer. You can obtain data on a disk or cassette and play it into your computer. But the most common way to *input* data is from the keyboard.

The only data that a computer can use is data in memory. You can input it from the keyboard or from some other source, or put it on program lines in the program itself. But, for data to be useful, it must be assigned a name and placed in memory. When the program needs the data, it calls for it by name.

INPUT

This is a BASIC word that tells the computer to *get ready* to receive data. This discussion shows how to input data from the keyboard. The word INPUT must be used in a program statement that also supplies a variable name for the data to be received. The type-designation symbol, which is part of the variable name, tells the computer what kind of data will be input—integer number, decimal number or string.

As data is typed, it is displayed on the screen, but is not placed in memory using its variable name until the RETURN key is pressed. This allows you to correct typing errors before entering the data. The RETURN keystroke tells the computer to accept what has been typed and put it into memory.

The maximum number of keystrokes that can be entered is 80—in other words, two rows on the screen. INPUT cannot be used in the immediate mode. It must be in a program line.

INPUT with a Decimal-Number Variable Name—The statement INPUT X tells the computer to receive data from the keyboard and assign it the name X. It tells the computer that the data will be a decimal number because X is a decimal-number variable name. You can use any valid decimal-number name such as RENT or EXPENSE. Enter this program:

```
NEW  
10 INPUT X  
20 PRINT X
```

Run the program. When line 10 executes, it displays a question mark as a signal to the user that he is expected to enter something from the keyboard. The program stops at line 10 and waits for the input.

Type 12333. Notice that whatever you type is displayed on the screen. If you make a typing error, and notice it before entering the data, you can correct it by deleting and retyping or by overtyping incorrect characters. Change the characters to 12345. Then press RETURN to enter what you typed. That places the number 12345 in memory with the name X.

When you have responded to the INPUT statement at line 10, the program moves to line 20. It prints whatever it stored under that variable name.

Run the program again. Enter ABC. The error message says REDO FROM START. The program will not accept ABC because it is not a number. The numeric variable name X requires a number.

The program did not stop and return to the immediate mode, as it does for some errors. It tells you the problem and jumps back to line 10 to wait for a valid input. This error message is really an error trap because it keeps the program running even if the user makes a mistake.

Press RUN STOP to interrupt program execution. It doesn't work. The program insists on a valid input before it will do anything else. Enter any number to get the program running again.

When a program is executing an INPUT statement and waiting for data from the keyboard, the only way to break out of the program is to press RUN STOP-RESTORE.

List the program and run it again. When you see the question mark, don't type anything. Just press RETURN. You entered nothing, but the computer displays a zero. When you are dealing with numbers, *nothing* is a zero. Run it again and enter

123.45

The variable name, X, is a *decimal-number* variable name—also called *floating-point* variable name. It will accept numbers with decimal fractions.

Positive numbers print with a space before the number, called a *leading space*. Run the program and enter

– 123.45

Negative numbers use that space to display a minus sign. Run it again and enter

12,345

The comma acts as a delimiter. The INPUT statement stops accepting data at the comma. It accepts the number 12—as you can see. The error message warns you that not all of the number was accepted but the program does not stop. Later in this book, I will show you a better way to handle commas in data.

INPUT With an Integer Variable Name—To use an INPUT statement with an integer variable, change line 10 like this:

10 INPUT X%

Run the program and enter 12345. It displays zero. Something is wrong! List the program and figure out what the problem is.

I did that to illustrate an important rule and a common program error. The rule is that you must be consistent in using variable names. The common error is not being consistent. Line 10 receives X% from the keyboard, but line 20 displays X. X% is not the same variable as X.

Line 10 worked correctly. It accepted 12345 from the keyboard and put it into memory as X%. To see that, enter

PRINT X%

X% is in memory, with the correct value. But line 20 printed the value of X. Because X received no value in this program, line 20 printed a zero.

When you change a variable name in a program, be sure to change it everywhere in the

How To Input Data From The Keyboard

program. Change X in line 20 to X%. Run the program and enter

123.99

The integer variable name X% accepts only the whole-number part of the number input. That's what it is supposed to do. To say it another way, X% accepts the *next smaller* whole number. 123 is the next smaller whole number below 123.99.

Run it again and enter

—123.99

The integer variable received the whole number smaller than the number that was input. —124 is the next smaller number below —123.99.

Try inputting numbers with commas.

INPUT with a String Variable Name—Change X% to X\$ everywhere in the program. Run it and enter

ABC

The program accepts ABC as a string and displays it. Notice that it was not necessary to use quotation marks to define ABC as a string. The INPUT statement defined it as a string by receiving the characters into the string variable name X\$.

You may wonder what would happen if you did use quotation marks. Try it. Run the program and enter

"ABC"

The quotation marks don't do anything. They are not needed, and they are ignored. Run the program and enter

12345

The program accepts the numbers as a string variable. Run it again and enter

123ABC

It also accepts any combination of letters and numbers as a string variable. Run it again and enter

ABC, DEF

The computer stops accepting data at the comma, even if you are entering data into a string variable name.

Earlier, you saw that you can put a comma in a string if you type the string in quotation marks, such as Y\$="LMN, OPQ", as part of an *assignment* statement. There is a difference between an assignment statement and inputting data from the keyboard. To emphasize that difference, list the program and add lines 30 and 40.

```
10 INPUT X$
```

```
20 PRINT X$
```

```
30 Y$="ABC, DEF"
```

```
40 PRINT Y$
```

Run it that way. Type and enter

ABC, DEF

The INPUT statement balks at the comma. Line 20 causes an error message and then displays ABC. The assignment statement accepts the comma along with the characters behind it.

You may be thinking that it is difficult to remember when the computer will accept a comma and when it won't. That's true—when you are learning.

For now, you don't have to remember all of these details. If you write a program and have trouble with commas, you will remember some of these facts even though you thought you had forgotten them. Even if you don't remember the details, you will remember that commas do funny

things. When necessary, look up commas in the index of this book and refresh your memory.

Or, you can ask the computer to remind you of the rules. Each of the simple demonstrations in this book is just a way of asking the computer to show you how it operates. When you need to know how a computer handles a comma, or anything else about how it works, write a little routine to find out. When you have done that a few times, the facts will stick in your mind.

INPUT & GET

INPUT and GET statements are two ways for a program to receive data from the keyboard. This is a review of their differences.

GET statements accept only one keystroke at a time. They assign it to the variable name specified by the GET statement and put it into memory instantly. It is not necessary to press RETURN to enter the keystroke. The keystroke is not displayed by the GET statement. A question mark is not displayed by a GET statement to tell the user to type something. While a GET routine is waiting for a keystroke, the cursor is invisible.

A GET statement does not stop program execution. It gets whatever is in the keyboard buffer. To stop the program and wait for a keystroke requires a GET *routine*. You have been using GET routines. They are useful for quick replies to questions on the screen, for inputs that can be made with a single keystroke, to put a pause into a program, and other purposes that you will see later.

INPUT statements require pressing RETURN to enter what was typed. INPUT statements accept up to 80 keystrokes, counting the RETURN keystroke. INPUT statements display what was typed and allow changing characters before they are entered. INPUT statements stop program execution. INPUT statements display a question mark. The cursor is visible to the right of the question mark.

INPUT statements are useful to accept long inputs, such as names and addresses or sentences.

TELLING THE USER WHAT TO INPUT

When a program wants the user to type something at the keyboard, it should provide instructions. With GET routines, the instructions should be displayed by a program line ahead of the GET routine. You have done it that way in several of the preceding demonstrations.

With an INPUT statement, the automatic question mark is a reminder that something should be entered. It doesn't say specifically what to enter.

You can use an earlier line in the program to tell the user what to type, or you can include instructions in the INPUT statement. When you do that, the instructions are called a *prompt*.

The form is

INPUT "*prompt*"; *variable name*

Please enter this example:

NEW

10 INPUT "PLEASE ENTER YOUR NAME "; NAME\$

20 PRINT NAME\$

A prompt that is part of an INPUT statement must be enclosed in quotation marks and followed by a semicolon. After the semicolon, type the variable name. Run it and enter your name. The purpose of line 20 is just to show that the INPUT statement worked.

A prompt may use more than one screen row, but not more than two. The computer will display the prompt, but it may wrap around to the next row, which wouldn't look very good. Here is way to solve that problem. Enter

NEW

10 PRINT "PLEASE ENTER YOUR NAME "

20 PRINT "IN THE SPACE BELOW."

30 INPUT NAME\$

40 PRINT NAME\$

How To Input Data From The Keyboard

That puts the prompt on two screen rows and allows most of a row for a very important person's very long name.

INPUTTING A LIST OF DATA ITEMS

Just as you can print a list of items, you can also input a list. Enter

```
NEW
10 INPUT A$, B$, C$
20 PRINT A$, B$, C$
```

Using commas to separate the variable names in the INPUT statement is necessary. If you omit them, a syntax error message results.

Commas in the PRINT statement of line 20 are not necessary. They are used here just to provide automatic tabs. Run the program. When you see the question mark, enter the letter A.

The program displays two question marks, indicating that more data items are needed to satisfy the INPUT statement. Enter B and then C.

That finishes the list of data items to be input, so the program moves to line 20 and displays the three data items entered. Those data items are in memory with three different variable names. Remove the commas from line 20 and run the program again, the same as before. Only the display is changed.

Mixing Numeric and String Variables—An INPUT list can use mixed variable types. Change the program like this:

```
10 INPUT A$, B%, C
20 PRINT A$, B%, C
```

Now, line 10 calls for a string, an integer number and a decimal number. If the input for any of those variables does not match the variable type, an error may result.

If you write programs like this, be sure that the instructions are specific and easy to understand. Run the program. Enter 123.45 for each variable. That works fine.

Run the program again and enter 123.45 APPLES for each variable. That is OK for the input to A\$ but not to B%. The REDO FROM START error message means start all over and input again to A\$, even though that input was OK. This can be confusing to an inexperienced keyboard operator.

It is often better to take all inputs as strings, and sort them out later. Press RUN
STOP - RESTORE.

VAL()

This is a BASIC word that changes a string expression into its numeric VALue. It allows you to input strings and then change them into numerics. Enter

```
X$="123"
PRINT X$
PRINT VAL(X$)
```

VAL() can be used to take the value of a string expression having both numbers and letters. It starts with the first character and evaluates the expression from left to right until it encounters a character that is not a number. Enter

```
X$="123ABC"
PRINT X$
PRINT VAL(X$)
```

If the first character in the string is a letter or symbol that cannot be evaluated as a number, the value of the string is zero. Enter

```
X$="123.45"
PRINT X$
PRINT VAL(X$)
```

Notice that VAL(X\$) retains the decimal fraction. Because VAL(X\$) is a number—the value of X\$—the computer can do arithmetic with it. Enter

```
PRINT VAL(X$)+2
```

It is usually better to assign the value of X\$ to a variable name and then do arithmetic using the variable name. X\$ is still in memory as the string 123.45. Enter

```
PRINT X$
PRINT VAL(X$)
Y=VAL(X$)
PRINT Y
PRINT Y+2
```

Because Y is a decimal-number variable name, it accepts 123.45 as the value of X\$. By using an integer variable name to take the value of X\$, you can do arithmetic with only the whole-number part. Enter

```
PRINT VAL(X$)
Y%=VAL(X$)
PRINT Y%
PRINT Y%+2
```

This method can be used to simplify inputting a single data item or a list of data items. Enter three identical data items, all as strings, and then change them to other data types. Enter

```
NEW
10 PRINT CHR$(147)
20 INPUT A$, B$, C$
30 B=VAL(B$)
40 C%=VAL(C$)
50 PRINT A$, B, C%
```

Run the program. For each variable in line 10, enter 123.45. Enter it three times. A\$, B\$ and C\$ are all 123.45.

Line 30 assigns the value of B\$ to a decimal-number variable name. Line 40 assigns the value of C\$ to an integer variable name.

Line 50 prints A\$ as originally input. Then it prints B, the decimal-number value of 123.45. Then it prints C%, which is the whole-number part of 123.45.

Run the program again and enter 123.45 APPLES for each variable. The program runs OK. Each of the three variables gets what it is supposed to from the input, and there are no error messages.

The advantage of inputting data as strings is that you don't have to worry about matching data types when entering the data.

USING A GET ROUTINE TO RECEIVE NUMBERS

So far, GET routines have been used only to receive a string and to stop the program while waiting. This should be a familiar routine: 70 GET A\$:IF A\$="" GOTO 70.

GET statements can also be used with a numeric variable such as A%. There may be a problem in stopping the program to wait for a keystroke. To see the problem, enter

```
NEW
10 PRINT CHR$(147)
20 PRINT"PRESS A NUMBER KEY"
30 GET A$:IF A%=0 GOTO 30
40 PRINT A%
```

Line 30 looks like it should work. Run the program and press 5. Run it again and press 0. When

How To Input Data From The Keyboard

A% is 0, line 30 continues looping. This routine will accept any number except 0. If 0 should be accepted as a valid input, this routine won't work.

A more serious problem with this routine is that it stops the program if a letter key is pressed by mistake. Run it and press any letter key.

It is better to take the input to a GET routine as a string variable and then change it to a numeric, if that's what the program needs. Change the program in memory so it looks like this:

```
NEW
10 PRINT CHR$(147)
20 PRINT "PRESS A NUMBER KEY"
30 GET A$:IF A$=" " GOTO 30
35 A%=VAL(A$)
40 PRINT A%
```

That program will accept 0 as a valid input. It won't stop if a letter key is pressed. It gives the letter a value of zero and continues to run. To prevent accepting letter keystrokes, put an error trap at line 32 to ignore keystrokes that are not one of the number keys:

```
32 IF ASC(A$)<48 OR ASC(A$)>57 GOTO 30
```

As you know, the numbers 0-9 have ASCII codes from 48-57. Line 32 checks the code to be sure that a number was input. Now, the program accepts any number key, including zero, but no other keystroke.

USING A GET STATEMENT TO RECEIVE A LIST OF DATA ITEMS

You can also input multiple data items to a single GET statement. Each data item can be only a single keystroke. A statement such as GET A\$,B\$,C\$,D\$ will get four characters from the keyboard buffer. There isn't any way to stop it between characters.

To use a statement like that, you must write the program so that the desired four characters are in the keyboard buffer before the GET statement executes. This routine isn't very practical, but it works. Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT "TYPE A FOUR-LETTER WORD"
30 FOR DL=1 TO 1000:NEXT
40 GET A$,B$,C$,D$
50 PRINT A$ B$ C$ D$
```

The delay loop at line 30 prevents the computer from taking characters from the keyboard buffer until the loop runs to completion. That requires you to type characters while that loop is running. Then line 40 will get the characters from the keyboard buffer.

Run it and don't enter anything. When the delay loop at line 30 is finished, line 40 executes. It gets four nulls from the keyboard buffer. Line 50 displays the four nulls, which are invisible.

Run it again and enter AA before the delay loop finishes. It displays AA followed by two nulls.

Run it again and enter AAAA before the delay loop finishes. If you typed four characters fast enough, they are displayed.

USING A GET ROUTINE IN A LOOP

In some programs, there is an advantage in taking keyboard input one character at a time, using a GET routine. It allows the program to examine each character and test it or process it in some way. The way to do that is in a loop. Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT "PLEASE TYPE YOUR NAME"
30 PRINT "AND THEN PRESS RETURN":PRINT
```

(Program continued on next page.)

```
40 GET L$:IF L$="" GOTO 40
50 IF L$=CHR$(13) GOTO 100
60 PRINT L$;
70 GOTO 40
100 PRINT:PRINT
110 PRINT"THANK YOU"
```

This program has a loop. The top is line 40 and the bottom is line 70. It gets keystrokes, one at a time, until the user has entered all characters in his name. Line 40 gets keystrokes and names them L\$, meaning *letter* string.

Line 50 tests each keystroke. CHR\$(13) is the ASCII code for RETURN. When line 50 detects a RETURN keystroke, it is a signal that the keyboard operator has finished typing his name. If L\$ is CHR\$(13), the program stops accepting keystrokes and jumps out of the loop to line 100.

If the keystroke is not CHR\$(13), line 60 executes. It displays the character typed. The semicolon at the end of line 60 holds the cursor on that row. Line 70 loops back to line 40 to get another keystroke.

Notice that each keystroke is tested as soon as it is entered, to see if it is CHR\$(13). Run the program and enter your name.

HOW TO PUT DATA INTO A PROGRAM

This chapter has shown how to input data from the keyboard while a program is running. Some programs need data to use while running, but they don't need it from the operator. For example, a program that converts inches to feet needs to "know" that there are 12 inches in a foot. But, the user shouldn't have to tell the program that.

One way to put data in a program is by assignment statements. Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT"ENTER THE NUMBER OF INCHES"
30 INPUT IN
40 CF=12: REM CONVERSION FACTOR
50 FT=IN/CF
60 PRINT
70 PRINT IN"INCHES ="FT" FEET"
```

In line 30, IN is the variable name for inches. In line 40, the number 12 is assigned to the name CF, which means *conversion factor*. This is the data that the program needs to convert inches into feet. It is built into the program by the assignment statement at line 40. Line 50 does the conversion. Line 70 displays the result.

Run it and enter any number of inches.

CONSTANTS

Values that never change are called *constants*. The number of inches in a foot is a constant. In line 40, CF is the name that represents the constant 12. It isn't a variable name because constants don't vary. It can be called a *numeric-constant name*, or just a *name*.

READ DATA

If a program needs a lot of constants, they can be put into the program in a more compact way than using one assignment statement for each constant. This is done with READ statements and DATA statements.

A DATA statement is a list of data items separated by commas. A READ statement tells the computer to find the DATA statement, wherever it is in the program, and read the data items into memory. The READ statement must supply names for each data item in the DATA statement. The

How To Input Data From The Keyboard

names must be separated by commas. Here's how it works. Enter

```
NEW
10 READ A
20 PRINT A
30 DATA 1
```

Line 10 tells the computer to find the DATA statement, READ one item, give it the name A and put it in memory. When line 10 executes, all of those things happen. Line 20 displays A by calling it from memory. The data is on line 30. It is just the number 1. Change the program like this:

```
10 READ A,B,C,D
20 PRINT A,B,C,D
30 DATA 1,2,3,4
40 PRINT "DONE"
```

Line 10 reads four data items and supplies names for each item. Line 30 provides the four items, separated by commas. Line 20 gets the four items from memory and displays them. Line 40 demonstrates that the program executes line 20, then line 40.

Line 30 is never executed as a program line, in the usual way. The computer just refers to the data on that line while executing line 10. A DATA statement is a reservoir of data that the program can use.

Matching Numbers of Items—In this example, the number of “reads” is determined by the number of names provided by the READ statement. The computer expects to find a data item for each name. If it doesn't find enough data to satisfy the READ statement, an error message results. Change line 10 like this and run it:

```
10 READ A,B,C,D,E
```

The READ statement is looking for five data items, but the DATA statement provides only four. The error message is ?OUT OF DATA ERROR IN 10.

If a program provides more data items than a READ statement needs, some data items are not read. But the computer does not consider that to be an error. Change line 10 like this and run it again:

```
10 READ A,B,C
20 PRINT A,B,C,D
30 DATA 1,2,3,4
40 PRINT "DONE"
```

Now there are three reads into four data items. Run the program.

There is no error message, but there may be a programmer's error. Line 20 displays values for A,B,C,D but the program established values only for A,B,C. The variable D did not receive a value in this program, so the computer displays a zero for its value. This is a normal action of the computer but it may be an incorrect display.

In other words, displaying zero for the value of D may be valid in your program or it may be misleading. It depends on the purpose of the program and the significance of the variables. It is valid in this program because it demonstrates that D did not receive a value.

DATA POINTER

When the computer is reading a data list, it moves an electronic pointer along the list. The pointer always designates the *next* data item to be read.

When a program with READ DATA statements first starts to run, the pointer points to the first data item. In the program you just ran, the program ended with the pointer pointing at data item 4 because only three items were read.

If a later line in the program has another READ statement, the computer will return to the DATA statement and resume reading data where it stopped—at the next item that has not yet been

read. To illustrate that, enter new lines 50 and 60.

```
50 READ D
60 PRINT A,B,C,D
```

Line 50 gets the last data item in the DATA statement. Line 60 displays all four data items. There is no more data.

USING THE DATA OVER AGAIN

In some programs you will want to read the same data from more than one place in the program. To illustrate that, let's read the same data again using different names. Enter lines 70, 80 and 90.

```
70 PRINT"FOURTH DATA ITEM READ"
80 READ E,F,G,H
90 PRINT E,F,G,H
```

List the program, check it and run it. Line 80 assigns new names but tries to use the same data again with the new names.

Of course, line 80 produces an out-of-data error. It looks for data in the program and doesn't find any. The electronic data pointer is at the very end of the data list in line 30 because all of that data has already been read by the program.

RESTORE

To read that data again, the pointer must be moved back to the beginning of the data. A RESTORE statement does that.

A RESTORE statement must be executed *after* the first read of the data and *before* the next read of the data. In this program, the last data item is read, the first time, at line 50. The RESTORE statement must follow line 50 but precede line 80. Let's put it at line 75. Enter

```
75 RESTORE
```

Run it. The RESTORE statement resets the pointer and makes the data available again to another READ statement. Because the second READ statement used different names, the original names are still in memory with the original data. The first data item was read first using the name A and then read again using the name E. Enter

```
PRINT A
PRINT E
PRINT B
PRINT F
```

List the program. Line 30 supplies four data items. Line 10 reads three of them. Line 20 displays the three data items that were read, plus another name that has received no value at that point in the program. Then the program displays the word DONE. It really means that only three reads have been done.

Line 50 reads the fourth data item. Line 60 displays all four items. This time, D has the value that it received from the data list. Line 70 reports the status of the program. The pointer is now at the end of the data list.

Line 75 restores the pointer to the beginning of the list. Line 80 reads the data again, supplying four new names. Line 90 displays the values received by the four new names.

A RESTORE statement in a program and the RESTORE key on the keyboard do different things. It is coincidence that they have the same name.

MULTIPLE READ AND DATA STATEMENTS

A program may have more than one READ statement and more than one DATA statement. The READ statements will be executed in order, according to their line numbers.

DATA statements can be anywhere in the program. Programs are easier to read if the DATA statements are near the READ statements, but you can put them at the end of the program if you want to.

If DATA statements are on more than one program line, each line must begin with DATA. Lines are read in numerical order, according to their line numbers—unless a RESTORE statement moves the pointer back to the first data item.

When all data items in one DATA statement have been read, the computer moves the pointer to the beginning of the next DATA statement and waits for the next read.

When a RESTORE statement is executed, the data pointer is moved back to the beginning of the *first* DATA statement in the program.

If you use more than one READ statement, you must be thoughtful about where you put a RESTORE statement. In this program, the RESTORE statement is at line 75. If it were at line 35 instead, the pointer would be reset after the first three data items had been read.

When line 50 executed, it would read data item 1 instead of data item 4. The READ statement at line 80 would begin with data item 2. An out-of-data error would occur because there would not be enough data items left for line 80. Try it that way. Then move the RESTORE statement back to line 75. Run it again to be sure it works correctly.

VARIABLE TYPES IN READ DATA ROUTINES

You can use any of the three variable types in these routines. The data supplied in the data list must match the type designation of the name that receives it. Change the variable name C to C\$ everywhere in the program. Change G to G\$ also, because it will read the same data item. Then change the third data item to ABCD.

Notice that quotation marks are not required around ABCD. The computer treats it as a string because it is read into a string name. This is similar to inputting a string from the keyboard into a string variable name.

List the program and check it. Run the program. It should display ABCD as the third data item.

REVIEW

You can input data from the keyboard, while a program is running. Do it with GET and INPUT statements. You can place data in the program itself, when you are writing it, by assignment statements or by READ DATA routines.

I suggest that you look over this chapter again to be sure you understand all of those methods. In the next chapter, loops are combined with READ DATA statements to build useful data structures called *arrays*.

BASIC words in this chapter are DATA, INPUT, READ, RESTORE and VAL. Look them up in Appendix B.

9 Arrays

An array is a list or table of data items and a way of *managing* the data. An example is a list of names, such as

Arthur
Betty
Charlie
Diane

It is the *management method* that makes it an array. Arrays are stored in memory.

The management plan is simple. If an array is a list of items, each item is identified by its position on the list—such as the third item from the top. A data item is placed in the array by specifying its position in the array and retrieved by specifying its position in the array.

Imagine a set of lockers, such as gymnasium lockers. There are only six and they are stacked—one on top of another in a vertical column of lockers. The top locker is number 1, and the bottom locker is number 6.

Suppose your locker is number 3. You will have no trouble finding it—count down 3 lockers from the top.

The simplest type of array is arranged in a similar way in computer memory. It is a series of locations in memory. Each location is a “locker” that can be used to store something.

It may help to imagine that the location with the smallest number is on top. To find the third location in this array, count three locations from the beginning or top of the array.

ARRAY NOTATION

A special way of writing something is called a *notation*. An array has a name. Each location in an array has a name. A special notation is used to write those two names.

In this book, the notation $X()$ will be used as the name of an array called *X array*. An array name has two parts. The first part is a variable name, such as X . The second part is a set of parentheses with nothing in them. The empty parentheses are a symbol that means *array*. $X()$ means *X array*.

Each location in an array has a name. Location names are similar to array names. Each has two parts. The first part is a variable name, such as X . The second part is the location number in the array, written in parentheses.

A location in an array is written like this: $X(3)$. The number 3, in parentheses, means that this location is the third location in an array called $X()$.

SUBSCRIPTED VARIABLES

In the notation $X(3)$, the number 3 is called a *subscript*. Variables with subscripts are therefore called *subscripted variables*.

In BASIC programs, the *only* use of subscripted variables is to refer to an array. When the computer sees a variable named $X(3)$, it knows automatically that this is location 3 in the array $X()$.

PUTTING VALUES INTO AN ARRAY

A location name in an array is used exactly like an ordinary variable name. It may be given a value by an assignment statement. This is an assignment statement using an ordinary variable name, X . Enter

```
X=7
```

That statement assigned the value 7 to the decimal-number variable name X . The computer stores the number 7 at any convenient place in memory and remembers where it is. Enter

```
PRINT X
```

That statement causes the computer to retrieve the value of X from wherever it is in memory and display it. Retrieving it from memory does not remove it. It is still in memory, stored as X . Enter

```
X(3)=7
```

That statement assigns the value 7 to location 3 of the array $X()$. If that array does not already exist in memory, the computer will create it automatically. The subscript (3) causes that to happen. Enter

```
PRINT X(3)
```

The computer goes to location 3 in the array $X()$, gets the value stored there, and displays it.

Now, the number 7 is stored at two places in computer memory, with two different names: X and $X(3)$.

Only one value can be stored as X . Many values can be stored in the array $X()$. The array can have one value at $X(1)$, another at $X(2)$, others at $X(3)$, $X(4)$, $X(5)$ and so forth.

Those array locations can be filled with values by assignment statements such as

```
X(1)=3
```

```
X(2)=47
```

and so forth.

USING LOOPS WITH ARRAYS

If there are many array locations, it is simpler and faster to fill them with a loop. To demonstrate that, enter

```
NEW
```

```
10 PRINT CHR$(147)
```

```
20 FOR L=1 TO 5
```

```
30 X(L)=2*L
```

```
40 NEXT L
```

Line 20 sets up a loop to run five times. The name of the loop counter is L , which means *location*.

Line 30 is an assignment statement “driven” by the loop. The first part of the statement is $X(L)$. The value of the loop counter, L , is “captured” inside the parentheses. As the loop operates, $X(L)$ becomes $X(1)$, $X(2)$ and so forth, up to $X(5)$.

The second part of the assignment statement is $2*L$. This is just a convenient way to provide values to put into those array locations. When L is 1, $2*L$ is 2. The value of $X(1)$ becomes 2.

On pass 5 through this loop, line 30 says $X(5) = 2 * 5$. The value of $X(5)$ becomes 10. To verify that the program worked, run it. Then enter

```
PRINT X(1)
PRINT X(2)
PRINT X(3)
PRINT X(4)
PRINT X(5)
PRINT X(6)
```

The first five locations have values because the program put them there. Location 6 has no value because the program didn't put any value there.

Instead of using individual statements to display the content of the array $X()$, you can use another loop. Enter lines 50-70.

```
10 PRINT CHR$(147)
20 FOR L=1 TO 5
30 X(L)=2*L
40 NEXT L
50 FOR D=1 TO 5
60 PRINT X(D)
70 NEXT D
```

The second loop uses a different name for the counter, but the number of loop operations is the same. The counter name, D , stands for *display*. It will count from 1 to 5. Line 60 will display the values of locations $X(1)$ to $X(5)$ as loop counter D changes from 1 to 5. Run it.

This program uses a loop to *fill* five locations of an array. The loop *supplies* five different names for those five locations by "plugging" different numbers into the location name. The names are $X(1)$, $X(2)$ and so forth.

Then it uses another loop to *read* the array. The second loop supplies the names of five locations to line 60. The value stored at each of those locations is displayed by line 60.

FILLING AN ARRAY FROM THE KEYBOARD

The preceding demonstration filled each array location with a number that was two times the value of the loop counter. That was just to demonstrate how to use loops to fill and read arrays.

In a real-world program, you may want to fill an array with inputs from the keyboard. This will demonstrate that. List the program in memory. Change line 30 as indicated here:

```
10 PRINT CHR$(147)
20 FOR L=1 TO 5
30 INPUT X(L)
40 NEXT L
50 FOR D=1 TO 5
60 PRINT X(D)
70 NEXT D
```

As you know, an INPUT statement must supply a variable name to receive the data input from the keyboard. Line 30 supplies the name $X(L)$. That name does two things. It receives the data typed from the keyboard. Then it tells the computer to put the data in the array $X()$ at location $X(L)$.

When L is 1, location $X(1)$ is filled with whatever you input from the keyboard. Then location $X(2)$ is filled, and so forth.

Run the program. It will request five keyboard inputs because the loop runs five times. Enter 11

for the first data item, 22 for the second, 33 for the third, then 44 and 55.

When the second loop operates, those numbers will be displayed.

CHOOSING ARRAY VARIABLE NAMES

Array variable names have type declarations, the same as ordinary variable names. You have been using the decimal-number variable name `X`. Run the program again. When you see the question mark, enter `A`.

You have been in that kind of trouble before. The variable `X` will not accept letters. Enter some numbers to finish running the program.

The type designation of the array name determines what kind of variables you can store in an array. The array `X()` accepts decimal numbers. `Y%()` accepts integer numbers. `Z$()` accepts strings.

It is often simpler to use string arrays to store all data as strings. Then you can change strings back to numbers as needed.

If *any* data item stored in an array is a string, then the array name must be a string variable name.

WHAT DOES *VALUE* MEAN?

I have been using the word *value* to mean the content of an array location or the value stored as a variable name. It is easy to think of the number 7 or 123.45 as a value.

For string variables and string arrays, we use the word *value* to mean the string stored using that variable name. If `Z$="ABC"` and `NAME$(5)="MARY"`, then the value of `Z$` is `ABC` and the value of `NAME$(5)` is `MARY`.

MAKING A LIST OF EMPLOYEES

Suppose you are writing a program to keep a list of employees at a small company. The program you have been running can be adapted to do that. Please list it. In lines 30 and 60, change `X` to `NAME$` so the program will accept strings.

```
10 PRINT CHR$(147)
20 FOR L=1 TO 5
30 INPUT NAME$(L)
40 NEXT L
50 FOR D=1 TO 5
60 PRINT NAME$(D)
70 NEXT D
```

Run it and input five names: `ART`, `BILL`, `CHUCK`, `DAN` and `EVE`. They should be displayed.

That program requires a lot more adapting before it is practical. What would happen if you ran it again? It would require you to enter five names again. Then it would display the names.

To avoid that problem, you could start the program by asking the user if he wishes to enter or display names. If the choice is display, the program would jump to line 50 and skip over the loop at line 20.

The main problem is that there is no way to store the names. If you turn off the computer, the names disappear from memory. To make record-keeping programs practical, you must have some way to store both the program and the data used by the program—in this case, the data is a list of names.

Permanent storage of programs and data is done by cassette recorders or disk drives. They are discussed later in this book.

CHANGING AN ITEM IN AN ARRAY

The locations in an array are called *elements*. To change an element in an array, just put a new value at that location. The new value will replace the old value. That would normally be done while

a program is running, but you can demonstrate it in the immediate mode. Enter

```
NAME$(3)="CINDY"
```

That should fire CHUCK and hire CINDY. List the program. To display the names, the program should execute starting at line 50. A RUN statement erases everything in memory except the program itself. RUN 50 will run the program starting with line 50, but nothing will be in memory.

A GOTO statement does not erase memory. Execute from line 50 by entering

```
GOTO 50
```

CINDY is on the job.

CLEARING AN ARRAY

To clear an array that has data in it, the data items should be replaced with nothing. If it is a numeric array, nothing is 0. If it is a string array, nothing is "". Let's take BILL off the payroll and not replace him. Enter

```
NAME$(2)=""
```

```
GOTO 50
```

To clear a lot of elements in an array, or the entire array, it would be better to use a loop. Use it to put nothing into each location.

Let's run a loop in the immediate mode to clear locations 4 and 5 of the array. Enter

```
FOR CL=4 TO 5: NAME$(CL)="" :NEXT
```

```
GOTO 50
```

Only ART and CINDY are left.

CLR

The BASIC word CLR means *clear*. It clears out everything in memory except the program. It is normally used on a program line, but can be used in the immediate mode.

If you have a single array in memory and want to clear it, use a CLR statement. If you have more than one array stored in memory and other variables, such as A, B\$ and E%, they will all disappear. If that's OK, use a CLR statement. If not, just clear the array that you want cleared, using a loop.

Clear the entire array by entering

```
CLR
```

```
GOTO 50
```

I think we are out of business. We have no employees.

HOW MANY ELEMENTS CAN AN ARRAY HAVE?

List the program in memory and change NAME\$ to E\$ wherever it appears. E\$ stands for *element* string. This program will be used to find out how many elements this array can hold.

Change lines 20 and 50 so the loops run from 1 to 10.

```
10 PRINT CHR$(147)
```

```
20 FOR L=1 TO 10
```

```
30 INPUT E$(L)
```

```
40 NEXT L
```

```
50 FOR D=1 TO 10
```

```
60 PRINT E$(D)
```

```
70 NEXT D
```

Run it and enter letters of the alphabet, starting with A and ending with J. Those letters should be displayed. Just for fun, enter

```
PRINT"GET ME A "E$(3) E$(1) E$(2)!"
```

You have filled locations 1 through 10 of the array E\$(). Add the letter K to the array by entering

```
E$(11)="K"
```

An error message results: ?BAD SUBSCRIPT ERROR. The subscript is 11 and the computer won't accept it. Try entering

```
E$(0)="K"  
PRINT E$(0)
```

The computer accepts a subscript of zero. This demonstrates two things. Array locations begin at zero. The highest subscript number that can be used with *this* array is 10. There is a way to use subscripts greater than 10.

When the maximum subscript value is 10, there are 11 array locations. Most people don't use array location zero, but it's there any time you want to use it.

DIMENSIONING AN ARRAY

Arrays with maximum subscripts of 10 or less are small arrays. The computer can normally find space in memory for a small array. It just puts the array in some convenient location.

Arrays with subscripts greater than 10 are large arrays. Large arrays may occupy a lot of space in memory. The computer requires advance notice of a large array, so it can plan for it and reserve space in memory for it.

The BASIC word DIM, meaning DIMension, is used to notify the computer that a large array will be in the program. The form is

```
DIM (n)
```

in which *n* is the number of elements in the array.

The maximum number of elements that an array can have is 32767. That would be a very large array and probably wouldn't fit into the available memory.

The DIM statement must execute before the array is created, so DIM statements are usually placed early in the program. Dimension this array to have a maximum subscript of 26 by entering

```
15 DIM E$(26)
```

That changes the program in memory to allow the array E\$() to use a maximum subscript of 26. Now, that array can hold all 26 letters of the alphabet.

Actually, it can hold 27 elements. It could hold the alphabet if it were dimensioned for a maximum subscript of 25, but you would have to begin with E\$(0) = "A". Most people think that E\$(1) should be A because A is the first letter of the alphabet. That's why most people don't use location zero in an array.

To help adjust your thinking, dimension the array for a maximum subscript of 25 and then enter all 26 letters of the alphabet. You will have to change the loop specifications also.

Your programs will take less space in memory if you use location zero in arrays—if you can make the mental adjustment to accept zero as the first number.

DIMENSIONING MORE THAN ONE ARRAY

A dimension statement can be used to dimension more than one array. Here is an example. Don't enter it.

```
20 DIM E$(50), A$(35), X(255)
```

That statement dimensions a list of arrays, separated by commas. It dimensions three types: a string array, an integer array and a decimal-number array.

ARRAYS WITH TWO DIMENSIONS

The array that you have been using is a list. A list is a single column of data. It has one dimension.

Arrays with two dimensions are used to hold data tables with more than one column of data.

Suppose you are writing a program to calculate transportation charges from Midville to neighboring towns. The program will figure the charge at \$1.67 per mile. To do that, it needs the distance to each destination.

That information can be provided by a lookup table that shows the names of nearby cities, followed by the distance in miles to each city. Here is an example:

FARVILLE	98
NEARVILLE	13
BIGVILLE	55
TINYVILLE	60

When this data is placed in memory as an array, the array will have the same organization as the table. There will be two columns of data with four rows in each column. Rows are horizontal. Columns are vertical.

The computer uses a lookup table just like you do. When the program needs to know how far it is to BIGVILLE, it looks in column 1 to find the name BIGVILLE. Then it reads straight across the table to the next column to find the number of miles to that city.

Let's call this array DIST\$(). The type designation means that it will be used to hold strings. Each location in this array is specified by stating its row number and column number in that order. The form is

DIST\$(*r,c*)

in which *r* is the row number and *c* is the column number.

By looking at the table, you can see that DIST\$(3,2) represents the data at row 3, column 2, which is the number 55. DIST\$(2,1) is the name NEARVILLE.

Here is a routine to put the data for the lookup table into the program itself, using READ DATA statements. When the program runs, it will read the data and place it in memory as an array.

Putting the data into memory makes it available to other parts of the program, where it can be used to calculate transportation charges. Begin with program line 100. Enter

```

NEW
100 DATA FARVILLE,98,NEARVILLE,13,BIGVILLE,55,TINYVILLE,60
110 FOR R=1 TO 4
120 FOR C=1 TO 2
130 READ DIST$(R,C)
140 NEXT C
150 NEXT R

```

Line 100 puts the data into a program line. Notice that the first two data items are row 1 of the table, reading from left to right. The next two data items are row 2, and so forth. The order of data items in the data list is important because it affects how the data will be read out of the list and into an array in memory. The data cannot be used until it is in memory.

Line 110 sets up an R loop, meaning *Row*, that counts from 1 to 4. There are 4 rows in the table. Line 120 sets up a C loop, meaning *Column*, that counts from 1 to 2. The table has two columns.

When this program runs, line 110 will select a row number. Then line 120 will select a column number. On the first pass, each loop counter will have the value 1. Row will be 1. Column will be 1.

Think of these two nested loops as *location generators*. As the loops operate, they will produce a series of locations as R,C. They are 1,1 1,2 2,1 2,2 3,1 3,2 and so forth.

Line 130 is a READ statement. As you know, READ statements must provide a name for each

data item being read from the DATA list. The name provided is DIST\$(R,C). That name is a subscripted variable. It tells the computer to put that data item into the array DIST\$() at location DIST\$(R,C).

On the first pass through these loops, the array location will be DIST\$(1,1). The first data item will be read from the data list, which is FARVILLE. The string, FARVILLE, goes into location DIST\$(1,1).

Then line 140 executes. It is the bottom of the C loop, which is nested in the R loop. Line 140 selects the next value of C and jumps back to line 130. R is still 1, but C becomes 2. Line 130 gets the next item on the data list, 98, and plugs it into DIST\$(1,2).

Now, DIST\$(1,1) holds the name FARVILLE and DIST\$(1,2) holds the distance to that place, which is 98 miles.

Line 140 executes again. The C loop has run to completion by counting from 1 to 2, so the program moves to line 150. Line 150 selects the next value of R, which is 2, and jumps back to line 120.

Line 120 starts the C loop all over again. Now, R is 2, and C is 1. Line 130 gets the third item on the data list, which is NEARVILLE, and puts it into array location DIST\$(2,1).

When this routine has finished, eight data items have been read from the data list and placed in the array. Run it.

To be sure the program worked, enter

```
PRINT DIST$(1,1)
PRINT DIST$(1,2)
PRINT DIST$(2,1)
PRINT DIST$(2,2)
```

and so forth until you have seen all elements of the array in memory.

HOW TO DISPLAY AN ARRAY WITH TWO DIMENSIONS

A real-world program would use the array as a lookup table, probably without displaying it. As a learning exercise, let's display it. Do it the easy way. Starting with line 110, overtype all line numbers that begin with 1 so they begin with 2. Then change line 230 to get array elements from memory and display them, like this:

```
200 DATA FARVILLE,98,NEARVILLE,13,BIGVIL
    LE,55,TINYVILLE,60
210 FOR R=1 TO 4
220 FOR C=1 TO 2
230 READ DIST$(R,C)
240 NEXT C
250 NEXT R
```

The second set of loops uses the same loop counters as the first set. That's OK when the first set has run to completion and doesn't need those counters any more. The second set of loops supplies location numbers used to get values from the array and displays them using a PRINT statement.

Run it. All of the values are displayed, but not as a table with two columns. The first step in fixing that problem is to put a comma at the end of line 230 to cause automatic tabs. Do that and run it again.

The first row of the array looks OK, but the second row is on the same line on the screen. After each row of the array is displayed, there should be a carriage return. Each row has been printed when the C loop has run to completion, by counting from 1 to 2. The carriage return should occur each time the C loop runs to completion.

That can be done by a new line 245. Enter

```
245 PRINT
```

List the program. Now, each time the C loop runs to completion, the cursor is moved down to the next row by the PRINT statement on line 245. Run it. When you need to display an array with two dimensions, that's how to do it.

USING AN ARRAY AS A LOOKUP TABLE

A program doesn't need to display the array to use it as a lookup table. Simplify the program again by deleting all lines beginning with 2—the ones used to display the array.

The plan is for the user to enter the name of a town in the array. Then, the program finds that town in the array and reads the distance in column 2 of the same row. Then, it multiplies the number of miles by \$1.67 to find the transportation charge.

The program should begin by asking for the name of the destination. Enter lines 10 and 20 so the program looks like this:

```
10 PRINT CHR$(147)
20 INPUT"ENTER DESTINATION";CITY$
100 DATA FARVILLE,98,NEARVILLE,13,BIGVILLE,55,TINYVILLE,60
110 FOR R=1 TO 4
120 FOR C=1 TO 2
130 READ DIST$(R,C)
140 NEXT C
150 NEXT R
```

Then enter these new lines:

```
160 MILE$=""
170 FOR S=1 TO 4:REM FIND CITY$ IN TABLE
180 IF DIST$(S,1)=CITY$ THEN MILE$=DIST$(S,2)
190 NEXT S
200 PRINT CITY$, MILE$
```

Line 160 initializes a new string variable name, MILE\$ by setting it equal to a null. MILE\$ will be used to hold the number of miles to the destination city.

Line 170 sets up a *search* loop to look down column 1 of the table to see if the city that was entered is on the table. The loop runs from 1 to 4 because that's how many rows the table has. The counter, S, means *search*.

Line 180 gets elements from the array to compare with CITY\$ to see if they match, using the relational operator =. It gets DIST\$(S,1) from the array. As S changes, it will get DIST\$(1,1), DIST\$(2,1) and so forth. The row number, S, changes from 1 to 4. The column number is always 1. Therefore, the program looks only at elements in column 1 of the array.

Each element, as it is obtained by line 180 is compared to CITY\$. If they are equal—meaning the same—then the variable name, MILE\$, is set equal to DIST\$(S,2). This is column 2 of the same row in the array. In other words, it is the distance to the selected city.

This loop will always run to completion. If CITY\$ is found in column 1 of the array, then MILE\$ is set equal to the number of miles to that city. If no match is found, the value of MILE\$ remains "".

A good way to see if a match was found is to test MILES\$. If it is still “ ”, the city that was entered is not in the array. This program doesn’t make that test, but you know how to do it.

Line 200 is a temporary line used to test the program. If it prints the city that was entered, and the correct distance to that city, the program is probably working OK up to line 200. Run it and enter NEARVILLE. The program should display NEARVILLE 13, which is correct. If you aren’t sure of that, look at the data statement on line 100.

Delete line 200. It has served its purpose.

All that remains to be done is multiply the number of miles by the charge per mile of \$1.67 and display the result. The variable name MILES\$ holds the number of miles, but that variable is a string, not a numeric. It must be converted to a number by a VAL statement. Enter

```
200 MILES=VAL(MILES$)
210 AMOUNT=MILES*1.67
220 PRINT:PRINT"THE CHARGE IS $"AMOUNT
```

Run it and enter FARVILLE. The charge should be \$163.66. If you don’t get that result, check your typing. Run it again and enter one of the other destination cities. Run it again and enter a city that is not on the list, such as CHICAGO. The result should be a charge of zero. If this were a real-world program, you should trap that error.

Two important things were demonstrated in that program. The program scanned down column 1 of the array looking for an element that matched CITY\$. You can scan an array by looking down one or more columns, or you can scan it by looking along one or more rows. Scanning is the way to find something in an array.

Once a match was found for CITY\$ in column 1, all of the data in that row applied to CITY\$. This program found CITY\$ in column 1 but took data from column 2 in that row—the number of miles to that city. That’s how to use an array to look up data.

DIMENSIONING A TWO-DIMENSIONAL ARRAY

Arrays with two dimensions don’t have to be dimensioned if the largest subscript that will be used is 10. I recommend dimensioning them anyway, because it makes the program easier to read and understand.

For two dimensions, the form is

`DIM (r,c)`

in which *r* is the number of rows and *c* is the number of columns.

To dimension this array, the statement would be `DIM(4,2)`.

TWO DIMENSIONS DOES NOT MEAN TWO COLUMNS

A two-dimensional array can have as many columns as you wish, up to 32767. As a practical matter, an array that large will not fit into the computer memory.

The statement `DIM(15,7)` dimensions an array with 15 rows and 7 columns.

If you dimension an array that will not fit into memory, the computer displays an out-of-memory error message.

AN ARRAY WITH FOUR COLUMNS

Please list the program you have been running. The number of rows is controlled by the R loop. If it ran from 1 to 15, there would be 15 rows in the array.

The number of columns is controlled by the C loop. If it ran from 1 to 7, there would be 7 columns in the array.

If the data comes from a data list, such as line 100, it would have to supply data suitable for a table with the specified number of rows and columns.

If the data comes from the keyboard, the user must supply entries for the number of rows and columns.

For practice, write a new program that creates an array with 5 rows and 4 columns. Use a string

variable name, such as `ARRAY$`. Fill the array from the keyboard by entering letters, or whatever you wish. Then display the array.

A program that does those things is at the end of this chapter.

ARRAYS WITH MORE THAN TWO DIMENSIONS

Suppose you were writing a program to record sales of various items at several branch offices for each month of the year. If you did that on paper, you would probably make a table for each month. The table for January might look something like this:

SALES FOR JANUARY

ITEM	NEW YORK	CHICAGO	LOS ANGELES
CHAIRS	1012	989	1456
BAR STOOLS	255	300	567
LAMPS	3050	1200	1650

As an array, that data would require 3 rows and 4 columns. The headings are not part of the array. That array could be dimensioned by the statement `DIM JAN$(3,4)`. If any element in an array is a string, the array name must be a string variable name.

The table for February would also have 3 rows and 4 columns. Column 1 would be the same but the numbers in the other three columns would be different. As an array, this could be dimensioned `FEB$(3,4)`. The program would end up with 12 arrays for the 12 months.

If those tables were printed on sheets of paper, there would be 12 sheets. You would probably number them, 1 to 12, to keep them in order. January sales would be page 1, and so forth.

In a program, you can do a similar thing by using an array with three dimensions, such as `SALES$(3,4,12)`. The third dimension, 12, means that there are 12 arrays in memory. Each array has the same number of rows and columns—3,4.

The array `SALES$(3,4,1)` holds data for January. `SALES$(3,4,2)` holds data for February, and so forth. The data in memory would be the same whether stored as 12 arrays with two dimensions or one array with three dimensions.

The advantages of using three dimensions are that the program is a little more compact and it is easier to do some things. For example, writing summary reports for the entire year is a little easier. To total all bar stools sold in Chicago during the year, you would add up

`SALES$(2,3,1)`

`SALES$(2,3,2)`

`SALES$(2,3,3)`

and so forth through the 12 months.

ARRAYS SHOULD BE DIMENSIONED ONLY ONCE

If a program encounters two dimension statements for the same array, it stops running and displays an error message that says `?REDIM'D ARRAY ERROR`. That can happen if there are two dimension statements for that array in the program or if the program loops back and executes the same dimension statement twice.

To avoid that, put all dimension statements near the beginning of your programs, before the beginning of any loops.

HOW TO CHANGE THE DIMENSION STATEMENT FOR AN ARRAY

Occasionally, you may want to use an array in a program, change the dimension statement and use the array again. This can be done with a `CLR` statement in the program. Dimension it once and use it. Then execute a `CLR` statement, dimension it again and use it again.

If you do that, remember that `CLR` clears *everything* from memory except the program. This includes all variables, all arrays, and all `FOR-NEXT` loop specifications. If you execute a `CLR` statement while a loop is running, the computer forgets how many times the loop is supposed to operate.

REVIEW

Arrays are essential in many programs. Large arrays are usually filled by loops and read by loops because loops do those things quickly and in an orderly way.

Even though loops are intimately associated with arrays in most programs, the loops are not part of the arrays. They are merely efficient ways to put data into arrays and get data back again.

An array is a data structure in memory. You can fill it and read it any way you choose.

I suggest that you review this chapter and be sure you understand how to use arrays. In my own experience, arrays with loops were difficult to understand. That's why I put so many detailed descriptions of loop operations in this chapter.

I used to write array routines by copying lines out of other programs. Sometimes my routines worked. Finally, in a sudden burst of comprehension, I got the idea of arrays. After that, it was easy.

I hope it isn't that difficult for you, but if it is, stick with it until you understand arrays. You can't write powerful programs without using arrays and loops.

The new BASIC words in this chapter are CLR and DIM. Look them up in Appendix B.

A PROGRAM TO CREATE A 5x4 ARRAY AND FILL IT FROM THE KEYBOARD

```
10 PRINT CHR$(147)
20 PRINT TAB(4)"THIS PROGRAM BUILDS AND
   DISPLAYS"CHR$(13)TAB(13)"A 5X4 ARRAY"
30 PRINT
40 FOR R=1 TO 5
50 FOR C=1 TO 4
60 PRINT"ENTER DATA FOR ROW"R", COLUMN"C
70 INPUT ARRAY$(R,C)
80 PRINT CHR$(147):PRINT:PRINT:PRINT
90 NEXT C:NEXT R
100 PRINT CHR$(147):PRINT:PRINT:PRINT
110 PRINT "THANK YOU. ARRAY WILL NOW BE
   DISPLAYED"
120 FOR DL=1 TO 1000:NEXT
130 PRINT CHR$(147):PRINT:PRINT:PRINT
140 FOR R=1 TO 5
150 FOR C=1 TO 4
160 PRINT ARRAY$(R,C),
170 NEXT C:PRINT:NEXT R
```

10

Strings And Things

Just as an array is the most important data structure, strings are the most useful form for individual data items. When data is handled as a string, you can do a lot of things with it, including converting to its numerical value by a `VAL()` statement.

HOW TO CONCATENATE STRINGS

You probably never thought that you might want to *concatenate* anything. That word means *connect together*, like boxcars in a train. By concatenating, you can connect one string to another to form a third string. The operator is a plus sign. It is used as follows. In this example, `B$` is a space in quotation marks. Enter

```
A$="HELLO"  
B$=" "  
C$="JOE"  
T$=A$+B$+C$  
PRINT T$
```

If you just want to print three strings, one behind the other, you don't need to concatenate them. Put them in a print list like this. Enter

```
PRINT A$ B$ C$
```

The result is the same, but that print statement did not create a new string. It just printed three strings that already existed.

Concatenation makes a new string by combining strings already in memory. When used with string variables, `+` does not mean add. It means *concatenate*.

COMPARING STRINGS

Strings can be compared in an IF-THEN statement using all of the relational operators: `=`, `<`, `>`, `<=`, `>=` and `<>`. Here's how the computer compares strings.

To begin, it compares the ASCII codes for the first character in each string. The ASCII code for A is 65. The ASCII code for B is 66. When compared as strings, B is greater than A.

After comparing the first characters of each string, it compares the second characters in the same way, then the third, and so forth.

The computer makes a decision when it finds the first two characters with different ASCII code numbers, or when it reaches the ends of the two strings without finding any different characters.

If the strings are the same length, and the characters are identical, the two strings are declared equal. Otherwise, one must be larger than the other. Here are some examples. Don't enter them

AAAB is larger than AAAA

ABC is equal to ABC

ABCD is larger than ABC

When ABCD is compared to ABC, they are equal through the third character of each string. ABCD has a fourth character. The fourth character of ABC is nothing, a null. The ASCII code for a null is 0. The ASCII code for D is greater than zero, so ABCD is larger than ABC.

LEN()

This BASIC word means **LEN**gth. When you put the name of a string in the parentheses, **LEN()** is the number of characters in that string.

You have three strings in memory. Enter

```
PRINT A$ B$ C$
```

B\$ is the space between HELLO and JOE. Enter

```
PRINT A$,LEN(A$),ASC(A$)
```

You see A\$, its length—five characters—and the ASCII code for the first character in A\$. The code for H is 72. Enter

```
PRINT B$,LEN(B$),ASC(B$)
```

You see the space produced by B\$, the length of B\$, and the information that ASCII code 32 represents a space. Enter

```
N$="12345"
```

```
PRINT N$,LEN(N$),ASC(N$)
```

```
N=VAL(N$)
```

```
PRINT N
```

Numbers *entered* as strings in quotation marks, such as N\$, do not have a leading space. When converted to a number, with **VAL()**, or entered as a number, positive numbers print with a leading space. There is also a trailing space, but you can't see it in this display.

Negative numbers print with a minus sign in front and a trailing space.

STR\$()

This BASIC word converts a number into a string. Put a number or a numeric variable name in the parentheses. Enter

```
X$=STR$(12345)
```

```
PRINT X$
```

```
PRINT LEN(X$)
```

The first of those three statements converted the number 12345 into a string variable named X\$. When a number is *converted* into a string by **STR\$()**, the leading space becomes part of the string. This string has six characters. Using a statement such as **X\$=STR\$(12345)** is doing it the hard way if you are typing the statement from the keyboard. **X\$=" 12345"** has the same result—including the leading space if you want one.

Strings And Things

None of the strings has more than 75 characters. Let's concatenate some of them to build longer strings. Enter

```
X$=A$+B$  
PRINT LEN(X$)  
PRINT X$
```

X\$ has 150 characters. Make a still longer string by entering

```
Y$=A$+B$+C$  
PRINT LEN(Y$)  
PRINT Y$
```

Y\$ has 225 characters. Everything seems OK. Make a still longer string by entering

```
Z$=A$+B$+C$+D$
```

That didn't work because the maximum length of a string in memory is 255 characters.

The longest string you can enter from the keyboard at one time is 75 characters. But, you can assemble longer strings in memory by concatenating—up to 255 characters. If you can get a string that long into memory, you can display it on the screen.

LEFT\$()

This BASIC word takes a specified number of characters, starting at the left end of a string.

The form is `LEFT$(t$,n)` in which *t\$* stands for the name of the *target string*—the string to be operated on. The number, *n*, is an integer that specifies how many characters are to be taken from the target string. This action does not actually remove any characters from the string. It just “copies” the specified group of characters into a new variable name. Enter

```
T$="ABCDE"  
PART$=LEFT$(T$,3)  
PRINT T$  
PRINT PART$
```

PART\$ is the leftmost three characters of T\$. The number of characters taken from a target string can be in the range of 0 to 255. If the number is 0, PART\$ will be a null string with zero length. If the number is greater than the length of the target string, PART\$ will be the entire target string.

USING LEFT\$()

Suppose you have a mailing list that uses full names rather than initials and the last name. Perhaps it was obtained from driver's-license applications or some other source requiring full names.

In a mailing list program, F\$ represents the first name, M\$ is the middle name and L\$ is the last name. You are making a list to print address labels for a mailing. To be sure each name will fit on the label, you decide to print only the initials of first and middle names, followed by the last name. Enter

```
NEW  
10 PRINT CHR$(147)  
20 F$="FRANCIS":M$="XAVIER":L$="SMITH"  
30 PRINT LEFT$(F$,1)“. " LEFT$(M$,1)“. "  
L$
```

Line 20 represents one name from the source list. In a real-world program, you would use a loop to get names from the list one at a time for processing. Line 30 prints a shorter version of that name.

After each initial, the program provides a period and a space. Run it.

RIGHT\$()

This is similar to LEFT\$(), except that it takes characters from the right end of the target string. Enter

```
T$="12345"
PART$=RIGHT$(T$,3)
PRINT T$
PRINT PART$
```

PART\$ is the rightmost 3 characters of T\$. The number of characters taken from the string can be in the range of 0 to 255. If the number is 0, PART\$ will be a null string with zero length. If the number is greater than the length of the target string, PART\$ will be the entire target string.

USING RIGHT\$()

Suppose you are writing a program to display a column of decimal numbers on the screen. Columns of numbers look best when they are arranged with decimal points lined up vertically in the column.

This demonstration is a routine that accepts numbers with two decimal places from the keyboard and displays them in a vertical column. The largest number to be entered is 99999.99.

Here is the plan: The routine accepts the keyboard inputs as strings. Then it adds five spaces to the front of each string by concatenating. Adding spaces to the front or back of a string is called *padding*.

Because five spaces are added to the front of each string, there are at least five characters to the left of the decimal point, even if the keyboard operator entered a number with no characters to the left of the decimal point, such as .11.

Then the routine takes the rightmost eight characters of each string. That takes the two characters to the right of the decimal point, the decimal point itself, and five characters to the left of the decimal point. Some of the characters to the left of the decimal point may be spaces added by padding out the numbers.

Then it prints each string, using a TAB() statement to locate the first character. Because all strings have eight characters and the decimal point is the third character from the right, the decimal points line up on the screen. Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT"ENTER THREE DECIMAL NUMBERS"
30 PRINT"IN THE RANGE OF 0 TO 99999.99"
40 PRINT"USING TWO DECIMAL PLACES."
50 PRINT:PRINT"DO NOT ENTER COMMAS."
60 PRINT
70 INPUT N1$,N2$,N3$
80 PAD$="      ":REM 5 SPACES
90 N1$=PAD$+N1$
100 N2$=PAD$+N2$
110 N3$=PAD$+N3$
120 N1$=RIGHT$(N1$,8)
130 N2$=RIGHT$(N2$,8)
140 N3$=RIGHT$(N3$,8)
```

(Program continued on next page.)

```
150 PRINT CHR$(147)
160 PRINT TAB(10)N1$
170 PRINT TAB(10)N2$
180 PRINT TAB(10)N3$
```

Line 80 creates PAD\$, which is five spaces. The number of spaces added by padding should be the same as the maximum number of places to the left of the decimal point in the number being padded.

Lines 90-110 concatenate PAD\$ with N1\$, N2\$ and N3\$. Line 90 says *the new N1\$ is equal to PAD\$ concatenated onto the front of the old N1\$*.

Then, lines 120-140 take the rightmost 8 characters of each string. If the number 99999.99 was entered, none of the spaces padded onto the front are used. If the number entered was .22, all five spaces are included in the eight characters taken by the RIGHT\$() operation. Lines 160-180 print the three strings, each tabbed over 10 spaces.

Run the program and enter these three numbers:

```
1.11
22.22
333.33
```

The decimal points line up vertically, if you entered the program correctly. Run it again and enter any numbers that you wish, following the instructions on the screen.

Suppose the numbers are amounts of money, and you want to display them with dollar signs. You have two choices:

You can concatenate a dollar sign onto each string as it is entered by using lines like this: 72 N1\$="\$"+N1\$. If you do that, the program must take the rightmost nine characters of each padded string. The result will look something like this:

```
$1.11
$22.22
$99999.99
```

The other choice is to print a dollar sign in front of each string in lines 160-170. For example, 160 PRINT TAB(10)"\$"+N1\$

Change the program to try it both ways.

MID\$()

You guessed it. MID\$() takes a chunk out of the middle. You must specify the name of the target string, the first character to be taken and the total number of characters to be taken. The form is MID\$(*t*,\$*f*,*n*) with *f* the location of the first character, counting from the left, and *n* the number of characters to be taken. Enter

```
T$="ABCDEF"
PART$=MID$(T$,2,3)
PRINT PART$
```

Beginning with the second character, three characters are taken as PART\$. They are BCD.

Both *f* and *n* may be in the range of 0 to 255. If *f* is larger than the number of characters in the string, the result is a null string with zero length.

If *n* is zero, the result is a null string. If *n* calls for more characters than exist in the target string, starting at *f*, then the remainder of the string is taken, no matter how long it is. If *n* is omitted, the remainder of the string is taken, starting with *f*.

A MID\$() operation is used for several purposes. One is to display part of a string. Another is to

take a string apart and put it back together again in a different way. Another is to search through a string, looking for a specified set of characters. I'll demonstrate all of those uses.

DISPLAYING THE TIME

The Commodore 64 has a built-in timer that starts from zero when you turn the computer on. It runs continuously, except when a cassette recorder is being used to receive data from the computer or send data to the computer. Output of the timer is available in two ways.

TI—One timer output is the numeric variable TI. This presents elapsed time in sixtieths of a second, which Commodore calls *jiffies*. A jiffy is 1/60 second. To watch jiffies for a while, enter

```
NEW
10 PRINT TI
20 GOTO 10
```

Run it. When you agree that time flies, break out of the program. To display elapsed time in seconds, change line 10 to read

```
10 PRINT TI/60
```

Run it. The numbers are seconds, with decimal fractions. You can use TI to time events, such as how long a delay loop takes to run. Enter

```
10 TS=TI
20 PRINT"STARTED AT "TS/60"SECONDS."
30 FOR DL=1 TO 1000:NEXT
40 TF=TI
50 PRINT"STOPPED AT "TF/60"SECONDS."
60 ET=TF-TS
70 PRINT"DELAY TIME WAS "ET/60"SECONDS."
```

When a loop does nothing but count, it takes about 1 second to count to 1000. Try other delays, if you wish.

TI\$—The output of the built-in timer is also available as the string variable TI\$. It has six digits, representing hours, minutes and seconds. It is a 24-hour clock.

The digits 093322 mean 9:33:22, which is 22 seconds past 9:33 in the morning. The digits 161359 mean 16:13:59 on a 24-hour clock, which is 4:13:59 in the afternoon. Enter

```
PRINT TI$
```

That time started when the computer was turned on. It does not relate to actual time unless you set it by an assignment statement.

TI\$ can be set using an assignment statement in the immediate mode. Suppose you want to set it to 9:30 AM exactly. Enter

```
TI$="093000"
PRINT TI$
```

Wait a few seconds and enter

```
PRINT TI$
```

Let's write a program to display the time in a more readable form. Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT"THE TIME IS "LEFT$(TI$,2)" : "
MID$(TI$,3,2);
30 PRINT" AND "RIGHT$(TI$,2)" SECONDS"
40 PRINT CHR$(19):GOTO 20
```

Line 10 clears the screen and homes the cursor, placing it at row 0 on the screen. Because line 10 is a print statement and the carriage return is not suppressed, the cursor moves to the beginning of row 1 on the screen after line 10 executes. The data printed by this program appears on row 1.

Line 20 prints the leftmost two characters followed by a colon. Then it uses MID\$() to take characters three and four, which are printed, followed by a semicolon to suppress a carriage return. That prints hours and minutes in the conventional form. Line 30 then prints the rightmost two characters, which are seconds.

Line 40 prints CHR\$(19). This homes the cursor without clearing the screen. Because there is no semicolon after that PRINT statement, the cursor ends up at the beginning of row 1 again.

Line 40 then jumps back to line 20. This prints the time message again, on top of the preceding display. Only changed characters, such as seconds, will appear to change in the display. Overprinting can be done in this case because all time messages have the same number of characters.

Run it. Then break out of the program. Using CHR\$(147) instead of CHR\$(19) in line 40 will do almost the same thing. It will leave the cursor on row 1, so each time message overprints the preceding message. But, it erases the screen each time the program loops back, which causes visible flicker. Change line 40 to read

```
40 PRINT CHR$(147):GOTO 20
```

Run it that way and then break out of the program. When you are displaying messages in a loop, and the messages have the same number of characters, overprinting produces a better display than clearing the screen between printings.

As an exercise, write a program that converts the value of TI\$ from a 24-hour clock to a 12-hour clock. Display the time according to the 12-hour clock, using AM and PM. My version of that program is at the end of this chapter.

SEARCHING FOR A KEYWORD

In search routines, one string is searched to see if it contains another, shorter, string. The string being searched is usually called the *target string*. What is being searched for is usually called a *keyword* even if it is more than one word. You can search a target string for the keyword *John* or the keyword *John Smith*. The technique is the same.

Word-processing programs use search routines to find all occurrences of a word or phrase in a document and replace it with a different word or phrase. For example, you can change *John Smith* to *Bill Jones* throughout a document.

A search operation is done with a loop. Enter

```
NEW
10 PRINT CHR$(147)
20 T$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 KW$="JKL"
40 COUNT=24
50 FOR S=1 TO COUNT
60 IF MID$(T$,S,3)=KW$ THEN PRINT"FOUND
   "KW$" AT "S
70 NEXT S
```

Lines 20 and 30 define the target string and the keyword string. Line 40 sets a variable, COUNT, to 24. It is how many searches will be made. Line 50 sets up a loop to control the search. The loop counter is S, which means *Search*. The number of search operations will be from 1 to COUNT.

Line 60 does the searching. It will start at the beginning of T\$ and test characters 1, 2 and 3 in T\$ to see if they match the three characters in KW\$. Then it tests characters 2-5, 3-6, 4-7, and so forth. The last test is characters 24-26.

The IF statement in line 60 says IF MID\$(T\$,S,3). In that statement, S specifies the first character to be examined. The number of characters to be examined is 3.

When the S loop operates, the first value of S is 1. Line 60 looks at MID\$(T\$,1,3), which is the first three characters of the target string.

When S is 2, line 60 looks at MID\$(T\$,2,3), and so forth. The last test is MID\$(T\$,24,3), which looks at characters 24, 25 and 26 in the target string.

If any group of three characters in the target string matches the three characters in KW\$, the program announces that KW\$ was found and states where the first character of KW\$ is located in T\$.

Check your typing and run the program. It finds JKL at location 10 in T\$. List the program. By counting from the beginning of T\$, you can see that J is the tenth character in T\$.

Change KW\$ in line 30 to RST instead of JKL. Run it again. Change KW\$ to AMX. The program won't find that keyword.

Recipes—That search was specific for a target string with 26 characters and a keyword with three characters. It searches from 1 to 24. There is no point in searching further because there are only three characters left in the target string when S is 24.

If the target string had 50 characters and the keyword had 12 characters, that routine wouldn't work. It wouldn't search all the way through the target string because it stops at character 24. It couldn't find the keyword because it would take groups of three characters at a time and compare them to a keyword with 12 characters.

It is common for the user of a keyword-search program to specify both the target string and the keyword. Therefore, the programmer doesn't know in advance how long they will be. To cope with that, a generalized search routine is used. It looks like a programmer's nightmare, but it works. This is it:

```
40 COUNT=LEN(T$)-LEN(KW$)+1
50 FOR S=1 TO COUNT
60 IF MID$(T$,S,LEN(KW$))=KW$ THEN PRINT
   "FOUND "KW$" AT "S
```

Line 40 sets COUNT to a value high enough so all of the target string is searched, but not more. It works for target and keyword strings of any length. If LEN(T\$) is 26 and LEN(KW\$) is 3, then LEN(T\$)-LEN(KW\$)+1 evaluates to 24. The search will stop when the loop counter is 24. That will examine characters 24, 25 and 26.

Line 60 searches the target string, using values of S from 1 to COUNT. It examines groups of characters whose length is LEN(KW\$). If the keyword has three characters, it will search in groups of three. If LEN(KW\$) is 12, it will examine groups of 12 characters, looking for a match.

Change lines 40 and 60 in the program to use the generalized search routine. Change KW\$ back to "JKL". List it and check your typing. It's easy to make mistakes in lines like these. Run it again. If typed correctly, it works the same as before.

At line 30, change KW\$ to GHIJKLMN. Because this search routine is general, the program works with KW\$ of any length.

I call the generalized statements on lines 40 and 50 *recipes*. Most computer books call them *algorithms*. An algorithm is a plan or method of doing something in a program.

CHANGING A STRING

The routine now in memory finds a keyword string, but doesn't do anything except say where it is in the target string. Let's modify the program so it takes the target string apart, changes some characters, and puts it back together again.

The plan is to find the keyword using MID\$(). Then divide the target string into two parts using LEFT\$() and RIGHT\$(), omitting KW\$. KW\$ will not be included in either part.

Then replace KW\$ with some other set of characters and put it all back together again.

List the program. Then change it as shown, starting at line 30. Type line 90 very carefully. Then I will explain it very carefully.

```
10 PRINT CHR$(147)
20 T$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 KW$="JKL":L=0
40 COUNT=LEN(T$)-LEN(KW$)+1
50 FOR S=1 TO COUNT
60 IF MID$(T$,S,LEN(KW$))=KW$ THEN L=S
70 NEXT S
80 P1$=LEFT$(T$,L-1)
90 P3$=RIGHT$(T$,LEN(T$)-L-LEN(KW$)+1)
100 PRINT P1$,P3$
```

Line 30 establishes a value for KW\$ and initializes a variable L. At line 60, when KW\$ is found, the keyword begins at location S in the target string. Line 60 makes a note of that location by setting $L=S$. L stands for *Location*. The program remembers where KW\$ is located in T\$ by storing the value of S in memory as L.

The loop continues to run and the value of S changes, but that doesn't matter because the location of KW\$ is already in memory as L.

Line 80 takes the left part of T\$, up to the beginning of KW\$. In this target string, JKL begins at location 10, so L has the value 10. To omit KW\$, the left part should be the first nine characters. Line 80 takes $\text{LEFT}\$(T$,L-1)$, which evaluates to $\text{LEFT}\$(T$,9)$.

Line 90 takes the right part of T\$, also omitting KW\$. It's another recipe. The first time I attempted to write a line like that, I experienced a mental collapse. I decided to give up computing and return to my regular occupation—shooting pool and drinking refreshing liquids.

By writing the target string on a piece of paper, and using information that the program knows, I finally figured out how to chop off the correct number of characters from the right end of T\$. Here's how I did it:

I will use an imaginary pointer to show how line 90 works. Remember that the purpose is to take characters from the right end of the target string, but omit JKL. Line 90 says $P3\$=\text{RIGHT}\$(T$,LEN(T$)-L-LEN(KW$)+1)$. P3\$ should be all of the characters to the right of JKL in the target string.

In the $\text{RIGHT}\$()$ statement, the number of characters taken from the right end of T\$ is $LEN(T$)-L-LEN(KW$)+1$. The target string has 26 characters. KW\$ begins at location 10 in the target string, so $L=10$. $LEN(KW$)=3$. Here is the target string with a pointer at the end:

ABCDEFGHIJKLMNOPQRSTUVWXYZ



To begin, take $LEN(T$)-L$ characters. That is $26-10$, which is 16. Here is the target string with the pointer set at character number 16 from the right end:

ABCDEFGHIJKLMNOPQRSTUVWXYZ



That would take characters K through Z, which is too many. It includes part of the keyword string. Adjust by taking fewer characters. Try reducing the number of characters by subtracting the length of KW\$. $LEN(T$)-L-LEN(KW$)$ is $26-10-3$, which is 13.

Here is the target string with the pointer at that location:

ABCDEFGHIJKLMNOPQRSTUVWXYZ



That takes characters N-Z, which is too few. It should be everything to the right of JKL, which is M-Z. To fix that, take one more character. Try $LEN(T$)-L-LEN(KW$)+1$, which is 14 characters. Here is the target string with the pointer at that location:

ABCDEFGHIJKLMNOPQRSTUVWXYZ



That gets the right number of characters. When I said that I used this method to figure out how to write this routine the first time, I wasn't joking. It took about two hours. Then I drank some refreshing liquids.

After you have done it a few times, it gets easier. When I wrote line 90 for this book, I got it right in only two tries. Now that you understand it, you can probably figure out a simpler way to do it.

When you have a difficult programming problem to solve, don't insist on doing it at the keyboard. Figure it out using pencil and paper. Make drawings and tables. Use a hand calculator or count on your fingers and toes. Once I cut out pieces of paper, wrote on them, and shuffled them around on a table until I figured out a method of doing something that had baffled me. The programmer's best friends are paper and persistence.

When lines 80 and 90 have executed, T\$ has been divided into two parts, P1\$ and P3\$. Neither part includes KW\$.

Line 100 is a temporary line to test the program. If the program is typed correctly, run it. The result should be two strings:

```
ABCDEFGHIJ MNOPQRSTUVWXYZ
```

The left part has 9 characters, the right part has 14 characters. JKL is not in either part.

Now, let's replace JKL with another string. Enter

```
100 P2$="12345"
110 T$=P1$+P2$+P3$
120 PRINT T$
```

Run it. The string 12345 is substituted for the string JKL. For practice, change the program so it deletes JKL and does not replace it with anything. Change the program so it replaces JKL with your name.

Line 110 changes T\$ in memory. It is not what it was at line 20. If you want to keep the old target string in memory, use a different string variable name at line 110, such as NT\$—meaning *New T\$*.

In this program, the new T\$ can be longer than the original T\$—up to a length of 255 characters. If the program makes the new T\$ longer than 255 characters, an error results.

SETTING A FLAG

Please list the keyword-search program. It is always best to run a FOR-NEXT loop to a normal termination, because the loop specifications are then erased from memory. This routine does that by setting L equal to the value of the loop counter when the keyword is found and then allowing the loop to run to completion.

After that, the value of L is used to disassemble the target string. Setting L equal to S at line 60 is like making a note of something that you don't want to forget. Later, you use the information in the note.

Setting a variable to some value that will be used later in the program is called *setting a flag*. The variable L is a flag.

Flags carry information. In this example, the value of L is the location of KW\$ in T\$.

L is initially set to zero at line 30. If KW\$ is found, it is set to the value of S during that pass through the loop. If KW\$ is never found, the value of L remains zero.

If the value of L is still zero when the search is completed, that provides another item of information. It means that the keyword was not found in the target string.

If line 80 executes when L is zero, it causes a program error. To see the error, change KW\$ to anything that will not be found, such as JLL, and run it.

The problem at line 80 is that L - 1 equals -1 when L is zero. LEFT\$(T\$, -1) is invalid. Instead of executing line 80 and producing an error, the program should tell the user that the keyword was

not found and then do whatever is appropriate. To do that, enter

```
75 IF L=0 GOTO 200
130 END
200 PRINT"KEYWORD NOT FOUND"
```

List the program. Line 130 is to prevent line 200 from executing when KW\$ is found. In a real-world program, line 200 would offer the user some alternatives: search again for a different keyword, end the program or whatever is appropriate.

You should still have KW\$ set at line 30 to JLL or something that can't be found. Run the program again to verify that the lines just added are working correctly.

SPEEDING UP THE SEARCH

Search routines in BASIC can be slow if there is a lot of searching to do. Once the keyword has been found, continued operations of the loop are usually not needed and just take more time. To stop looping and bring the loop to a normal termination when the keyword has been found, put in a statement to set the loop counter to its maximum value. When the NEXT statement finds that the loop counter has reached its maximum value, it ends loop operation.

It is interesting to see how much time is saved by doing that. Begin by putting a timer into the program. Enter

```
35 T1=TI
72 T2=TI
73 PRINT"TIME ="T2-T1
```

List the program. Line 73 will display the elapsed time in jiffies.

You should still have a value of KW\$ at line 30 that can't be found. KW\$ should be three characters, such as JLL. Run the program a few times. It takes about 13 jiffies, which is 13/60 second. That is the maximum amount of time the search loop will take, with that T\$ and that KW\$, because it runs to completion without finding the keyword.

To see how many times the loop operated, enter

```
PRINT S
```

When the loop ends with S set to 25, it actually operated 24 times. A FOR-NEXT loop always ends with the counter set one number greater than the number of loop operations.

As you can see in line 40, the number of loop operations depends both on the length of T\$ and the length of KW\$. If KW\$ is longer, larger segments of T\$ are examined in each pass through the loop and the loop runs fewer times. If T\$ is longer, it takes more passes to search through it.

Change KW\$ in line 30 to XXXXXX and run it again. It takes about 12 jiffies and makes 21 passes through the loop—still without finding the keyword.

At line 30, change KW\$ back to JKL. Now, KW\$ will be found. Run the program again.

It doesn't matter if KW\$ is found or not. The execution time with a three-character keyword is the same either way. That's because the loop runs to completion whether the keyword is found or not.

Change the program so the loop stops running when KW\$ is found. List the program. It's becoming fairly complicated. Look it over to be sure you remember what each line does. Then change line 60 like this:

```
60 IF MID$(T$,S,LEN(KW$))=KW$ THEN L=S:
   S=COUNT
```

To make your line 60 look like my line 60, type the colon and then press the space bar until the cursor is on the next row.

Now, when KW\$ is found, L is set to S as before. Then, S is set to equal COUNT by the statement you just added to line 60.

After line 60 executes, the loop counter has been set to its maximum value as specified in line 50. Line 70 stops loop operation. Then, the rest of the program executes.

Run the program. It found JKL, deleted it and substituted 12345. The search took only 6 jiffies instead of 13. That's because the search operation stopped as soon as JKL was found. JKL is near the center of the target string, so search time was reduced to about half. Enter

PRINT S

It is still 25. S no longer indicates the number of loop operations because S is set to its maximum value at line 60, as soon as KW\$ is found. L is the actual number of loop operations. Enter

PRINT L

The loop ran 10 times and stopped. It found JKL at location 10 in the target string. At line 30, change KW\$ to ABC. Run the program. Notice that ABC is replaced by 12345. The search operation took 1 jiffy and the loop ran 1 time.

Change KW\$ to Z and run the program. It took 15 jiffies and 25 searches.

This is the most complicated program shown so far. You may want to spend a few minutes analyzing it and experimenting with it. This book won't use it again, so change it any way you wish.

SUMMARY OF WAYS TO MAKE A STRING	
X\$="12ABC34"	Typed <i>directly</i> into a program line. Defined by quotation marks.
X\$="ABCDE"	X\$ defined by typing it inside quotation marks.
Y\$=X\$	Y\$ defined <i>indirectly</i> by setting it equal to X\$.
INPUT X\$	Accepts keyboard input as X\$. Keyboard input is not typed inside quotation marks. Will not accept commas. Press RETURN to enter data.
GET X\$	Accepts single keystroke as X\$. Accepts commas. Typing inside quotation marks not necessary. Pressing RETURN to enter data not necessary.
READ X\$	Reads data item from data list and assigns it to string variable name. Item in data list is not typed inside quotation marks.
Y\$=LEFT\$(X\$,3)	Forms new Y\$ using leftmost three characters of X\$.
Y\$=RIGHT\$(X\$,5)	Forms new Y\$ using rightmost five characters of X\$.
Y\$=MID\$(X\$,5,8)	Forms new Y\$ using characters 5 through 12 of X\$.

HOW TO SHORTEN A STRING

This is another recipe. As you will see in the next section, it is sometimes necessary to shorten a string by one or more characters. You may not know how long the string is, so you need a generalized program line to do that. This will do it. Enter

NEW

10 INPUT "ENTER A PHRASE";P\$

20 PRINT P\$

30 P\$=LEFT\$(P\$,LEN(P\$)-1)

40 PRINT P\$

The automatic string-shortener recipe is line 30. In the LEFT\$() statement, the number of characters to be taken is LEN(P\$) - 1. If P\$ is entered with 10 characters, line 30 takes the leftmost 10 - 1 characters, which is 9. The string is shortened by one character. Run the program and enter NOW IS THE TIME.

COPING WITH COMMAS

When you use an INPUT statement to receive data from the keyboard, the computer will not accept commas. It accepts whatever was typed ahead of the comma, displays an error message saying ?EXTRA IGNORED, and continues running the program. Enter

```
NEW
10 INPUT A
20 PRINT A
```

Run it and enter 12,345. Change A to A\$ everywhere in that long program and run it again. Enter RUN JACK, RUN!

Some people cannot overcome the habit of entering commas where they think commas belong. If that happens, and the user ignores ?EXTRA IGNORED, the program will proceed with 12 instead of 12,345 or RUN JACK instead of the complete phrase.

People who ignore ?EXTRA IGNORED usually think the program made a mistake when it produces incorrect results. It is common to put warnings on the screen saying DON'T ENTER ANY COMMAS. But people tend to follow habits better than warnings.

You may decide to allow users of your programs to enter commas, if they insist, without truncating the input at the comma. *Truncating* means cutting off. That can be done by using GET statements to take keyboard inputs one character at a time. GET statements will accept commas. Enter

```
NEW
10 GET A$:IF A$=" "GOTO 10
20 PRINT A$
```

Run it and enter a comma. GET likes commas. To accept commas from the keyboard, put a GET routine in a loop that runs until the user presses RETURN. Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT"TYPE SOMETHING WITH COMMAS"
   CHR$(13)"THEN PRESS RETURN":PRINT
30 S$=" "
40 GET K$:IF K$=" "GOTO 40
50 IF K$=CHR$(13) GOTO 110
60 IF K$=CHR$(20) AND LEN(S$) > 0 THEN
   S$=LEFT$(S$,LEN(S$)-1):GOTO 80
70 S$=S$+K$
80 PRINT CHR$(19):PRINT:PRINT:PRINT
90 PRINT S$" "
100 GOTO 40
110 PRINT:PRINT:PRINT S$
120 PRINT LEN(S$)
```

Line 30 initializes the variable S\$ to a null. S stands for *Sentence*. This program will use S\$ to hold all keystrokes entered before pressing RETURN.

Line 40 gets keystrokes, one at a time, and names each one K\$. KS stands for *Keystroke*.

Line 50 tests each keystroke to see if it is the RETURN key. If so, the user has finished entering data and the program jumps out of the loop to line 110.

I will discuss line 60 in a minute. Line 70 concatenates S\$ and K\$. S\$ starts as a null. As each new K\$ is received by line 40, it is stuck onto the end of S\$. S\$ will become longer and longer as keystrokes are made.

Line 80 locates the cursor, getting ready to print S\$. CHR\$(19) homes the cursor without erasing the screen. The following PRINT statements move the cursor down.

Line 90 prints S\$ followed by a space. Each new S\$ is overprinted on top of the preceding S\$. You can see S\$ growing longer on the screen. Line 100 jumps back to line 40 to get the next keystroke.

When another keystroke has been received by line 40, line 50 tests again to see if it is CHR\$(13). If not, line 60 may execute. When the user of this routine makes a typing error, he will normally press the INST DEL key to back up and fix the mistake. CHR\$(20) is produced when the delete key is pressed, unshifted. Line 60 tests each keystroke to see if it is the delete key. If so, S\$ should be shortened by one character because that's what the user expects the delete key to do.

However, S\$ should not be shortened if it has no characters. Therefore line 60 tests first for the delete key and then tests the length of S\$. If delete was pressed *and* S\$ is longer than zero, line 60 executes. It shortens S\$ by one character using the recipe given earlier. Then it jumps ahead to line 80 to display the shortened S\$.

The reason line 60 jumps to line 80 is to avoid line 70. That line concatenates K\$ onto the end of S\$. If K\$ is a delete keystroke, it should not be stuck on the end of S\$.

When S\$ is printed by line 90, it also prints a space at the end of S\$. That is needed when the user is deleting. Suppose the user had typed NOW IS THE TIMA. Those characters will be on the screen. Then the user presses the delete key to delete the incorrect A in TIMA. The automatic string shortener at line 60 changes S\$ to NOW IS THE TIM by removing the last character.

If NOW IS THE TIM is overprinted on top of NOW IS THE TIMA, the screen will still display NOW IS THE TIMA. The user will think the delete key didn't work.

But, if NOW IS THE TIM *followed by a space* is overprinted on NOW IS THE TIMA, the space erases the final A and the screen shows the deletion.

The reasons for lines 110 and 120 are to show what is in memory as S\$ and to show how long it is.

I think this routine is interesting to run, when you know what it does. Run it and enter some strings with commas. Delete characters occasionally to see how that works.

Start with a short phrase and mentally keep track of the characters and deletes. Then press RETURN and compare what is in memory to what you think should be in memory. If you have a mental collapse, use pencil and paper to keep a record of keystrokes.

When you have done that, enter a string that is longer than 255 keystrokes. After you see the error message, check to see what is held in memory as S\$. Check its length.

You will notice that the keyboard action seems to slow down as you enter longer strings. That's because each keystroke requires data processing.

This routine demonstrates several useful ideas. One is a way to allow commas to be input from the keyboard. More important is using a GET routine to process keystrokes one at a time. This allows testing each keystroke and a lot of other tricks. Also, you got another recipe.

SUBROUTINES

A GOTO statement sends the program to the specified line number. Then the program follows normal program flow from that line in the program.

Sometimes, you want to execute a routine and then return to the original location in the program. For example, you may want to leave line 520, execute a routine somewhere else in the program, and then return to the main program and continue from where you left it.

This allows using a routine repeatedly, perhaps at different locations in the program, without typing it repeatedly. A routine used more than once, by this method, is called a *subroutine*.

The BASIC word GOSUB sends the program to a specified line number to execute a subroutine. For example, a line that says 520 GOSUB 1000 causes a jump from line 520 to line 1000. We say that the GOSUB statement on line 520 *calls the subroutine* at line 1000.

At the end of the subroutine starting at line 1000, perhaps at line 1080, you must place the BASIC word RETURN. The RETURN statement sends the program back to the statement

immediately following the GOSUB statement that called the subroutine.

If the only statement on line 520 is GOSUB 1000, then the following statement is on the next program line, possibly line 530.

If there are two statements on line 520, such as 520 GOSUB 1000: PRINT X, then the subroutine will return to the statement PRINT X and execute it. Then, the program continues its normal flow from line 520.

The following section demonstrates using a subroutine.

HOW TO BUILD A COMMA EATER

When you have made a program accept commas without scolding, it will seem friendly to users. If the user enters sentences or maybe poetry, the commas should remain in the strings.

If the user enters numbers that will be used in calculations, you must *remove the commas* before converting the strings to numbers.

VAL() takes the numeric value of a string up to the first character that isn't a number. It will stop at the first comma. If the string is 12,345, VAL() gets the value 12.

To solve that problem, run the string through a keyword-search routine similar to the one used earlier in this chapter. Find all commas and replace them with nothing. Then convert the string to a numeric and do the calculations.

Doing that is a little simpler than a generalized keyword-search routine. You know that the length of the keyword is 1 character, because you are looking for commas. The routine searches all the way through the string to remove each comma. Nothing is substituted for commas.

This program eats commas using a hungry subroutine. Enter

```
NEW
500 S$="12,345"
510 FOR K=1 TO LEN(S$)
520 IF MID$(S$,K,1)="," THEN GOSUB 1000
530 NEXT K
540 PRINT S$, VAL(S$)
550 END
990 REM SUBR TO EAT COMMAS
1000 P1$=LEFT$(S$,K-1)
1010 P3$=RIGHT$(S$,LEN(S$)-K)
1020 S$=P1$+P3$
1030 K=K-1:RETURN
```

Line 500 puts a value for S\$ into memory so the rest of the program will have something to chew on. Line 520 looks for commas. If it finds one, it calls subroutine 1000.

Line 1000 takes the left part of S\$, without the comma. Line 1010 takes the right part, without the comma. Both of these lines use K, the value of the loop counter when a comma is found. Both are simpler than a general-purpose keyword search because the length of the keyword is 1.

Line 1020 puts S\$ back together without the comma. Line 1030 reduces the value of the loop counter by 1 and then returns to the statement immediately following the GOSUB statement that called this subroutine.

The loop counter is reduced by 1 in case somebody enters two or more commas in sequence. Reducing a number by one is called *decrementing* the number. I don't know why anybody would enter two or more commas in sequence but, if you don't protect against it, somebody surely will.

Suppose there are commas at locations 3 and 4 in S\$ and the number 7 at location 5. The program finds the comma at location 3, removes it and closes up S\$.

Closing up S\$ moves the comma that used to be at location 4 into location 3. If the program then searches location 4, it finds the number 7. It would never see the comma that sneaked into location 3 and it would leave it in S\$.

Obviously, if the search routine backed up and searched location 3 again, it would find the second comma. That's why line 1030 decrements K each time a comma is found.

In this routine, the GOSUB statement is the last statement on line 520. From line 1030, the subroutine returns to line 530, and the routine searches S\$ for more commas.

Run the program. It displays 12345 for S\$ and 12345 for the value of S\$.

At line 500, change S\$ to ,,,1,,,2,3,4,5,,,. If this routine can cope with that, it can probably snack on commas all day long. Run it. Try as many different values for S\$ as you wish.

Then list the program. Now that you have seen the subroutine work, think about how the program could be written without using a subroutine.

The subroutine should execute only when line 520 finds a comma. Obviously, this program would work the same way if all of the comma-eating statements in the subroutine were part of line 520—behind the IF statement.

When possible, that's the best way to write routines like this. In this example, the statements used in the subroutine won't fit on line 520.

If you write them on other lines inside the loop, the program must do a lot of jumping around. One way is to set a flag at line 520 if a comma is found. Then, following lines test the flag to determine whether or not to remove commas.

The subroutine makes this program simpler than trying to do it all in the loop.

GOTO AND GOSUB WITH IF-THEN STATEMENTS

Here is an important difference between GOTO and GOSUB. An IF-THEN statement that *goes to* another line can be written three ways, all with the same result:

IF A=B THEN GOTO 330

IF A=B THEN 330

IF A=B GOTO 330

With a GOSUB statement, you don't have those options. You must use both THEN and GOSUB. Line 520 is an example.

USING ASC()

This discussion gives you more information about ASC() and also shows some programming techniques that you should know. ASC(X\$) produces the ASCII code number for the first character in X\$. If X\$ is ZEBRA, then ASC(X\$) is 90 because the ASCII code for Z is 90.

CHR\$(n) represents the keystroke that has ASCII code n. The statement PRINT CHR\$(90) will display the letter Z.

When writing programs, you may want to use ASC() to get the ASCII code number of a keystroke.

There is one use of ASC() that won't work. Here are some facts that will help explain that: CHR\$(0) is a null—it is nothing. In a program, a null is represented by "", which is a string with nothing in it. The statements PRINT "" and PRINT CHR\$(0) mean the same thing. They both print a null. ASC("") *should be zero* because the ASCII code for a null is zero.

The computer can cope with all of those facts, except one. Enter

```
PRINT ""
```

```
PRINT CHR$(0)
```

You are looking at two actual nulls, right there on the screen. Enter

```
X$="A":PRINT X$,:PRINT ASC(X$)
```

That works fine, it prints X\$ and its ASCII code number. Enter

```
X$="":PRINT X$,:PRINT ASC(X$)
```

That doesn't work. The part that doesn't work is ASC(X\$) when X\$ is a null. Instead of printing

zero, which is the ASCII code for a null, it displays the error message ?ILLEGAL QUANTITY ERROR. To be sure, enter

```
PRINT ASC(" ")
```

Another way to represent a null is CHR\$(0). Try entering this:

```
PRINT ASC(CHR$(0))
```

That works. There is no special reason that ASC(X\$) produces an error message when X\$ is a null and ASC(" ") produces the same error message. But ASC(CHR\$(0)) works. It's just the way the computer is programmed. If you don't know about it, you can get into trouble that is hard to diagnose.

As an example, here is a program that accepts keystrokes and takes the ASCII code of each keystroke using ASC(). Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT "PRESS A CHARACTER KEY,"
30 PRINT "OR PRESS Q TO QUIT."
40 PRINT:PRINT "THEN PRESS RETURN."
50 PRINT:INPUT KS$
60 N=ASC(KS$)
70 PRINT
80 PRINT "THE ASCII CODE FOR "KS$" IS "N
90 FOR DL=1 TO 2000:NEXT
100 IF KS$<>"Q" THEN CLR:GOTO 10
110 PRINT:PRINT "END OF PROGRAM"
```

Line 60 takes the ASCII code of the keystroke KS\$ and names it N. Line 80 displays the keystroke and its ASCII code. Line 90 provides a delay to read the display. Line 100 tests the keystroke to see if it was Q. If not, line 100 executes and the program runs again. Otherwise, it ends.

Actually, line 100 tests KS\$ to see if it *is not* Q. Often a program is easier to write if tests are made using <> rather than =.

That line does two things when it executes. Both are controlled by the IF statement. The first is CLR. That clears memory so the program can run again with no possibility of values being in memory from earlier runs.

If line 100 executes at all, it executes completely. The second statement on that line jumps back to the beginning of the program. Especially in lines that begin with an IF-THEN statement, you will find it very useful to do several things on one line.

The order of events in this program is significant. If the keystroke is Q, the program first displays the ASCII code for Q at line 80. Then it tests the keystroke to see if the program should continue or end. If the test were made first, the program would end without displaying the ASCII code for Q.

Run it and enter various keystrokes. When you have seen enough codes, press RETURN without making a keystroke first. That enters a null as the value of KS\$. Line 60 attempts to get ASC(KS\$), but KS\$ is a null, and an error results.

Here is a way to prevent that error. List the program. Change line 60 to read

```
60 N=ASC(KS$+CHR$(0))
```

Run the program again and press RETURN without pressing any other key first. Now it works. When RETURN is pressed without any other keystroke, a null is entered. The program displays a null and then reports that its ASCII code is zero. Try entering characters. That works also.

List the program. Line 50 accepts a keystroke and names it KS\$. If no keystroke is made before pressing RETURN, KS\$ is a null. At line 60, the keystroke KS\$ is concatenated with CHR\$(0), which is a null. If KS\$ is a null, the result seems to be null+null. However, these two nulls are represented in two different ways. The first null is "". The other is CHR\$(0).

ASC() does not recognize "", but it does recognize CHR\$(0). Therefore, this peculiar null+null actually looks like just one null to the computer.

Suppose KS\$ is not a null. Perhaps it is A. Then the concatenated string produced by line 60 is A+null. ASC() takes the ASCII code of the first character in a string with more than one character. It will ignore the CHR\$(0).

With any computer, the programmer must do some things just to adapt the program to the computer. With another brand, you may not have to make that adaptation, but you usually have to adapt to the computer in other ways.

REVIEW

It may seem that writing a program to accept commas and then take some of them back out is a lot of trouble for a small reward. Not so. If anybody else will use your programs, especially people who are not programmers, it can make the difference between a program that is enjoyed and one that is rejected.

When you have done it a few times, you will probably decide to write all programs that way, even those that you don't expect anybody else to use.

It may seem that I am spending a lot of your time messing around with screen displays, strings, ASCII codes and the associated programming techniques. Because computers are for calculating, you may be anxious to get on with the arithmetic.

Calculations are the easy part. Most good programs have a few lines that do calculations and many lines that display instructions, accept user inputs, check for errors and display the results of calculations. What has been discussed so far is the hard part. If you can't do these things, you can't write a successful program—no matter how good you are at calculations.

I suggest that you review this chapter. The demonstration routines and algorithms are practical and can be used without much change in a large variety of programs. Programming is partly knowing BASIC words and partly knowing routines.

Spend a little time reviewing the routines in this chapter, so you add them to your arsenal of routines. The new BASIC words in this chapter are listed in the accompanying table. Read about them in Appendix B.

In case this is good news, calculations are in the next chapter.

BASIC WORDS IN THIS CHAPTER	
LEFT\$()	STR\$()
LEN\$()	RIGHT\$()
MID\$()	TI
	TIS

A PROGRAM THAT DISPLAYS TIME USING A 12-HOUR CLOCK

```
5 REM THIS PROGRAM DISPLAYS THE TIME
6 REM SET TIMER BEFORE RUNNING PROGRAM
10 PRINT CHR$(147)
20 HOUR=VAL(LEFT$(TI$,2))
30 AP$="AM":IF HOUR>11 THEN AP$="PM"
40 IF HOUR>12 THEN HOUR=HOUR-12
50 HOUR$=STR$(HOUR)
60 HOUR$=RIGHT$(HOUR$,2):REM 2 DIGITS
70 MIN$=MID$(TI$,3,2)
80 SEC$=RIGHT$(TI$,2)
90 PRINT"THE TIME IS "HOUR$"."MIN$" "AP$
  " AND "SEC$" SECONDS"
100 PRINT CHR$(19):GOTO 20
```

Note: At line 50, HOUR may be 3 digits, such as 12 with a leading space. When converted to a string by STR\$, HOUR\$ will be 3 characters including the leading space. Line 60 omits the leading space, if there are 3 characters.

This causes the time message to have the same number of characters at any hour, so one time message can be overprinted on another.

11

Mathematical Operations

Your computer can do as much math as you can tell it to. It has symbols for everything we do in arithmetic—multiply, divide and so forth. It also has symbols and the capability for virtually everything we do in algebra, including nested parentheses.

It understands logarithms and trigonometry. In higher fields of mathematics, such as calculus, it can provide answers by *number crunching*—repeated operations with numbers. It also does *logical* operations, sometimes called *Boolean algebra*.

The difficulty in discussing these operations is that some will be appreciated only by people who understand the relevant math. If I say that the computer can take the natural logarithm of a number, that's good news to log fans but no news at all to people who think logs grow on trees.

The easy way out is to describe all of the mathematical words and symbols but demonstrate only the more common operations. Without hesitation, I will take the easy way.

ARITHMETIC OPERATORS

The first thing is to learn the symbols and keystrokes used to produce them. You know most of them already.

- = Equals
- + Add
- Subtract
- * Multiply
- / Divide
- ↑ Exponentiation (raising to a power)
- () Evaluate the expression inside the parentheses

Here are some examples. Please clear memory. Then enter them and observe the results.

```
X=2: PRINT X
X=2+3: PRINT X
X=5-3: PRINT X
X=5*3: PRINT X
X=15/4: PRINT X
X=5↑2: PRINT X
X=2*(2+2): PRINT X
```

Mathematical Operations

Operations in parentheses are completed first. Then other operations are performed.

With a minus sign, you can do *negation*, which means to change the sign of a number. In the expression $Y = 2 * (-X)$, negation is performed inside the parentheses.

ORDER OF PRECEDENCE

In a complicated mathematical expression, several operations may be indicated by symbols. The result may depend on the order in which the operations are performed. The BASIC interpreter will perform operations in this order:

- () Evaluate parenthetical expressions
- ↑ Exponentiation
- Negation
- * or / Multiply or Divide in left-to-right order
- + or — Add or Subtract in left-to-right order

When you write a program line to do a calculation, test it with numbers. If the answer is wrong and you don't see a reason, check the order of precedence. You may expect the computer to do the calculation one way, but it actually does it another way.

For example, the expression $3 + 2 * 2$ would be 10 if addition were performed first. $3 + 2$ is 5. $5 * 2$ is 10. Enter

```
PRINT 3+2*2
```

By following the order of precedence, the computer evaluates that expression as 7. $2 * 2$ is 4. $3 + 4$ is 7.

USING PARENTHESES

By using parentheses, you can control how an expression is evaluated. The computer will always evaluate expressions in parentheses first. Then it will perform the other indicated operations. Enter

```
PRINT (3+2)*2
```

The parentheses cause the computer to add $3 + 2$ first, then multiply the result by 2. The result is 10.

If more than one operation is to be performed inside the parentheses, the order of precedence will be used within the parentheses.

Nested Parentheses—When you use parentheses to control how an expression is evaluated, you may end up with one set of parentheses inside another set. This is called *nested parentheses*. You can use up to 10 nested parentheses. Enter

```
X=2+(2*(2+(2*3)))
```

```
PRINT X
```

That expression has three nested parentheses. The computer will begin with the expression in the innermost parentheses and work outward from there. In evaluating that expression, the innermost parenthetical expression is $(2 * 3)$.

Working outward: $2 * 3$ is 6. $2 + 6$ is 8. $2 * 8$ is 16. $2 + 16$ is 18.

Parentheses Must Be Used in Pairs—Every left parenthesis symbol must have a matching right parenthesis symbol. When you are using nested parentheses in complicated expressions, it's easy to leave one out. The result is an error message that says **?SYNTAX ERROR**.

Count the number of left parenthesis symbols and the number of right parenthesis symbols. If they are not the same, you found the syntax error.

NOTATION

The BASIC word to be discussed next is normally written ABS. In this book, BASIC words that require a number or variable name to be typed in parentheses after the word are written with following parentheses, such as $ABS()$. The parentheses are a reminder that a number or variable name must be used in parentheses with that BASIC word.

ABS()

This means ABSolute value, which is the positive value of the expression in parentheses, even if the expression is negative. Enter

```
X = -4
Y = ABS(X)
PRINT Y
```

INT()

This means INTeger. It produces the integer value of the expression in parentheses. If X is the integer value of Y, then X is the largest whole number that is not larger than Y. Enter

```
PRINT INT(8.1)
PRINT INT(8.999)
PRINT INT(8)
PRINT INT(7.999)
PRINT INT(0)
PRINT INT(-4.1)
PRINT INT(-3.9)
```

For positive numbers with decimal fractions, the integer value is obtained just by discarding the fraction. For negative numbers with decimal fractions, the integer value is the next negative whole number. For example, -4 is the integer value of -3.9.

When a number is defined as an integer by using the integer type-designation symbol, %, those rules apply. Enter

```
X = 12.345
PRINT INT(X)
Y = INT(X)
PRINT Y
Y% = X
PRINT Y%
```

All of those operations have the same result, but there are some differences. PRINT INT(X) displays the integer value of X but it does not establish a variable name with that value. In memory, X remains 12.345.

Y = INT(X) establishes a new variable name in memory. It is Y. Its value is INT(X), which is 12. But, Y is not an integer variable name. It doesn't have to be an integer. The expression Y = INT(X) assigned the integer value of X to Y because the programmer wanted to do that.

If you use the statement Y% = X, the variable Y% can accept only the integer part of X because Y% is an integer variable name. Y% = X has the same result as Y% = INT(X). Y% can never have a decimal fraction. Enter

```
PRINT Y%
Y% = 2.3 * Y%
PRINT Y%
```

CONVERTING UNITS OF MEASURE

Here is a useful routine to convert ounces to pounds. Similar programs can be used to make metric conversions, convert money from one set of units to another, and similar purposes. Enter

```
10 PRINT CHR$(147)
20 INPUT "ENTER NUMBER OF OUNCES.";OZ
```

(Program continued on next page.)

```
30 LB%=OZ/16
40 USED=LB%*16
50 LEFT=OZ-USED:PRINT
60 PRINT"THAT IS "LB%"LB, "LEFT" OZ"
```

Line 30 divides ounces by 16 and assigns the result to the integer variable name LB%. Suppose the number of ounces is 20. Dividing OZ/16 yields 1.25. LB% takes the value 1, which is the number of *whole pounds* in 20 ounces. The decimal fraction, .25, is discarded and *disappears*.

After line 30 executes, the program knows OZ and LB%—the number of ounces and the number of whole pounds.

Line 40 calculates the number of ounces “used” to make whole pounds. It multiplies LB% by 16 and assigns the result to the name USED. Now, the program knows how many OZ it started with and how many were USED to make full pounds.

Line 50 calculates how many ounces are left over. It subtracts USED from OZ to find how many are LEFT. Line 60 prints the number of pounds and ounces represented by the number of ounces that were input at the beginning of the program.

Run it and input the number 20. Run it again and input the number zero. Try other numbers of ounces, such as 16, 32 and 100.

SQR()

This takes the square root of the expression in parentheses. It may have a decimal-number result. Enter

```
PRINT SQR(4)
PRINT SQR(15)
PRINT SQR(-9)
```

SQR() doesn't work with negative numbers.

EXPONENTIATION

This is raising a number to a power. The power is called an *exponent*. The symbol is ↑. Enter

```
PRINT 4↑2
```

The number 4, raised to the second power, is 4 squared or 4*4. Enter

```
PRINT 2↑3
```

That is equivalent to 2*2*2, which is 8. Enter

```
PRINT -4↑2
```

The result is -16. If you intend the expression -4↑2 to mean (-4)*(-4), the answer is not -16. You may have been taught in school that “a minus times a minus is a plus.” By that rule, (-4)*(-4) is 16. Let's see what rule the computer uses. Enter

```
PRINT (-4)*(-4)
```

The result is 16. The two answers are different because of the order of precedence. The expression -4↑2 is executed like this: -(4↑2). The number is raised to the indicated power, then the minus sign is applied to the result. In the expression (-4)*(-4), there is no exponentiation. Two negative numbers are multiplied.

SGN()

This tests the sign of an expression in parentheses. If the variable X is positive, the value of SGN(X) is 1. If X is zero, SGN(X) is zero. If X is negative, SGN(X) is -1.

SGN() is used to make decisions based on the sign of a variable. In an accounting program, you might do one thing if a balance is positive, a different thing if the balance is negative, and neither of the above if the balance is zero.

FLOATING-POINT NOTATION

This is the method used in the Commodore 64 to display numbers with decimal fractions.

Ordinary Numbers—For large numbers up to 999999999, nine digits are displayed and the decimal-point “floats” back and forth to show the value of the number. To see that, enter

```
NEW
10 INPUT X
20 PRINT " "X
30 GOTO 10
```

Run it and enter

```
123456789
1.23456789
12345.6789
```

No matter where you place the decimal point, nine digits are displayed. Zeros entered before the first significant digit, or after the last significant digit are ignored. Enter

```
000123.456000
```

If you enter more than nine digits, but the value of the number is less than 999999999, it is rounded off to nine digits and displayed as an ordinary decimal number. Enter

```
123.123456789
```

The ninth digit in the number that was entered is 6. The ninth digit in the rounded-off number displayed is 7.

Rounding off begins by dropping the last digit and follows this rule: If the digit being dropped is 5 or larger, the number to its left is increased by one. If the digit being dropped is 4 or less, the number to its left is not changed. This is called the *5/4 roundoff rule*.

Apply that rule to the number that was input, while mentally rounding it off to nine digits. You will see that the number 6 should be changed to 7.

I refer to the numbers on the screen as *ordinary* decimal numbers because there is nothing special about the way they are displayed. The largest number that can be displayed that way is 999999999. The smallest number that will be displayed as an ordinary decimal number is .01.

Scientific Notation—Decimal numbers larger than 999999999 or smaller than .01 are displayed using *scientific notation*. You may have learned this in school as *powers of ten*. Pocket calculators use this method to display very large and very small numbers.

When scientific notation is used to display a number, it will have three parts, such as 1.23456789E+09

The number to the left of the letter E is called the *mantissa*. The mantissa is rounded off to 9 digits. The decimal point in the mantissa is always placed so the value of the mantissa, considered alone, is 1 or larger but less than 10.

The letter E stands for *Exponent*. The number following the letter E is called the *exponent*.

The exponent tells how many places to move the decimal point in the mantissa to restore its original value, and in which direction. In this example, the exponent is +09. That means, move the decimal point 9 places to the right.

Enter this number. It has 10 digits to the left of the decimal point:

```
1234512345.
```

The computer displays 1.23451235E+09. The mantissa is 1.23451235, which is the original number rounded off to 9 digits with the decimal point relocated. The exponent is +09.

To restore the value of that number, the decimal point in the mantissa must be moved 9 places to the right. To do that, you must add a zero at the end of the number so there will be nine digits after the decimal point. Adding a zero at the end does not change the value of the mantissa.

Now it is 1.234512350 and the decimal point can be moved. Move it 9 places to the right. The result is 1234512350.

The number now has 10 digits to the left of the decimal point, as it did when you entered it. But, it is not exactly the same number, because it was rounded off to nine digits when expressed in scientific notation.

You put in 1234512345 and got back 1234512350. The difference is a tiny percent of the original number, but it is not the same. Sometimes it is important to know that.

Although the computer rounds off to nine digits and displays nine digits, it actually uses 10 digits internally in calculations.

To see what happens with small numbers, enter

.01

That decimal fraction is displayed in the ordinary way. Enter

.001

That causes the computer to use scientific notation. It displays 1E-03. The minus sign preceding the exponent means move the decimal point three places to the left.

The mantissa is 1. Add two zeros to the left and it is 001. Then move the decimal point three places to the left, and it becomes .001, which is the number entered originally.

The program you are running is unusual. It's an endless loop, but you probably can't break out of it by pressing RUN STOP. That's because it spends most of the time at line 10 waiting for an input. The INPUT statement doesn't recognize RUN STOP. You can get out of it by pressing RUN STOP-RESTORE.

ROUND OFF ERRORS

Roundoff errors caused by scientific notation are usually tolerable because they are such a small percentage of the original number.

In some cases, they may not be tolerable. In accounting programs, people expect numbers to be accurate to the penny. It takes a huge amount of money to force the computer into scientific notation. If it should happen, you can solve the problem with some fancy programming.

When doing calculations, the computer will round off when necessary. This may cause small roundoff errors several places to the right of the decimal point.

A ROUND OFF ALGORITHM

You will sometimes want to round off numbers in a program, so the computer doesn't display a lot of digits to the right of the decimal point. This is a 5/4 roundoff algorithm.

It adds 0.5 to a number and then takes the integer of the result. Suppose the number is greater than 8.5 but less than 9. By the 5/4 roundoff rule, it should round off to 9. Adding 0.5 will cause it to be greater than 9. The INT() operation will convert it to 9.

Suppose the number is 8 or larger but smaller than 8.5. It should round off to 8. Adding 0.5 will make it larger than 8.5 but smaller than 9. The INT() operation will convert it to 8. Enter

NEW

10 PRINT CHR\$(147)

20 INPUT "ENTER A DECIMAL NUMBER";N

30 N=INT(N+.5)

40 PRINT N

50 GOTO 20

Run it. Enter numbers such as 8.5 and 22.66 and watch it round them off. Enter some negative numbers. Break out of the program.

ROUNDING OFF TO TWO OR MORE DECIMAL PLACES

This is a more general routine that will round off to any desired number of decimal places.

Suppose you are dealing with money and want to round off to two decimal places. The idea is the same but you must first multiply the number being rounded off by 100. That changes the number so everything to the left of the decimal point is pennies instead of dollars. If there are still some numbers to the right of the decimal point, they should be rounded off using the 5/4 rule.

When that has been done, you have a whole number of pennies. Divide it by 100 to convert it back to dollars and cents. Change line 30 to read

```
30 N=INT((N*100)+.5)/100
```

Notice that you multiply by 100 first, then add .5, then divide by 100. Run it and enter some numbers with more than two decimal places. It rounds off to two places. If you don't see how it works, do it with pencil and paper.

To round off to one place, use 10 as the multiplier. To round off to three places, use 1000, and so forth.

ALLOWABLE RANGES OF NUMBERS

The allowable range of integers is -32768 to $+32767$. If you enter a number outside of this range, an error message results. If the computer *calculates* a number out of this range, while a program is running, it stops and displays an error message. To meet a new error message, enter and run

```
NEW
```

```
10 X%=30000
```

```
20 Y%=2*X%
```

Decimal numbers can have a maximum value of $1.70141183E+38$, which is a huge number. If this number is exceeded, the computer stops and displays an error message that says ?OVERFLOW ERROR.

The smallest number that the computer recognizes is $2.93873588E-39$. If a calculation produces a smaller value, the computer does not stop. It uses zero for the smaller value and continues running the program.

DIVIDING BY ZERO

Division by zero is usually not allowed in mathematics because it yields no sensible result. If a program line attempts to divide by zero, the program stops and displays an error message. Enter

```
X=4/0
```

Usually, division-by-zero errors are not that obvious, but the error message is obvious, and it tells you where to look.

RND()

Random numbers are selected at random. If a number is truly random, there is no way to predict what it will be.

The Commodore 64 has a built-in number generator that produces a series of decimal numbers between 0 and 1 that appear to be random. They are larger than 0 and smaller than 1.

The series is produced inside the computer by a numerical procedure that begins with a single number called a *seed*. Each time the computer is turned on, it begins with the same seed number and produces the same series of numbers that appear to be random. Because this series of numbers repeats, the numbers are not truly random. They are *pseudo-random*, but are usually called *random*.

The BASIC word RND() gets the next number in the series of pseudo-random numbers. RND() requires a number in the parentheses, but the *value* of the number doesn't affect the result. The *sign* of the number does. Because the value of the number doesn't matter, it's called a *dummy number*.

If the number is positive, such as RND(1), the next number is taken from the series of numbers that is available from the number generator.

Mathematical Operations

Turn off the computer and then turn it back on. Enter

```
FOR I=1 TO 3:PRINT RND(1):NEXT
```

Write down the three numbers. Turn the computer off and back on again. Enter

```
FOR I=1 TO 3:PRINT RND(7):NEXT
```

The same series of numbers is repeated. From power on, RND(7) produces the same series as RND(1) or any other positive number in the parentheses.

If the number in RND() is negative, such as RND(-4), the number generator is given a different seed number and produces a different series of numbers. RND(-4) reseeds the generator and gets one random number from that series of numbers.

RND(-4) will always produce the same series of numbers because it specifies the same seed number as a starting point. RND(-5) will produce a different series, but the different series will always repeat when the generator is reseeded by RND(-5).

After the generator is reseeded by a negative number, using a positive number with RND() will get a number from the series that was started by reseeding. It will not change the seed. Without turning off the computer, enter

```
PRINT RND(-1)
```

That reseeds the generator and gets the first number in a different series. Enter

```
PRINT RND(-1)
```

That reseeds the generator and starts over again. It gets the first number in the same series as before. Enter

```
PRINT RND(3)
```

That gets the next number in the series that was just started by RND(-1). Enter

```
PRINT RND(-7)
```

That starts a new series. Enter

```
FOR I=1 TO 3:PRINT RND(1):NEXT
```

That gets the next three numbers in the series that was started by RND(-7).

Using RND(0) causes the computer to derive numbers from the jiffy timer. It is possible for this series to repeat, but not likely. Enter

```
FOR I=1 TO 3:PRINT RND(0):NEXT
```

Controlling the Range of Random Numbers—As generated by the computer, random numbers are between 0 and 1. To get larger numbers, multiply each by a factor. Multiplying by 10 produces numbers between 0 and 10, which means greater than 0 but less than 10.

If you multiply by 10 and then take the integer value, the result is integers with values from 0 to 9. Enter

```
NEW
```

```
10 N%=10*RND(1)
```

```
20 PRINT N%
```

```
30 FOR DL=1 TO 300:NEXT
```

```
40 GOTO 10
```

Run that for a while and then break out of the endless loop. To increase the range of numbers produced by this routine, change the factor in line 10. If it is 100, the range becomes 0 to 99, and so forth.

Uses of Random Numbers—Random numbers are used in games as a way of introducing some

unpredictable action for the players. Some programs use random numbers in screen displays to create random motions of objects.

Random numbers are also used in programs to simulate something that is random in nature. There are programs that simulate crap games by selecting random numbers to represent throws of the dice.

How well a crap-game simulation works depends on two things. Are the numbers produced by rolling dice really random? An honest program doesn't simulate crooked dice. Is the number series that represents throwing dice truly random?

Testing Randomness—Over a very long time, each number in a series of random numbers should appear as often as any other number.

Here is a program that checks each number in a pseudo-random series with integers from 0 to 9. It displays the number of zeros, the number of ones, and so forth, so you can see if the number generator is producing about the same quantity of each number. It looks formidable, but it is easy to enter if you overtype lines 100-190 and then change them as shown.

When it first starts running, you will see how people "get lucky" when playing games or gambling. For a while, some numbers appear more often than others. You will also see that the "law of averages" seems to work, eventually. Enter

```
NEW
10 PRINT CHR$(147)
20 N%=10*RND(1)
30 PRINT CHR$(19):PRINT
40 PRINT"THIS NUMBER IS: "N%:PRINT
50 PRINT"DISTRIBUTION OF "CT" NUMBERS:"
60 CT=CT+1:PRINT
100 PRINT"0: "N0: IF N%=0 THEN N0=N0+1
110 PRINT"1: "N1: IF N%=1 THEN N1=N1+1
120 PRINT"2: "N2: IF N%=2 THEN N2=N2+1
130 PRINT"3: "N3: IF N%=3 THEN N3=N3+1
140 PRINT"4: "N4: IF N%=4 THEN N4=N4+1
150 PRINT"5: "N5: IF N%=5 THEN N5=N5+1
160 PRINT"6: "N6: IF N%=6 THEN N6=N6+1
170 PRINT"7: "N7: IF N%=7 THEN N7=N7+1
180 PRINT"8: "N8: IF N%=8 THEN N8=N8+1
190 PRINT"9: "N9: IF N%=9 THEN N9=N9+1
200 GOTO 20
```

LOG()

This takes the *natural logarithm* of an expression in parentheses. The base number for natural logs is the numeric constant e . The value of e is approximately 2.71828. Enter

```
PRINT LOG(25)
```

EXP()

This is the reverse of LOG(). It raises e to a power. Enter

```
EXP(8)
```

```
EXP(1)
```

That displays the constant e , raised to the first power, which is just the value of e .

HOW TO DO CALCULATIONS IN A PROGRAM

Most of us try to put money in a savings account or some other investment that pays interest,

hoping it will become a big pile someday. When you see how fast money grows in a savings account, you may decide to reduce your expectations or start saving more money.

Usually, interest is *compounded*. At the end of a certain period of time, called an *accounting period*, the interest on your account is calculated and added to the principal. Compounding makes your money grow faster because you earn interest on interest.

The accounting period can be a year, a month or a day. If interest is stated as a yearly rate, it is divided by the accounting period to find the rate to use for each period. For example, 12% annual interest is 1% per month.

The formula for compound interest is $SUM = PRINCIPAL * (1 + RATE)^N$, in which SUM is the amount in the account. PRINCIPAL is the original investment. RATE is the interest rate for the accounting period being used, expressed as a decimal instead of a percentage. N is the number of accounting periods.

To solve an equation like that in a program, first write program lines to establish values for all of the variables except the one that you want the program to calculate. In this case, establish values for everything except SUM. Then, enter the equation on a program line and let the program solve it for you. Here is a program that does that. Enter

```
NEW
10 PRINT CHR$(147)
20 INPUT "ORIGINAL INVESTMENT";P
30 INPUT "ANNUAL INTEREST RATE AS %";I
40 RATE=I/100:REM CONVERT % TO DECIMAL
50 INPUT "ACCOUNTING PERIODS/YEAR";NP
60 RATE=RATE/NP:REM RATE/PERIOD
70 INPUT "CALCULATE HOW MANY YEARS";Y
80 N=NP*Y:REM PERIODS IN Y
90 SUM=P*(1+RATE)^N
100 SUM=INT((SUM*100)+.5)/100:REM ROUND OFF
110 PRINT:PRINT "THE SUM IS $"SUM
```

List it to check your typing. Then run it. To test a program, it's a good idea to enter some easy numbers at first. For the original investment, enter \$100.

For annual interest rate, enter 10, not 10%. For accounting periods/year, enter 1. That means interest will be calculated only once, at the end of the year. For the number of years, enter 1.

The result should be \$110. If it is, the program is probably OK. Try investing \$100 at 12%, compounded monthly for one year. Monthly compounding means that there are 12 accounting periods per year. The result should be \$112.68.

Here's a rule of thumb to estimate how many years it will take to double your money. Divide 72 by the annual percentage rate of interest. For example, at 10% annual interest, your money should double in 7.2 years approximately.

Run the program. Invest \$100 at 10% for 7.2 years, with one accounting period per year. You should have \$198.62.

Occasionally, we read in the newspaper about somebody who claims to own England or some other country because his great-great ancestor put some money in the bank there and never drew it out.

Suppose you find a document showing that your ancestor deposited \$3 in the Big Bank of Kentucky 200 years ago. It has been drawing interest at 3%, compounded annually. Do you own the bank? Run the program and see. That won't even pay the lawyer.

What if the interest rate had been 8%? I'll go with you to collect the money. Compound interest is a powerful way to multiply money, but it is much more powerful at high interest rates.

Financial planning is good for everybody. It makes us humble—or maybe desperate. To plan your financial future, get a personal-finance book at the library. These books show how to calculate the present value of future money, the future value of present money, mortgage interest, how much you should invest annually to have a big pile someday, and other miracles.

Program those equations and figure out how to provide for your future in luxury, relaxing somewhere in the golden sunshine. When you make it, you will probably show your neighbor the actual computer program that made it all possible. Then, he will show you his.

SOCIAL SECURITY

You may know someone who can use this little program. Under present law, a person in the U.S. can start drawing reduced social-security payments at age 62. The payments are 80% of what they would be if the person waited until age 65.

This program compares the total amount received in future years, based on drawing 80% starting at age 62 or 100% starting at age 65.

```

NEW
10 PRINT CHR$(147)
20 PRINT" TOTAL PENSION, BASED ON 100%=
$100"
30 PRINT
40 PRINT"AGE"TAB(10)"80% AT 62"TAB(25)
"100% AT 65"
50 PRINT:T2=0:T5=0
60 FOR AGE=62 TO 80
70 T2=T2+80
80 IF AGE<65 GOTO 100
90 T5=T5+100
100 PRINT AGE TAB(12)T2 TAB(28)T5
110 NEXT:PRINT CHR$(145);:REM CRSR UP

```

List it and check your typing. At line 110, CHR\$(145) moves the cursor up one row to prevent the BASIC prompt from causing a scroll.

Run it. This program assumes an entitlement of \$100 at age 65. If the entitlement is actually \$400, multiply the numbers by 4, or whatever is appropriate. Or else, just consider the numbers in the table to be percentages.

As you can see, the person who retires at 62 collects more total money through age 75. At age 76, it's a tie. After that, it would have been better to wait until age 65. Obviously, this program doesn't make the decision. But, it provides data that may help. If you want to see what happens past age 80, change the program.

Once you start exploring a problem by writing a program, a lot of "what if" ideas will occur to you.

Look at the zeros at the top of the right column on the screen. If a person can choose between these two options, then he must be able to do without extra income until age 65. In that case, consider taking 80% at 62 but not spending it. Invest it until age 65 and then add the interest that it earns to the pension.

That requires an investment phase for the first three years, using the compound-interest formula discussed earlier. Then, this program would be modified to add the interest received to the 80% pension, starting at age 65. Depending on the interest rate, that makes the 80% pension a better deal until about age 78—which is the approximate life expectancy for people in their early sixties.

TRIGONOMETRY

If you are not interested in this subject, skip ahead to the section entitled *FUNCTIONS*. Commodore BASIC has four trig functions: sine, cosine, tangent and arctangent. With these, other trig functions can be calculated using standard formulas that you can find in math books.

SIN(radians)

Takes the sine of an angle expressed in radians. Enter

```
X=SIN(1)
```

```
PRINT X
```

COS(radians)

Takes the cosine of an angle expressed in radians. Enter

```
X=COS(1)
```

```
PRINT X
```

TAN(radians)

Takes the tangent of an angle expressed in radians. Enter

```
X=TAN(1)
```

```
PRINT X
```

ATN(tangent)

This is an inverse trig function that you may know as $\arctan()$. It provides the angle in radians for the specified tangent value. Enter

```
X=ATN(1)
```

```
PRINT X
```

FUNCTIONS

Some BASIC words are called *functions*. A function derives one value from another by a mathematical procedure. An example is the function $\text{SQR}()$, which takes the square root of a number or numeric variable placed in the parentheses.

The value of $\text{SQR}(X)$ is a *function* of the value of X . That means it is determined by the value of X .

HOW TO DEFINE YOUR OWN FUNCTIONS

If a program uses a complicated equation at more than one place in the program, you will prefer not to type it repeatedly into the program. One way to do that is put the equation in a subroutine and call the subroutine when needed.

If the equation or function were already part of BASIC, you wouldn't have to type it at all. Just use the function by typing its name and placing a number or numeric variable in the parentheses.

For example, $Y=\text{SQR}(9)$ uses the function $\text{SQR}()$ to take the square root of 9, which is then assigned to the name Y . Inside the computer, a mathematical routine is used to take the square root, but you don't have to type that routine.

Defining a Function—BASIC allows you to define up to 10 functions by typing them into the program once and giving them names. What you type is the equation that provides the needed function. Then you can use that equation just by referring to its name.

This is best explained by an example. Suppose you are writing a program that must repeatedly calculate the area of a circle. That is not a complicated equation, but let's define it as a function anyway.

The area of a circle is calculated by $A=\pi*R^2$. A is area and R is the radius of the circle. The

number π is a constant. It's value is built into the computer. To type a π symbol, press SHIFT- \uparrow . To see its value, enter

```
PRINT  $\pi$ 
```

The value of π is 3.14159265. To be sure the equation works, enter

```
AREA= $\pi$ *3 $\uparrow$ 2
```

```
PRINT AREA
```

The result should be 28.2743339. That is the area of a circle of radius 3. If radius is entered in feet, the area is in square feet.

You must choose a name for the new function that you will define. Because it calculates the *Area* of a *Circle*, let's call it *AC*. It can be any legal numeric variable name. Then, you must decide what *AC* will be a function of. The area of a circle is determined by its radius. Therefore, *AC* is a function of *R*.

Now, you have an equation for the function, a name for the function, and the name of the variable that will determine the value of the function.

You are ready to define the function. That is done by a DEF FN statement, which means *define function*. Enter

```
NEW
```

```
10 REM DEMONSTRATE DEF FN
```

```
20 PRINT CHR$(147)
```

```
30 DEF FN AC (R)= $\pi$ *R $\uparrow$ 2
```

Line 30 says *define function AC of R as π *R \uparrow 2*. *AC* is a function of *R* because *R* determines its value. The DEF FN statement tells the computer how to calculate area when *R* is specified.

Using a User-Defined Function—In a program, the name of this function is FN AC. FN is necessary so the computer will know that this is a user-defined function, not an ordinary variable named AC.

Once defined, it is used like any other function, such as SQR(). When using SQR(), put a number in the parentheses and the SQR() function produces the square root of the number. Some computer books say it *returns* the square root.

This function is FN AC (). Put a radius inside the parentheses and the function produces the area of that circle. Now, you can use FN AC in the program. Enter

```
40 REM NOW USE FN AC
```

```
50 INPUT "ENTER RADIUS";R
```

```
60 AREA=FN AC (R)
```

```
70 PRINT AREA
```

Line 50 accepts a value of *R* from the keyboard. Line 60 puts *R* into the parentheses. FN AC (R) then calculates the area, which is assigned to the variable name AREA.

Run the program and enter 3 for the radius. The area should be 28.2743339, the same as calculated earlier. If you were planning to calculate the areas of some circles today, this is a good time to do it.

FUNCTIONS WITH MORE THAN ONE VARIABLE

Function AC of R used only one variable to calculate area—the variable *R*. Let's define a function that uses two variables.

Suppose you are writing a program to calculate miles per gallon (MPG) for an automobile. Obviously, MPG is calculated using two variables—miles driven and gallons of fuel used. Both of those values will be entered from the keyboard, using *M* as the variable name for miles and *G* as the name for gallons.

Mathematical Operations

The formula is M/G . Let's name the function $MPG()$. Now, we must decide what it is a function of. If it is a function of gallons, G will be placed in the parentheses. If it is a function of miles, M will be placed in the parentheses. The result will be the same either way.

Let's make MPG a function of M , the number of miles driven. Enter

```
NEW
10 REM CALC MILES/GALLON
20 PRINT CHR$(147)
30 INPUT "ENTER MILES DRIVEN";M
40 INPUT "GALLONS USED";G
50 DEF FN MPG(G)=M/G
60 FE=FN MPG(G):REM FUEL ECONOMY
70 PRINT:PRINT "FUEL ECONOMY IS"FE"MPG"
```

Run the program. Suppose you drove 100 miles and used 5 gallons. Enter the data. Fuel economy is 20 MPG.

Change the program so $FN\ MPG$ is a function of M instead of G . To do that, change (G) to (M) in lines 50 and 60. Run the program again and enter the same data. The result should be the same.

Run the program. Enter 1000 miles and 30 gallons. Usually, MPG is not reported with that many decimal points. Add a roundoff routine to report MPG to only one decimal place.

The equation used to define a function can be as complicated as necessary with as many variables as necessary. Only one of the variables is used to define the function. In this example, it doesn't matter much which variable is used in the definition. Sometimes, one variable will seem more logical than another.

REVIEW

This chapter is like a buffet. I suggest that you look it over again and take more of whatever you want from it. If you plan to write programs using a lot of math, everything in the chapter may be of interest.

If some of the information is beyond your present knowledge or interest, that's OK. As you write programs, your interest and knowledge of math will increase automatically.

BASIC words used in this chapter are in the accompanying table. Please look them up in Appendix B.

BASIC WORDS USED IN THIS CHAPTER	
ABS()	INT()
ATN()	LOG()
COS()	RND()
DEF FN()	SIN()
EXP()	SGN()
FN()	SQR()
	TAN()

12 Screen Displays And Menus

To manage the screen display, you must control what prints, where it prints and what it looks like. Also, you must know how to erase parts or all of the screen.

Screen displays are used to provide *instructions* to use a program and to display the *results* of using the program. Every display is important because it is the interface between program and user. If the display is clear and easy to understand, the user will like the program and probably use it correctly.

Instructions are often a list of options that the user can select. Such lists are usually called *menus*.

HOW CHARACTERS ARE PLACED ON THE SCREEN

This is a quick review of information presented in earlier chapters. Characters and symbols are placed on the screen by a display controller that uses screen-display codes. These codes were discussed in Chapter 7.

To create a screen display, the display controller refers to a screen memory map at locations 1024 to 2023. The screen-display codes representing the characters to be displayed are stored in that map. The location of a code in the map governs the location of that character on the screen. There are 1000 locations in the screen memory map, corresponding to 1000 print locations on the screen.

The display controller also refers to a color memory map at locations 55296 to 56295. Character color codes for each screen location are stored in the color memory map. Each character on the screen is displayed, using the color stored at the corresponding location in the color memory map. The character color codes range from 0 to 15, to select any of 16 character colors.

There is only one set of screen-display codes, ranging from 0 to 255. However, there are two character sets stored in ROM. By poking a number into memory location 53272, you can select one of the two character sets. POKE 53272,21 selects upper case and graphics. POKE 53272,23 selects upper and lower case.

When the display controller finds a screen-display code in the screen memory map, it refers to the selected character set and displays a character from that set. A screen-display code produces one of two characters, depending on which character set has been selected.

Characters can be displayed by poking screen-display codes into the screen memory map and character color codes into the corresponding locations on the color memory map.

Another way to display characters is to press character keys on the keyboard. These keystrokes can be executed in the immediate mode or stored in a program.

ASCII codes are used to represent keystrokes. PRINT A and PRINT CHR\$(65) have the same meaning to the computer.

When the computer “sees” keystrokes or their ASCII codes, it does one of two things. If the keystroke or code should display a character on the screen, that keystroke or code is translated automatically into the equivalent screen-display code number and placed in the screen memory map, so it will be displayed.

If the computer receives a PRINT statement using a control keystroke, it does whatever the control keystroke code calls for, such as clear the screen or perform a carriage return.

The computer recognizes control keystrokes in either of two forms: as a control symbol, such as a reversed heart, or as an ASCII code, such as CHR\$(147). The programmer must be aware of the quote mode and the insert mode when typing program lines.

Character color can be set for individual characters by poking codes into the color memory map. It can also be set by ASCII codes. When set by an ASCII code, it applies to all following characters on the screen until another color is selected.

Background color and border colors are set by poking color codes from 0 to 15 into memory. Location 53280 controls border color. Location 53281 controls background color.

In case you need to refer to them while reading this chapter, ASCII codes, screen-display codes and other display-related information are in Appendix A.

CLEARING THE SCREEN

The fastest way to clear the screen is PRINT CHR\$(147). A slower method is to run a loop that pokes screen-display code 32 into each location of the screen memory map. To see this method, use a loop to poke the screen full of characters and then use another loop to rub 'em out. Enter

```
NEW
10 POKE 53272,21:REM UPPER CASE
20 FOR I=0 TO 999:REM 1000 LOCATIONS
30 POKE 1024+I,1:REM CHARACTER
40 POKE 55296+I,3:REM COLOR
50 NEXT
60 REM NOW CLEAR SCREEN
70 FOR I=0 TO 999:REM 1000 LOCATIONS
80 POKE 1024+I,32:REM POKE A SPACE
90 NEXT
```

The entire screen will be filled and then erased.

CLEARING SOME ROWS ON THE SCREEN

Even though inefficient to clear the entire screen, using a loop to poke code 32 into the screen memory map is the only way to clear selected rows on the screen.

To demonstrate the method, clear the screen and put row numbers at the beginning of each row. Start with the cursor at the top-left corner. Press 0 and RETURN. Then press 1 and RETURN. Do that through 23. Type 24 and do not press RETURN. That puts row numbers on the screen. Move the cursor up to row 2. Overtyping the 2 by entering

```
FOR I=1384 TO 1663:POKE I,32:NEXT
```

It clears rows 9 through 15. Finding the beginning screen-memory-map location for any screen row is easy, remembering that the top row is 0. Use this formula: $LOCATION = 1024 + (ROW \times 40)$.

When clearing complete rows on the screen, the simplest way to find the last location to be cleared is to find the first location *on the following row* and then subtract 1.

Using the immediate mode, write a loop to clear rows 15 through 20 and test it.

MAKING A WINDOW ON THE SCREEN

Sometimes, it is useful to clear a rectangular window on the screen by poking 32 into each

location in the rectangle. You can use the same method to put characters where you want them and select the character color. Enter

```
NEW
10 PRINT CHR$(147)
20 FOR ROW=5 TO 10
30 FOR COL=20 TO 30
40 POKE 1024+(ROW*40)+COL,81:REM GRAPHIC
50 POKE 55296+(ROW*40)+COL,1:REM WHITE
60 NEXT:NEXT
70 PRINT CHR$(19);:REM HOME CURSOR
80 PRINT"NORMAL COLOR AGAIN"
```

Line 20 designates rows on the screen. Line 30 selects columns 20 to 30. Line 40 calculates corresponding locations on the screen memory map and pokes screen-display code 81.

Line 50 calculates corresponding locations on the color memory map and pokes color-code 1 into each location. That produces white characters. Run it.

CONTROLLING SCREEN LOCATION BY PRINT AND TAB STATEMENTS

Often there is more than one way to do something. This routine produces the same display. Enter and run

```
NEW
10 PRINT CHR$(147)
20 PRINT CHR$(5):REM WHITE
30 PRINT CHR$(19);:REM HOME CURSOR
40 FOR I=0 TO 4:PRINT:NEXT: REM
  CURSOR TO ROW 5
50 FOR ROW=5 TO 10:PRINT TAB(20)
60 FOR COL=20 TO 30:PRINT CHR$(113);
70 NEXT:PRINT:NEXT
80 PRINT CHR$(154)CHR$(19)"NORMAL COLOR
  AGAIN"
```

List the program. When poking codes into the memory maps, in the preceding demonstration, ROW and COL were used both as loop counters and to calculate memory locations in the POKE statements. In this program, ROW and COL are used only as loop counters.

Line 40 moves the cursor down to row 5. Line 50 would work the same if it were FOR ROW=1 TO 6. Either way, it operates 6 times. Writing it as FOR ROW=5 TO 10 makes it clearer that the program is printing on rows 5 to 10. At each of those rows, the TAB statement moves the cursor to column 20.

The loop at line 60 would do the same thing if it counted from 1 to 11 instead of 20 to 30. Writing it as FOR COL=20 to 30 makes it more clear that the program is printing on columns 20 to 30.

The semicolon at the end of line 60 holds the cursor on the same screen row while it prints all of the characters. At line 70, the first NEXT is the end of the inner loop. The PRINT statement moves the cursor to the next row, to begin printing the next row of characters at column 20 of that row.

Change this program to print your name in each row instead of graphics characters.

WHICH IS BEST?

For the Commodore 64, there is no special advantage of one method over the other. Poking numbers into the memory maps is a special technique for this computer. Controlling a screen

display by using PRINT statements with TAB() or SPC() statements to manage the cursor is a more general method.

I think you will become a better programmer if you use the second method—controlling the cursor—as your usual way of locating characters on the screen. Poke characters into the memory maps only when there is some reason to do it that way.

A good general procedure to control the cursor position is done in three steps. Home the cursor. Move it down to the desired row with PRINT statements. Move it over by TAB() or SPC() statements.

When homing the cursor by PRINT CHR\$(19), it's a good idea to do it the same way everytime. If you use a semicolon after the statement, the cursor remains on row 0. If not, the cursor ends up on row 1. Knowing where the cursor is makes it easier to calculate how many PRINT statements are needed to move it down to the desired row on the screen.

Of course, you don't have to home the cursor each time you want to move it to a desired location. If you know exactly where it is, start moving it from there. Homing is just a handy way to know where it is.

Except when an INPUT statement is executing, the cursor is invisible when a program is running. Sometimes, it isn't where you think it is.

If you write a routine to position the cursor, and it doesn't work correctly, print something. Use a temporary statement to print any character, such as X. That will show you where the cursor was at that instant. Then, you can figure out how it got there instead of where you intended it to be.

MANAGING SPACES

When using PRINT statements, you will sometimes get spaces where you don't want them or not have a space where one is needed. Strings display exactly what is inside the quotation marks or contained in the string. Positive numbers print with a leading and trailing space. Negative numbers print with a leading minus sign and a trailing space.

Printing a list of items with a single PRINT statement causes one to follow the other on the same row. Sometimes, spaces cause a problem in printing a list of items. You will see that in some of these examples. When there is a problem, following examples show ways to solve it. I think they are all self-explanatory. Enter

```
PRINT "A" "B"
PRINT 12345
PRINT -12345
PRINT 12345 6789
PRINT 12345 "W"
PRINT 12345"W"
PRINT "W" 12345
PRINT "W" -12345
X=-12345
PRINT "W" X
X=1
Y=2
PRINT X
PRINT Y
PRINT XY
PRINT X Y
PRINT X,Y
A$="A"
PRINT A$ X
```

(Program continued on next page.)

```
PRINT X A$
PRINT X,A$
PRINT X SPC(5) A$
PRINT X" "A$
```

When you write a PRINT statement, test it. If it doesn't execute correctly, fix it.

MENUS

Many programs display one or more menus to allow the user to choose what the program will do next. For example, in a business program, a menu might display three choices:

```
CALCULATE SALES
CALCULATE PROFIT
CALCULATE INVENTORY
```

There are four common methods used to make a choice.

One is to number the menu items. To select an item, the user presses a number key. Another is to begin each item with a different letter. To make a choice, press the letter key that corresponds to the first letter of the desired item on the menu.

Function keys are often used to make menu choices. I think it is because computers have function keys and programmers want to do something with them. There is no essential difference between pressing function key 1, number key 1 or letter A.

A more sophisticated method is to move a pointer symbol opposite the menu choice and press the RETURN key. All four of these methods will be demonstrated.

Other methods of moving the cursor to make a selection include using joysticks and mice. Joysticks are used with computer games. A mouse is a gadget on the end of an electronic cable connected to the computer. The position of the cursor is controlled by moving the mouse around on a flat surface. For menu selection, using a joystick or mouse is like using a broomstick to turn on a light switch. They do a simple task in a complicated way.

MENU SELECTION BY NUMBER

This method is simple and very easy to write. Enter

```
NEW
10 PRINT CHR$(147)
20 PRINT:PRINT:PRINT
30 PRINT"CHOOSE BY NUMBER":PRINT
40 PRINT" 1 - SALES"
50 PRINT" 2 - PROFIT"
60 PRINT" 3 - INVENTORY"
70 GET C$:IF C$="" GOTO 70
80 C=VAL(C$):PRINT CHR$(147);
90 IF C=1 GOTO 1000
100 IF C=2 GOTO 2000
110 IF C=3 GOTO 3000
120 IF C<1 OR C>3 THEN PRINT
    "ERROR. CHOOSE AGAIN.":GOTO 20
1000 PRINT"SALES":END
2000 PRINT"PROFIT":END
3000 PRINT"INVENTORY":END
```

Usually, it is simpler to take the input to a GET routine as a string variable, as this program does. C means *choice*. Line 80 converts C\$ to a numeric, C, and clears the screen. The screen is cleared because a menu choice has been made. Displaying the menu is no longer necessary.

Lines 90-110 test the value of C and branch accordingly. The error trap at line 120 follows the lines that execute if a valid choice is made. This trap uses screen row 0 to tell the user that an error was made. Then it reconstructs the screen display and waits for another input.

Lines 1000, 2000 and 3000 are used to test the program. They represent routines that begin at those locations in the program to perform the selected actions.

Run the program. Make all three selections and then make some invalid selections to test the error trap.

ON N GOTO

There is a simplification of routines involving choice by number. It uses the BASIC statement ON N GOTO, in which N is any numeric variable name. This statement can replace a series of IF statements, such as lines 90, 100 and 110.

Delete lines 90, 100 and 110. Then enter

```
100 ON C GOTO 1000, 2000, 3000
```

List the program. In line 100, C is the numeric variable representing the menu choice. It has three possible values, 1, 2 and 3. The GOTO statement lists three line-number destinations. If the *first* of the three possible values for C is selected, line 100 will branch to the *first* of the three line numbers that are listed. If C has its second possible value, the program branches to the second listed line number, and so forth.

Run the program to test it. It works the same, but is a little shorter and easier to write. If an invalid selection is made for C, line 100 will not execute, and the program will move to the following line. The error trap at line 120 will execute.

In an ON N GOTO statement, the possible values of N must begin with 1 and be a continuous series, such as 1, 2, 3, 4. That's because the list of destination line numbers cannot begin with a zero item or an item with a negative number.

The number of destination line numbers in the list must be the same as the number of possible values of N.

ON N GOTO is sometimes used with SGN() to make an *automatic* jump in a program, depending on the result of a calculation. SGN(X) has the value -1, 0 or 1 when X is negative, zero or positive.

To use SGN(X) as the variable in an ON N GOTO statement requires adding 2 to the value of SGN(X). Then, instead of having values that range from -1 to 1, its range will be from 1 to 3. The statement ON SGN(X) + 2 GOTO will make an automatic jump, depending on the sign of X.

ON N GOSUB

Another form of this statement is ON N GOSUB. It branches to subroutines, which then return to the following statement in the program.

CLEARING THE KEYBOARD BUFFER

As you know, the keyboard buffer holds up to 10 keystrokes until they are accepted by the computer. Most people keep their fingers on the keyboard while running a program. It is easy to press a key accidentally, perhaps without being aware of it.

If a loop is running or the computer is otherwise occupied just before this menu routine executes, it is possible for a keystroke to be waiting in the buffer when the GET routine at line 70 executes. If so, the GET routine will accept the first keystroke waiting in the buffer instead of the *next* keystroke from the keyboard.

To demonstrate that, put a loop ahead of line 10. Enter

```
5 FOR DL=1 TO 2000:NEXT
```

Run the program. Press Q immediately, while the delay loop is running. Because Q is an invalid menu choice, the menu routine *begins* by telling the user that he made an error.

Run it again and press 2 while the loop is operating. The program doesn't stop for another

keyboard input. If pressing 2 was inadvertent, the user may be running a branch of the program that he didn't intend to run.

Avoid this potential problem by emptying the keyboard buffer before making menu selections. It should be emptied just before the GET statement executes. Enter

```
65 FOR MT=1 TO 10:GET THROWAWAY$:NEXT
```

Line 65 empties the buffer and “throws away” whatever is in it—by doing nothing with the characters that it finds in the buffer. The loop counter is MT. MT stands for *empty*. Run and test the program again. It ignores keystrokes made before the menu is displayed.

Notice the difference between this GET routine and the one you have been using. This routine doesn't stop the program and wait for a keystroke that is not “ ”.

It takes whatever is in the keyboard buffer, and does it 10 times. If “ ” is in the buffer, it assigns “ ” to the variable THROWAWAY\$ and gets the next character in the buffer. After 10 operations of the MT loop, the keyboard buffer must be empty because it can hold only 10 keystrokes.

MENU SELECTION BY LETTER

This method is similar to using numbers to make menu selections. Instead of pressing a number, the user presses the letter key that matches the first letter of the menu item being selected.

Menu items must begin with different letters. You will be surprised at how often the best words for a menu begin with the same letters. Entering and running this program is optional. You can see what it does just by looking at it.

```
5 FOR DL=1 TO 2000:NEXT
10 PRINT CHR$(147)
20 PRINT:PRINT:PRINT
30 PRINT"CHOOSE BY LETTER":PRINT
40 PRINT" SALES"
50 PRINT" PROFIT"
60 PRINT" INVENTORY"
65 FOR MT=1 TO 10:GET THROWAWAY$:NEXT
70 GET C$:IF C$=" "GOTO 70
80 PRINT CHR$(147);
90 IF C$="S" GOTO 1000
100 IF C$="P" GOTO 2000
110 IF C$="I" GOTO 3000
120 PRINT"ERROR. CHOOSE AGAIN.":
    GOTO 20
1000 PRINT"SALES":END
2000 PRINT"PROFIT":END
3000 PRINT"INVENTORY":END
```

The error trap at line 120 is simplified, compared to the same line in the preceding demonstration. The reasoning is that the program can't reach line 120 if a valid choice is made from the menu. Therefore, line 120 can execute only when an invalid choice has been made. It is not necessary to test the value of C\$ at line 120. The same error trap could have been used in the preceding demonstration.

USING FUNCTION KEYS TO MAKE SELECTIONS

When a function key is pressed, it generates an ASCII code number that can be detected by a

GET routine but not by an INPUT statement. To use function keys for menu selection, change the program like this:

```
5 FOR DL=1 TO 2000:NEXT
10 PRINT CHR$(147)
20 PRINT:PRINT:PRINT
30 PRINT"PRESS FUNCTION KEY":PRINT
40 PRINT" F1 - SALES"
50 PRINT" F3 - PROFIT"
60 PRINT" F5 - INVENTORY"
65 FOR MT=1 TO 10:GET THROWAWAY$:NEXT
70 GET C$:IF C$="" GOTO 70
80 PRINT CHR$(147)
90 IF C$=CHR$(133) GOTO 1000
100 IF C$=CHR$(134) GOTO 2000
110 IF C$=CHR$(135) GOTO 3000
120 PRINT "ERROR. CHOOSE AGAIN.":
    GOTO 20
1000 PRINT"SALES":END
2000 PRINT"PROFIT":END
3000 PRINT"INVENTORY":END
```

Run and test this version. It works about the same as the preceding two methods. Keep the program in memory.

WHICH IS BEST?

The three methods demonstrated are simple. The next one isn't. Among the simple methods, the main difference is how they appear to the user of a program.

I have a theory that a menu communicates with the user's mind and the communication should be as direct as possible. When numbers are used to make a choice, the user must first scan the menu items to select one. Then, he must mentally associate the number of that item with the purpose of that item. Then, he finds the number key and presses it. He ends up pressing 2 for PROFIT, if that was the choice.

Using function keys involves a similar mental process. They may have an advantage because they are special and easy to find on the keyboard. Conversely, the user may wonder why the function-key numbers on the menu are 1, 3 and 5. He may be distracted by wondering if there are choices numbered 2 and 4 that he should know about. If you use function keys 2, 6 or 8, you must tell the user how to select them. Using a SHIFT key to select F2 makes the menu-selection process more complicated and difficult for a non-programmer to cope with.

I think that using initial letters of the menu items is the best method among these three because the user makes a quick association of the item being selected and the method of selecting it. Press S for SALES. But he may have to look around on the keyboard to find the S key. I may be biased because I can find S—often on the first try.

USING A MOVABLE SYMBOL TO MAKE MENU SELECTIONS

With the method to be demonstrated, the user presses the space bar one or more times to move a pointer symbol so it is opposite the desired item on the menu. Then, he presses RETURN, and the program executes that choice. The symbol in this program is a diamond shape, CHR\$(122). Any symbol can be used as the pointer.

The plan is to display the choices and locate the pointer opposite the first choice. Pressing the

space bar moves the pointer down one row. When the pointer is at the choice on the bottom of the list, pressing the space bar moves it back up to the top.

The program keeps track of which screen row the pointer is in. When RETURN is pressed, the program uses the row number to determine which selection was made. It branches accordingly, with an ON N GOTO statement.

During selection from the menu, there are only two valid keystrokes, the space bar and RETURN. The space bar produces CHR\$(32) and RETURN produces CHR\$(13). These keystrokes are received by a GET routine. All other keystrokes are rejected by an error trap.

When a program is running, the cursor is not visible except when executing an INPUT statement.

With Commodore BASIC, it is necessary to use a symbol for display. This program uses CHR\$(122). You can use any other symbol, including a rectangle resembling a cursor. Whatever you use for a pointer is moved along the list of menu items by pressing the space bar.

To demonstrate this method, it will be simpler to write a new program. It's longer than usual, but it demonstrates some useful programming techniques.

Delete lines 5-120 of the program in memory but keep 1000-3000. Do it the easy way. List the program, so lines 1000-3000 are in view. Then enter NEW. Now there is no program in memory. Move the cursor into line 1000 and press RETURN. Now the new program has line 1000. Take lines 2000 and 3000 the same way. List the program.

Then enter the following lines. I put a lot of remarks in this program. You will probably understand it as you enter it.

```
10 PRINT CHR$(147)
20 PRINT "MOVE DIAMOND TO SELECTION BY
PRESSING "
30 PRINT "SPACE BAR. THEN PRESS RETURN."
40 PRINT:PRINT:PRINT
50 PRINT TAB(15)"SALES"
60 PRINT TAB(15)"PROFIT"
70 PRINT TAB(15)"INVENTORY"
75 STOP: REM FOR TEST
```

To test, run that segment. Notice how the spaces are managed on the program lines and the result on the screen. I have a personal preference for readable program lines, so I will usually spend extra time fixing lines so words don't wrap around on the screen.

If you remain more open-minded than I about how program lines should look on the screen, you will write programs faster. But I will think your programs are hard to read. List the program and enter

```
75 ROW=5:REM INITIALIZE
80 PRINT CHR$(19):REM HOME
90 FOR I=1 TO ROW:PRINT:NEXT:
  REM LOCATE CURSOR
95 REM DISPLAY DIAMOND
100 PRINT TAB(14)CHR$(122);
105 STOP: REM FOR TEST
```

Screen Displays And Menus

Line 80 homes the cursor. Line 90 runs a loop from 1 to ROW and executes PRINT statements to move the cursor down. When ROW is 5, the loop runs from 1 to 5.

Line 100 tabs over 14 spaces, and displays a diamond symbol produced by CHR\$(122). The semicolon holds the cursor at the next column on that line.

That places the diamond symbol opposite the first item on the menu. Now, the user can make a selection. Test by running the program. The menu should be displayed with a diamond symbol opposite the first item. List the program and enter

```
105 REM MAKE CHOICE
110 GET KS$:IF KS$=" "GOTO 110
120 IF KS$=CHR$(13) GOTO 200:
    REM CHOICE MADE
130 IF KS$=CHR$(32) THEN ROW=ROW+1:
    GOTO 150: REM MOVE DOWN
140 GOTO 110:REM TRAP
150 PRINT CHR$(157)CHR$(32):
    REM ERASE OLD DIAMOND.
160 IF ROW=8 THEN ROW=5:
    REM LOWER LIMIT FOR DIAMOND
170 GOTO 80:REM TO MOVE DIAMOND
190 REM SELECTION MADE
200 PRINT CHR$(147)
210 ON ROW-4 GOTO 1000, 2000, 3000
1000 PRINT"SALES":END
2000 PRINT"PROFIT":END
3000 PRINT"INVENTORY":END
```

Line 110 gets a keystroke. The user can either move the diamond down, to select a different menu item, or press RETURN to select the item on that row.

At line 120, if the keystroke is RETURN, the user has selected a menu item. It is the item opposite the diamond. Because the program always "knows" which row the diamond symbol is on, it also knows which menu item was selected. The row number identifies the menu item.

If the user pressed RETURN with the diamond on row 6, ROW is 6. The program jumps ahead to line 200 to execute the selected menu choice.

If the user did not press RETURN, the only other valid action is to press the space bar to move the diamond on the screen, so it is opposite a different menu item. If the user did not press RETURN, line 120 does not execute.

At line 130, if the user pressed the space bar, ROW is incremented by 1, and the program jumps to line 150. The new value of ROW will be used to display the diamond symbol again, one row down on the screen.

List the entire program. Line 140 is an error trap. If a keystroke was made, and it was not RETURN or the space bar, line 140 executes. The program jumps back to the GET routine to wait for another keystroke. The invalid keystroke is ignored.

If the error trap at line 140 does not execute, a valid keystroke was made. It was the space bar. The diamond symbol should be moved down one row on the screen. It must be erased at its present location, before being moved down, so it is not visible at two locations simultaneously.

Line 150 erases the diamond in two steps. The cursor, even though invisible, is in the column to the right of the diamond symbol. First, CHR\$(157) moves the cursor one column to the left. That puts the cursor on top of the diamond. Then, CHR\$(32) prints a space on top of the diamond, which erases it.

Line 160 sets a lower limit for the diamond symbol. The last item on the menu is on row 7. If the space bar is pressed enough times to cause ROW to equal 8, that would place the diamond below the last item on the menu. When ROW=8, line 160 sets it back to 5 again. This moves the diamond back to the top of the menu. The diamond appears to rotate from the bottom of the menu back up to the top.

At line 170, a new value for ROW has been produced by pressing the space bar. It must be in the range of 5 to 7. Line 170 jumps back to line 80. The program displays the diamond symbol again at the new location specified by the new value of ROW.

When the user presses RETURN to select the menu item opposite the diamond, line 120 jumps to line 200. Line 200 clears the screen.

Line 210 uses an ON N GOTO statement. The variable representing the menu choice is ROW-4. This causes the range of choices to be from 1 to 3, rather than 5 to 7. When ROW is 5, the first selection on the menu has been made. ROW-4 is 1. Line 210 jumps to the first listed line, which is line 1000. When ROW is 6, ROW-4 is 2 and line 210 will jump to line 2000.

Run the program to see how it works. Of the four methods demonstrated, I think this one is best because it requires the least from the user. It is longer than the other methods, but you can leave out most of the remarks.

Obviously, you can use any keystroke to move the pointer symbol on the menu, including function keys. For a person who is accustomed to using cursor keys, a cursor-down keystroke would be OK.

If you are writing a program to be used by people who are not programmers, the cursor keys may be intimidating. Also, what is being moved is not the cursor. It is a pointer symbol. That's why this routine uses the space bar.

USING COLOR IN A MENU

Color can be used to make a screen display more attractive, or to provide some functional value, or both. Users of programs appreciate anything that helps them run the program—sometimes without being aware of the help they are receiving.

It is helpful to use one color combination when a program is running and different colors when the program is waiting for a keyboard input. The different color is an effective prompt to the user that he should do something.

To demonstrate that, change these lines in the program:

```
10 PRINT CHR$(147)CHR$(151):POKE 53281,1
200 PRINT CHR$(147)CHR$(154):POKE 53281,6
```

I think that makes a worthwhile difference. Of course, you can do that with any method of menu selection, using any colors that you like.

Another possibility is to change the character color of the selected item. If the user presses RETURN with the pointer at INVENTORY, that word becomes a different color from the other menu items. You can do that by poking a color code into the corresponding locations on the color memory map.

If your program clears the screen immediately after a menu selection is made, you will have to put in a delay loop so the user can see the color change. I would not slow down a program just to decorate it with color. But some applications are made more effective with color changes.

STEPS IN PROGRAM DEVELOPMENT

Writing a good program is done in several steps:

- 1) Make a plan.
- 2) Make each segment work as soon as you write it.
- 3) Make the overall program work.
- 4) Make make it work right.
- 5) Make it foolproof with error traps.

6) Make it friendly with good menus and instructions.

7) Make it look good.

When you have a program working, you are about half through. By testing, you will find that it isn't working right because of oversights or mistakes. A financial program may work just fine when the balance is positive, but fail when it is negative. Fix those things.

By testing, you will find things that can be done from the keyboard that cause the program to fail. When somebody else runs your program, he will always do something you don't expect.

You may think that only dummies would do that. He will think that only dummies write programs that fail. He wins.

By running the program and pretending that you don't know how it works, you will find confusing instructions and menus. Better still, get someone else to run it while you watch without helping. That is often a painfully humiliating experience. When you get it fixed, ask the person to run it again. Work on the program until it helps the user and doesn't allow mistakes.

When you have put that much effort into a program, take the final step and make it look good. Use color, clear the screen when it has unneeded information on it, make interesting and effective displays, use sound effects and music if appropriate. Sound is discussed later.

Making a good program without making it look good is like building a house and not painting it.

REVIEW

Most of this chapter discusses programming methods using information presented earlier. ON N GOTO and ON N GOSUB are BASIC statements that haven't been discussed before. Look them up in Appendix B.

As an exercise, change the program in memory to use a different symbol for the pointer. Change the program to put more items on the menu. Add ACCOUNTS PAYABLE and ACCOUNTS RECEIVABLE to the menu.

13

Sorting Routines

Many programs put a list of items in some kind of order. If the items are words, such as names, you will usually put them in alphabetical order. If the items are numbers, you will usually put them in ascending or descending order.

The first step in sorting a group of items is to get them into a list—in any order. That is called an *unsorted* list. Then you sort it.

LISTS

In computer language, a list usually means a group of items in an array. An example is `ARRAY(1)`, `ARRAY(2)`, `ARRAY(3)` and so forth.

It is possible to make a list using some orderly arrangement of variable names, such as `A1`, `A2`, `A3` and so forth. An array is all at one location in memory, but individual variables may be scattered around in memory.

Both are lists because the item names form a sequence that can easily be read out of memory in order. Both can be sorted. It is usually best to put items to be sorted into an array and then sort the array. This chapter demonstrates sorting an array.

A SORTING PLAN

Before you write a sorting routine, you must decide what is to be sorted and what the sorting rules will be. To sort a list of cities and states, for example, you would probably decide to put them in alphabetical order. But, you could put them in alphabetical order by states, or by cities. Sorting by ZIP code is another useful possibility. It would depend on the purpose of the list.

Think about how the list will be used. Choose the the best arrangement of the list for that purpose. Then decide on the sorting rules that will put the list into that order.

A SORTING METHOD

All sorts are done by comparing items, two at a time. Suppose you are sorting numbers into numerical order—the smallest number on top. If one number is 4 and the other is 7, the sorting routine must compare them. Then it must put 4 higher in the list than 7.

When two items are compared, they will already be in an unsorted list. Depending on the result

of the comparison, the two items may be interchanged in the list, or they may be left as they are. Here is a short segment of an unsorted list:

7
3
9
4
6

Sorting routines normally make comparisons by moving along the list, from one end to the other. One “trip” along the list is called a *pass through the list*. Starting at the top, the first two items are compared. The number 3 is smaller than 7, so the first two items are interchanged. The result is

3
7
9
4
6

The next comparison is items 2 and 3, which are the numbers 7 and 9 on this list. Because 7 is smaller than 9, that pair of numbers would not be interchanged on this pass. The sort procedure continues to the bottom of the list. That completes the first pass.

Usually, the list is not completely sorted by a single pass. The sort routine returns to the top of the list and makes another pass. After several passes, the list is in order.

With the sorting method just illustrated, smaller numbers rise to the top during each pass—like bubbles in a glass of champagne. That sorting method is called a *bubble sort*.

There are other sorting methods, but the bubble sort is useful and a good one to begin with.

INTERCHANGING ELEMENTS IN AN ARRAY

When you are sorting items in an array, the program works with the array names of the items. If ARRAY(3) holds the value 7 and ARRAY(4) holds 3, the sort routine compares ARRAY(3) and ARRAY(4). Of course it is actually comparing the *values* of ARRAY(3) and ARRAY(4).

Interchanging items in an array requires a programmer’s trick. Here is a segment of an array:

ARRAY(3)
ARRAY(4) (value=99)
ARRAY(5) (value=77)
ARRAY(6)

If the list is to be put into numerical order, ARRAY(4) and ARRAY(5) must be interchanged. That is done by assignment statements.

The statement ARRAY(4)=ARRAY(5) does part of the job. It puts the value of ARRAY(5) into ARRAY(4). That moves the value 77 up to the next higher position on the list.

The next part is to put the value of ARRAY(4) into ARRAY(5). Using the method just shown, the original value of ARRAY(4) has been lost. The statement ARRAY(4)=ARRAY(5) caused both array elements to have the same value—both are now 77. Obviously, that won’t work.

To interchange two variables, a third “holding variable” is needed. Read these three statements:

```
100 HOLD=ARRAY(4):REM HOLD VALUE OF ARRAY(4)
110 ARRAY(4)=ARRAY(5):REM MOVE VALUE OF ARRAY(5) UP
120 ARRAY(5)=HOLD:REM OLD VALUE OF ARRAY(4) INTO ARRAY(5)
```

That works. Line 100 uses a holding variable, named HOLD, to hold the value of ARRAY(4). Then, line 110 puts the value of ARRAY(5) into ARRAY(4). Now, both array elements have the same value, but HOLD holds the original value of ARRAY(4). Then line 120 sets ARRAY(5) equal to HOLD. The interchange has been made.

BUBBLE SORTS

For short lists, up to perhaps 50 items, this method is practical. For long lists, it is too slow. You will prefer a faster sorting method.

Bubble sorts can begin at either end of the list. You can start each pass at the top and work down, or you can start at the bottom and work up. The basic idea is the same. Sometimes, one method has an advantage over the other.

Either way, adjacent items on the list are compared and interchanged according to the sorting rule. Repeated passes through the list are made until all items are in the correct locations.

Number of Passes—Suppose a list of numbers has 20 items and the smallest number is on the bottom. It should be on top.

Each sorting pass begins at the top and works to the bottom. During each pass, smaller items can move up only one place on the list. Therefore, it would require 19 passes through the list to complete the sort.

Having the smallest item on the bottom is the worst case. By the time it reaches the top, all other items will be in their correct locations.

If a list has 20 items to sort, using a bubble sort, the *maximum* number of passes is $20 - 1$. In general, a list with X items takes a maximum of $X - 1$ passes for a bubble sort.

With long lists, each pass takes more time, and there must be more passes. That's why sorting long lists with a bubble sort is very slow.

MAKING AN UNORDERED LIST

To have a list of numbers to sort, let's use the random-number generator in the computer.

This demonstration program uses a remark followed by dashes to separate major program segments. Each segment begins with a remark that says what it does. Each segment begins at a line number that is a multiple of 100—such as 100, 200 and so forth.

Even though it takes a little more typing and a little more space in memory, writing programs this way makes them easier to read and understand. Enter

```
NEW
10 PRINT CHR$(147):REM BUBBLE SORT DEMO
20 DIM NUM(15)
30 REM MAKE RANDOM NUMBER LIST
40 FOR I=1 TO 15
50 NUM(I)=INT(RND(0)*10):REM FILL ARRAY
60 NEXT I
70 REM -----
```

The first program segment builds a list of 15 random numbers. Line 20 dimensions an array NUM(15) to hold an array with subscripts up to 15.

Line 30 sets up a loop to run 15 times. Line 40 gets 15 random numbers, one at a time, and plugs them into the array NUM(.). On the first pass, it fills array element NUM(1), and so forth, until NUM(15) is filled.

On each pass, a random number is produced at line 50. The expression INT(RND(0)*10) begins with a random number between 0 and 1. Then it multiplies the number by 10, so it is in the range of 1 to 10. Then it takes the integer, so it is a whole number. The result is placed in the array NUM(I) at location I.

The sort routine would work the same using numbers with decimal fractions, but the display would not be as easy to read. To see the unsorted list, enter

```
90 REM DISPLAY UNORDERED LIST
100 T=0:REM TAB VALUE
110 FOR I=1 TO 15
120 PRINT TAB(T)NUM(I)
130 NEXT I
140 REM -----
```

Sorting Routines

Line 90 is a title for the routine beginning at line 100. This segment displays the unsorted list. A variable, T, is used to tab the list on the screen. This list is displayed with T=0, which places it fully to the left on the screen. The next display will show the result of the first sorting pass. It will be displayed with a tab value of T=2, just to the right of the first list.

Run the program. Run it again. Notice that the list is different each time it is generated. Using RND(0) gets a seed from the internal clock in the computer. Usually, the seed will be a different number each time the program is run, so the list will be different.

SORTING THE LIST

Now that the program has an unsorted list, let's sort it. Enter

```
190 REM BUBBLE SORT
200 FOR P=1 TO 14:REM MAKE 14 PASSES
210 PRINT CHR$(19):REM FOR DISPLAY
220 FOR C=1 TO 14:REM 14 COMPARISONS
230 HOLD=NUM(C)
240 IF NUM(C+1)<NUM(C) THEN
    NUM(C)=NUM(C+1):NUM(C+1)=HOLD
250 NEXT C
260 REM -----
```

Because there are 15 items on the list, the maximum number of passes required to sort the list is 14. Line 200 sets up a loop, using the counter P, to make 14 sorting passes through the list.

Line 210 homes the cursor so the next list to be displayed will be just to the right of the preceding list.

Because there are 15 items on the list, each pass requires 14 comparisons and possible interchanges. Line 220 sets up a C loop to make 14 comparisons during each pass of the P loop.

Line 230 is the hold variable. When the value of C is 1, it holds NUM(1).

Line 240 does the comparisons and interchanges. It compares NUM(C+1) to NUM(C). When C is 1, it compares NUM(2) to NUM(1). If NUM(2) is smaller than NUM(1), then NUM(1) is set equal to NUM(2). That puts the smaller number on top. Then NUM(2) is set equal to HOLD. This puts the original value of NUM(1) into NUM(2). The interchange is complete.

If NUM(2) is not smaller than NUM(1), line 240 does not execute, and the two values are not interchanged.

On the next pass through the C loop, C is 2. HOLD holds NUM(2) while NUM(3) is compared to NUM(2). As the loop runs, each pair of items on the list is compared and interchanged if necessary. Line 250 is the bottom of the C loop.

Number of Comparisons—The C loop runs only to 14 even though there are 15 items to be sorted. That's because one of the items being compared is NUM(C+1). When C is 14, NUM(C+1) is 15. That gets the bottom item on the list.

If C ran to 15, the program would sort NUM(16) into this list. What is the value of NUM(16)? The NUM() array is now in memory. To find out, enter

```
PRINT NUM(16)
```

When an array is dimensioned to 15, calling for NUM(16) produces an error. If the array were dimensioned to a larger value, such as 20, but NUM(16) had never been filled since the program was run, its value would be zero.

Reaching down too far while sorting an array causes a problem either way. It may produce a program error. If not, it probably gets a zero. If it gets a zero, it will sort it all the way up to the top of the list. If you find zeros at the top of a list, and they weren't in the unsorted list, your program is reaching too far down the list in the sorting routine.

DISPLAY THE SORT

So far, this program produces an unsorted list of random numbers, displays it, and sorts it one time. To see the result of the first sort, enter

```
290 REM DISPLAY RESULT OF SORT
300 T=T+2:REM TAB OVER
310 FOR I=1 TO 15
320 PRINT TAB(T)NUM(I)
330 NEXT I
340 REM -----
```

Run the program. It should display an unsorted list and the result of the first sort. Notice that smaller numbers on the unsorted list moved up one position in the first sorting pass. The list is probably not completely sorted.

COMPLETE THE SORT

There was only one sort when you ran the program because the P loop at line 200 doesn't have a NEXT statement. The program ran to the end and stopped. Put a NEXT P statement in by entering

```
390 REM NEXT PASS
400 NEXT P
```

Run the program. It should display the unsorted list plus 14 sorts. Now you can plainly see how small numbers move up one position at a time. If the unsorted list has duplicate numbers, they end up at the correct locations.

Even though smaller numbers can move up only one position on each pass, larger numbers can drop like a rock. Find a large number in the unsorted list. Notice how far it drops from sort to sort. That's because a large number is involved in repeated comparisons during each pass.

For example, suppose NUM(4) is larger than NUM(5). They will be interchanged. The value at NUM(4) becomes the value at NUM(5). Then NUM(5) and NUM(6) are compared, but the value at NUM(5) used to be at NUM(4). If the new value at NUM(5) is larger than the value at NUM(6), it moves down another position.

It is possible for a large number to move from the top of the list all the way to the bottom in a single pass.

HOW MANY PASSES ARE REALLY REQUIRED?

Even though the maximum number of passes is the number of items minus one, that is the worst possible situation—when the item on the bottom belongs on top. On average, it takes about two-thirds as many passes as there are items on the list.

Run the program again. Starting at the extreme right, compare the sorted lists. Usually, several of them are identical. That means the sort was completed in fewer than the maximum possible number of passes.

Run the program several times and observe how many passes are needed to complete the sort. It depends on the arrangement of the unsorted list. Sometimes, a list can be sorted in only two or three passes.

STOP WHEN THE SORT IS COMPLETE

When the sort is complete, additional passes are wasted time. Your programs will run faster if you stop sorting when the sort is finished, no matter how many passes it takes.

Please list the program. Notice that when the sort is complete, line 240 will not execute during a pass.

The way to detect a completed sort is to use a variable as a flag. Set it to some value before each sorting pass. Change the value of the flag at the end of line 240. If line 240 does not execute, the value will not be changed.

Sorting Routines

At the end of each pass, test the flag to see if it was changed. If not, no interchanges were made by line 240. The sort is complete. Stop loop operation. These lines will demonstrate that. Enter new lines 205 and 395. Change line 240 as shown.

```
205 F=0:REM COMPLETION FLAG
240 IF NUM(C+1)<NUM(C) THEN
    NUM(C)=NUM(C+1):NUM(C+1)=HOLD:F=F+1
395 IF F=0 THEN P=14:REM STOP SORT
```

List the program. At the beginning of each pass through the P loop, line 205 sets flag F to 0. If line 240 executes, F is incremented by 1, so it is no longer zero.

At line 395, the flag is tested. If it is zero, the sort is complete. If so, line 395 sets the loop counter, P, to its maximum value of 14. Then line 400 executes. Because P is 14, line 400 does not run the loop again. This program ends. A working program would do something with the sorted list.

Run the program a few times. It stops sorting as soon as one pass is made without interchanging any items. The last two columns on the display will be identical. The sorting loop actually ran one more time after the sort was finished.

It's interesting to watch a few sorts, to see how the number of passes varies and to see the small numbers rising up on the list. Do it the easy way. Put a statement at line 410 that clears the screen and jumps back to line 40. Run it a while. Then break out of the loop and delete line 410.

ADDING AN ITEM TO A SORTED LIST

Some programs that manage lists keep them in sorted order all of the time. When a new item is received, it is immediately placed at the correct location in the list.

The sorting method is similar to a bubble sort, except that only one item is dealt with. In effect, the program looks along the list to find where the new item belongs, and inserts it at that point. For that reason, this sort is sometimes called an *insert* sort.

INSERT SORT

The simplest way to do an insert is to add the item to the list and then sort the list. With an array, the easiest place to add an item is at the bottom.

A good insert procedure is to stick the item on the bottom of the list. Then do comparisons starting at the bottom and working up.

Earlier, you saw that larger items in a bubble sort can drop to the bottom like a rock. That's because the sort demonstrated starts at the top.

If the sort starts at the bottom, a smaller item can move all the way to the top in a single pass. This sort starts at the bottom. Therefore it needs only one pass to insert one item. To see an insert sort work, change the dimension statement at line 20 to allow a larger array and enter these new lines:

```
20 DIM NUM(20)
410 REM -----
490 REM ADD AN ITEM
500 L=15:REM LENGTH OF LIST
510 L=L+1:REM PREPARE FOR ADDED ITEM
520 PRINT:PRINT"ENTER A NUMBER, 0-9"
530 GET A$:IF A$=""GOTO 530
540 N=VAL(A$)
550 IF N<0 OR N>9 GOTO 510
560 REM -----
```

When you are adding items to a list, the program must keep track of the number of items on the list. Line 500 starts that procedure by setting L equal to the number of items in the existing sorted list.

Line 510 increases L by 1 to prepare to receive a new item. The new item will be placed in the array at NUM(L), which automatically puts it at the bottom.

At line 530, this routine gets a number in the range of 0-9 from the keyboard. Larger numbers would be sorted, but the display is designed for single-digit numbers. Line 530 is a trap to prevent entering larger numbers. When this routine executes, the program has a new item to insert. Enter

```
590 REM INSERT SORT
600 NUM(L)=N:REM PUT AT BOTTOM
610 FOR C=L TO 2 STEP -1
620 HOLD=NUM(C-1)
630 IF NUM(C)<NUM(C-1) THEN
    NUM(C-1)=NUM(C):NUM(C)=HOLD
640 NEXT C
650 REM -----
```

This routine does the insert sort. It will always do it in one pass. It uses the variable L as the number of items on the list. L includes the added item. Because L was incremented at line 510, to prepare for the added item, its value is now 16.

Line 600 puts the new value, N, at the bottom of the array by placing it at NUM(L). Line 610 sets up a loop to count backward from L to 2, using STEP -1. It stops at 2 because one of the comparisons is NUM(C-1). That comparison gets item 1 in the array.

Line 620 provides a hold variable. Line 630 does the comparisons and exchanges, working upward from the bottom.

When sorting from bottom to top, you must be careful not to reach too high. If the sorting loop gets NUM(0), its value is probably zero, so it won't be moved down on the list. If it has some other value, it will be moved down.

You can always use item 0 in an array, but this program doesn't use it. If you are using item 0, and the sorting loop tries to get NUM(-1), an error results.

To see the insert, enter

```
690 REM DISPLAY NEW LIST
700 PRINT CHR$(19)
710 T=T+2
720 FOR I=1 TO L
730 PRINT TAB(T)NUM(I)
740 NEXT I
```

Run the program. After the sort is finished, enter a number that you will recognize in the final list. Notice where it appears. Run the program a few times to be sure it is working correctly.

ADDING ANOTHER ITEM

Because the array NUM() is now dimensioned for a maximum subscript of 20, you can add more items. To do that, enter

```
750 REM -----
790 REM ENTER ANOTHER ITEM
800 GOTO 510
```

Notice that line 510 increments L again to prepare for another added item. Run the program and enter new items until the array is full. Entering one more item will produce an error message.

SORTING STRINGS

Sorting routines work with either numeric or string variables. The difference is in how the variables are compared. With numeric variables, values are compared. 999 is larger than 678.

Sorting Routines

With string variables, the comparison starts with the leftmost character of each variable. What is compared is the ASCII codes for those characters. If one code is smaller than the other, the decision is made. If not, a comparison is made between the second characters in each string, and so forth. The decision is made when the first different characters are found. The string with the first smaller code number is “smaller” than the other one.

The usual purpose of sorting string variables, such as names, is to put them in alphabetical order. When two strings are compared, they will be placed in alphabetical order if the “smaller” string is placed above the larger string in the list.

ABC is smaller than XBC because the ASCII code for A is smaller than the code for X. AAAAX is smaller than AAAAZ. The ASCII codes are listed in Appendix A.

The strings do not have to be the same length. If ABCD is compared with ABCDE, ABCD is judged smaller. The fifth character in ABCD is a null, which has an ASCII code of zero. The ASCII code for the fifth character in ABCDE is 69.

If the strings being compared are numbers, the comparison is made using ASCII codes for the individual digits. Because smaller numbers have smaller ASCII codes, the result will be a numerical sort even though it was done by comparing ASCII codes. 12345 is smaller than 12555. 1234 is smaller than 12345.

If strings containing both letters and numbers are sorted, the result may not have any significance. It isn't alphabetical, and it isn't numerical. But, it will be sorted according to the individual ASCII codes for each character.

Demonstration—This routine will show how strings are compared and demonstrate that placing “smaller” strings at the top of a list puts the list in alphabetical order.

The routine accepts two strings from the keyboard. It displays both of the strings and the ASCII codes for each character of each string. Then it announces which string is smaller. Enter

```
NEW
10 PRINT CHR$(147):REM COMPARE STRINGS
20 PRINT TAB(11)“COMPARE STRINGS”:PRINT:
   PRINT“ USE UP TO 6 ”;
30 PRINT“KEYSTROKES PER STRING”:PRINT
40 INPUT“ENTER STRING 1”;S1$:PRINT
50 INPUT“ENTER STRING 2”;S2$:PRINT
60 REM -----
```

To test that routine, run it and enter two strings using six keystrokes per string. Then enter

```
PRINT S1$, S2$
```

The two strings should be displayed. List the program and enter

```
90 REM DISPLAY STRING 1
100 PRINT CHR$(147)
110 PRINT“STRING 1:”;
120 FOR I=1 TO 6
130 PRINT TAB(6+(5*I)) MID$(S1$,I,1);
140 NEXT I
150 REM -----
```

This routine takes string 1 apart to put spaces between each character. Line 110 prints a label, STRING 1:, at the left side of the screen. The semicolon holds the cursor on the same screen row.

Line 120 sets up a loop, using I as the counter, to separate the six-keystroke string into six individual characters.

Line 130 begins with a TAB() statement that will cause the first character in the string to be tabbed over and puts spaces between the characters. When I is 1, the tab is 6 + (5*1), which is 11

spaces. The first character is printed 11 columns over from the left. When I is 2, the tab is 16, and so forth. Each character is tabbed over by 5 spaces from the preceding character. That puts 4 spaces between characters.

The second statement on line 130 takes the string apart, using a MID\$() statement. When I is 1, it gets MID\$(1,1), which is the first character in the string. When I is 2, it gets MID\$(2,1), which is the second character, and so forth.

To test this routine, run the program and enter two six-character strings. The first one should be displayed.

Then the routine displays the ASCII code for each character, directly below that character. List the program and enter

```
190 REM DISPLAY ASCII, STRING 1
200 PRINT
210 PRINT" ASCII:";
220 FOR I=1 TO 6
230 L$=MID$(S1$,I,1):REM GET CHARACTER
240 PRINT TAB(5+(5*I)) ASC(L$+CHR$(0));:
    REM DISPLAY ASCII FOR CHARACTER
250 NEXT I
260 REM -----
```

Line 230 is used to simplify the rest of the routine. It gets the characters, one at a time, from string 1 and names them L\$. The following line then operates on L\$, rather than a complicated MID\$() expression representing L\$.

Line 240 tabs over in a similar way to the preceding routine. The fixed increment is 5, rather than 6, because this routine prints numbers. Numbers carry a leading space. As you will see, this causes the ASCII code numbers to line up vertically with the characters displayed directly above.

The second statement on line 240 gets the ASCII code for each character, L\$. The instructions for this routine ask the user to input up to six characters. Fewer than six is OK. But, this routine takes ASC() of six characters, whether the string is that long or not.

If fewer than six characters are entered, this routine will attempt to take ASC() of a character that doesn't exist in the string—a null. The ASC() function produces an error message if it attempts to evaluate ASC(*null*). The correct ASCII code for a null is zero, but ASC() won't get it.

As you remember from an earlier chapter, the solution to this problem is to take ASC(L\$+CHR\$(0)). If L\$=" ", the function takes ASC(CHR\$(0)), which is 0. That causes the ASC() function to give the "right answer" when it is evaluating a null.

By that method, line 240 will print the ASCII code for each character in the string and zeros for characters not input—if the string has less than six characters.

To test this routine, run the program and input two strings of only four characters. The first string should be displayed. Below each character, the ASCII code for that character is displayed. The next two routines are almost duplicates of the preceding two. They display string 2 and the ASCII codes for the characters in string 2. Enter

```
290 REM DISPLAY STRING 2
300 PRINT:PRINT
310 PRINT"STRING 2:";
320 FOR I=1 TO 6
330 PRINT TAB(6+(5*I)) MID$(S2$,I,1);
340 NEXT I
350 REM -----
390 REM DISPLAY ASCII, STRING 2
```

(Program continued on next page.)

Sorting Routines

```
400 PRINT
410 PRINT" ASCII:";
420 FOR I=1 TO 6
430 L$=MID$(S2$,I,1):REM GET CHARACTER
440 PRINT TAB(5+(5*I)) ASC(L$+CHR$(0));
    REM DISPLAY ASCII FOR CHARACTER
450 NEXT I
460 REM -----
```

Test by running the program and entering two strings. Both should be displayed. The ASCII code for each character is displayed below each character. By looking at the ASCII code numbers, and remembering how strings are compared, you can see which of the two strings should be declared smaller.

That doesn't show which of them the computer will actually declare smaller because this routine has not yet actually used a relational operator to compare the strings. Do that by entering

```
490 REM COMPARE STRINGS
500 PRINT:PRINT:PRINT
510 IF S1$<S2$ THEN PRINT
    S1$" IS SMALLER THAN "S2$
520 IF S2$<S1$ THEN PRINT
    S2$" IS SMALLER THAN "S1$
530 REM -----
590 REM DO AGAIN?
600 PRINT:PRINT:PRINT"DO AGAIN? Y/N?"
610 GET A$:IF A$=""GOTO 610
620 IF A$="Y"GOTO 10
```

Run the program several times. Enter a variety of strings including some with letters only, letters and numbers, and numbers only. Try strings of different lengths.

Notice that the computer has no difficulty making comparisons of strings of different lengths. At lines 240 and 440, it was necessary to concatenate CHR\$(0) with L\$ to prevent a program error when L\$ is operated on by ASC() and L\$ is a null. That trick is needed only by the ASC() function. It is not needed when the computer compares two strings using relational operators.

COMPARISONS OF UPPER-AND-LOWER-CASE CHARACTERS

If you are sorting lists of strings, you need to know what happens if some of the characters are capital letters and some lower case. So far, this demonstration has shown that upper-case letters are correctly sorted.

There are two things to consider: the ASCII code number produced by a keystroke and the character displayed by that keystroke. The character displayed for any keystroke is determined by the character set that has been selected for display.

What actually affects sorting is the ASCII codes produced by keystrokes, not what is displayed. Because relational operators compare variables stored in memory, it is the strings in memory that determine the result of a comparison.

In this program, lines 510 and 520 use the relational operators < and > to compare strings S1\$ and S2\$ *as they are stored in memory*.

What this program reports as the result of a comparison is the way a sorting routine will actually compare those strings. Any conclusions you make from this demonstration program are valid for working programs that sort strings.

The following tests will demonstrate three things:

- 1) The ASCII code produced by a keystroke and stored in memory is determined only by whether or not the keyboard is shifted or unshifted when the keystroke is made.
- 2) If upper-and-lower-case is used at the keyboard to enter strings, such as Mary and COD, the resulting strings *may not be sorted correctly* by a sorting routine.
- 3) What is displayed on the screen has nothing to do with the way strings are compared and sorted.

Demonstration—Set the keyboard so only capital letters are displayed. Run the program. Enter these two strings: AAAAAA and BBBBBB. Notice the ASCII codes and the comparison. A is code 65 and B is code 66. AAAAAA is smaller than BBBBBB.

Press N to end the program. Enter

```
PRINT S1$,S2$
```

The strings in memory are exactly what you entered: AAAAAA and BBBBBB.

Keep the computer in the upper-case mode and run the program again. For string 1, press SHIFT and press key A three times. Then release the SHIFT key and press A three more times. Displayed are three spade symbols followed by three letter A symbols.

Make a similar entry using the B key. Displayed are three vertical bars followed by three letter B symbols.

When the ASCII codes are displayed, notice that SHIFT-A produces code 193 and unshifted A produces code 65. SHIFT-B produces code 194 and unshifted B produces code 66.

By the rule that smaller codes identify “smaller” strings, string 1 and string 2 were sorted correctly—even though sorting graphics symbols doesn’t mean much.

Press N to end the program. Enter

```
PRINT S1$,S2$
```

The strings in memory are exactly what you entered: three graphics symbols followed by three capital letters.

Now press Commodore-SHIFT to change the display to upper- and lower-case letters. Run the program again. Enter AAAaaa and BBBbbb. Notice that the codes are exactly the same as before: shifted A produces 193; unshifted A produces 65; shifted B produces 194; unshifted B produces 66. The codes stored in memory are determined by *whether or not the keyboard is shifted*.

What is displayed doesn’t matter. You can change the display by pressing Commodore-SHIFT, or from a program. When the display changes, the codes don’t.

Were AAAaaa and BBBbbb sorted correctly? According to the codes, they were. Most alphabetical lists, such as dictionaries and indexes, disregard capitalization when placing words in alphabetical order. I think that is a correct sort.

While continuing to use upper and lower case, enter apples and bakers. That appears to be a correct sort.

Now enter Apples and bakers. That appears to be an incorrect sort. It would place bakers nearer the top of an alphabetical list than Apples.

What to Do About That—There are two ways to prevent incorrect alphabetical sorts. One is to use capital letters only. Many mailing-list programs print address labels. If you prefer upper and lower case for address labels, using just upper-case letters is not a solution.

The other method is not to use relational operators to compare strings using upper-and-lower-case letters. Instead, take each string apart and obtain code numbers for each character. The demonstration program you have been running does that to display individual codes.

Then write program lines to assign the same code number to each letter, whether capitalized or not. Then do comparisons and sorting. The result will be an alphabetized list disregarding capitalization.

A practical method is to convert all letter codes to the same range of 26 numbers, so they can be compared. To see what the conversion should be, run the demonstration program using the upper-and-lower-case mode. For string 1, enter azAZ. For string 2, just press RETURN.

Sorting Routines

The codes for string 1 show one range for a to z and another range of numbers for A to Z. In the upper-and-lower-case mode, codes for lower-case letters are from 65 to 90. In that mode, codes for the upper case letters are from 193 to 218. Each of the higher codes is larger than the equivalent lower-code number by 128. I suggest converting all of the higher-code numbers into their lower equivalents.

To do that, first test each code to see if it is in the range of 65-90 or 193-218. If not, it isn't a letter. If so, and it is in the range of 193-218, subtract 128 to convert it to the lower code number.

When that is done, any letter A produces code 65. Any letter B produces code 66, and so forth. Any letter B will be placed nearer the bottom of an alphabetical list than any letter A, no matter if it is upper case or lower case.

But to sort the strings, you will have to compare individual code numbers after the high codes are converted to low codes—not the two strings.

If you compare the two strings, *bakers* is smaller than *Apples*. If you compare *converted* code numbers, *b*=66, *B*=66, *a*=65 and *A*=65. *Apples* is smaller than *bakers*.

SORTING MAILING LISTS

Computers are commonly used to store and manage mailing lists. Usually, the *master* list is stored in some useful order.

Other lists are derived from the master list by selecting and sorting. A list may be sorted first by states. Within each state, the list may be sorted by cities. Within each city, the list may be sorted by zip code.

That gives the user precise control over a mailing. It can be to any state, any city, or even any zip code.

To provide this flexibility, names and addresses are usually stored as *fields*. A field is a separate item of data. Using the symbol / to mark the fields, a name and address may be stored like this:

J. W. Smith/1432 Poplar/Sumtown/Ohio/12345

That item has five fields. To put a mailing list in some desired order, a sort is performed on the list. Because there are several fields, the list can be *sorted by fields*. Sometimes this is called *sorting by keys*.

A good way to do that is to put the mailing list into an array in memory, with each field as a separate array element. Suppose there are 1000 entries on the list. That would require an array with 1000 rows and 5 columns. It would be dimensioned `ARRAY$(1000,5)`.

If J. W. Smith is the 99th name on that list, the 99th row in the array would look like this:

<code>ARRAY\$(99,1)</code>	<code>ARRAY\$(99,2)</code>	<code>ARRAY\$(99,3)</code>	<code>ARRAY\$(99,4)</code>	<code>ARRAY\$(99,5)</code>
J. W. Smith	1432 Poplar	Sumtown	Ohio	12345

Suppose you need to select only addresses in Ohio and you want the cities in alphabetical order. That requires two "looks" at the list.

Begin by forming a new list that includes all addresses in Ohio, but rejects all other states. To do that, you would make a new array to get Ohio addresses from the master list, which holds addresses from all states.

Use a loop to examine column 4 in each row of the array. If that array item is not *Ohio*, skip to the next item. If it is *Ohio*, put that row of the master array into the new array.

That decision is based on the content of column 4 in the array. But, if a name is selected, all 5 columns from the master list must be moved into the new list.

These program lines illustrate that. Don't enter them.

```
100 C=1:REM COUNTER FOR NEW LIST
110 FOR N=1 TO 1000:REM 1000 ENTRIES
110 IF ML$(N,4)<>"Ohio" GOTO 200:REM MASTER LIST
120 FOR FIELD=1 TO 5
```

(Program continued on next page.)

```

130 NL$(C,FIELD)=ML$(N,FIELD):REM COPY ROW IN ML$ TO ROW IN NL$
140 NEXT FIELD
150 C=C+1
200 NEXT N

```

Line 100 provides C as a counter to assemble a new array named NL\$(), which means *new list*. Line 110 sets up a loop to scan the master list. The master list is the array ML\$(). It has 1000 entries. The loop will run 1000 times, using N as the counter. N means *entry*. This loop will look at 1000 entries on the master list.

Line 110 tests column 4 of each row of the master list array ML\$(). It looks at ML\$(N,4). When N is 1, it looks at ML\$(1,4), and so forth. If that array element is not *Ohio*, the program jumps ahead to line 200. The program doesn't accept entries that are not in Ohio.

If line 110 does not execute, the entry is in Ohio. Line 120 sets up a loop to copy the 5 fields from ML\$() to NL\$(). The loop counter is FIELD. It will range from 1 to 5.

Line 130 says, NL\$(C,FIELD)=ML\$(N,FIELD). Suppose this is the first entry in the master list that is in Ohio. Suppose it is item 99 in the master list. The loop counter for ML\$() will be at 99.

That item in ML\$() will become the first item in the new list, NL\$(). The variable C provides row numbers to assemble the array NL\$(). Its value will be 1 if this is the first name to be copied from ML\$() to NL\$().

As the loop at 110 runs from 1 to 5, the five fields from ML\$() will be copied into NL\$() like this:

```

NL$(1,1)=ML$(99,1)
NL$(1,2)=ML$(99,2)
NL$(1,3)=ML$(99,3)
NL$(1,4)=ML$(99,4)
NL$(1,5)=ML$(99,5)

```

Line 140 is the bottom of the FIELD loop. Line 150 increments C so it can be used to fill the next row of elements in NL\$(). Line 200 is the bottom of the N loop that scans the master list.

When that program segment runs, a new list of names and addresses in Ohio is selected from the master list. Items in the new list will be in the same relative order that they were in the master list.

Suppose that order is not correct for the purpose of NL\$(). You must sort the new list into the correct order, by examining the correct field in the array NL\$().

Even though sorting is done by examining only one field in NL\$(), when an interchange is made, all fields in each row must be interchanged. Do that with a loop operation similar to the one just illustrated.

When NL\$() has been sorted into the desired order, it is ready to use as a mailing list.

OTHER SORTING METHODS

The bubble sort and the insert sort are *linear sorting* methods because they start at one end of the list and work to the other. Linear sorts are OK for short lists. With long lists, they are slow.

A category of faster sorts is called *sublist* sorts. They are beyond the scope of this book, but I will give you the general idea.

A sublist sort begins with an unsorted list. On the first pass, the unsorted list is divided into two lists by a sorting rule. The rule is that all items with values greater than some arbitrary value, X, go into one of the two new lists. All items with values smaller than X go into the other. X can be whatever is appropriate.

Then each of the two lists is divided into two more lists, following a similar rule. Now, there are four lists. The program controls the range of values in each list.

By continuing that procedure, there is finally a large number of short lists. They are then combined into one list again, in correct order.

The advantage of sublist sorts is that the computer makes fewer passes by each item. The total sorting time is less. They are more complicated to program, but usually worth the effort when large lists are to be sorted.

MEMORY SPACE

Arrays take a lot of space in memory. If you create too many, the program will stop and tell you that it is out of memory. The Commodore 64 doesn't have a way to clear one array out of memory, when you don't need it any more, without clearing everything from memory. A CLR statement clears everything except the program.

A way to avoid memory-space problems when sorting large lists or creating a lot of arrays is to use magnetic tape or a disk as storage.

The method is to read all or part of an array to be sorted from tape or disk. Sort it and put it back on tape or disk at a different location. These are called *tape-to-tape* or *disk-to-disk* sorts. Using tape and disk storage is discussed in the next chapter.

CHEAPSORT

Sorting methods sometimes have colorful names. One sublist method is called *heapsort*.

Once, I was trying to devise a way to put some items in numerical order. I wrote some elaborate routines that didn't work. After I made them even more complicated, they still wouldn't work. Finally, I had an idea that was absurdly simple.

I created an array to hold the sorted list. Then I put item 7 into array location 7, item 12 into location 12 and so forth. When all items were plugged into the array, the list was sorted.

I won't say how many hours it took me to rise to that level of simplicity. I was so pleased with my "invention" that I called it *cheapsort*. It doesn't take much time or effort to do it.

REVIEW

The important part of this chapter is the bubble sort. The insert method is a simplified bubble sort with only one item to be sorted into a list.

Even though the idea of a bubble sort is simple, you won't know how to do a bubble sort until you write a routine that works. I suggest that you review the bubble-sort demonstration in this chapter.

For practice, write a routine that accepts 10 names from the keyboard and puts them in an array in the order that they are entered. Use names such as BILL and MARY, in upper case. When the array is full, sort it in alphabetical order and display the sorted list.

Here is a routine that does that. After you have written yours, and made it work, look over mine. There is usually more than one way to write a program.

AN ALPHABETICAL BUBBLE SORT ROUTINE

```

10 PRINT CHR$(147):REM ALPHA SORT
20 FOR I=1 TO 10
30 PRINT"ENTER NAME NUMBER "I
40 INPUT N$(I)
50 NEXT
60 PRINT CHR$(147)
70 REM -----
90 REM SORT N$( )
100 F=0:REM DONE FLAG
110 FOR P=1 TO 9:REM PASSES
120 H$=N$(P)
130 IF N$(P+1)<N$(P) THEN
    N$(P)=N$(P+1):N$(P+1)=H$:F=F+1
140 NEXT
150 IF F<>0 GOTO 100:REM TIL DONE
160 REM -----
190 REM DISPLAY LIST
200 PRINT"SORTED LIST:":PRINT
210 FOR I=1 TO 10
220 PRINT N$(I)
230 NEXT

```

14

Disk And Cassette Files

When you turn off your computer, whatever is stored in random-access memory disappears. If you have a program and associated data in memory, it disappears.

The purpose of disk and cassette machines is to save programs and data permanently. Send the program and data that you want to save to the storage device *before* turning off the computer. When stored on disk or tape, programs and data can be played back into the computer to be used again.

Programs are stored in *program files*. Data created or used by a program, such as an array full of information, is stored in a *data file*.

Later, you may wish to run that program again. To do that, you would first play the program from disk or tape back into the computer memory. Then, run the program.

When the program needs information stored in a data file, such as a sorted list, the *program* must command the storage device to send that data into memory. Of course, you must write the program so it will do that. When in memory, the data can be used by the program.

These data exchanges are between the memory in the computer and the storage device. Before anything can be sent to disk or tape, it must first be placed in memory. Then it can flow from memory to the storage device. When a program or data is played back from disk or tape, it flows into the computer memory.

Putting a program on disk or tape is called *saving* it. Playing it back from tape or disk into the computer memory is called *loading* it. Playing it back does not remove it from the tape or disk.

Programs are saved and loaded by the computer operator. That is done in the immediate mode by entering commands at the keyboard. The operator tells the computer which program to save or load.

Data files are managed by a program. They are put on disk or tape and retrieved from disk or tape by program statements while the program is running.

INPUT-OUTPUT

The computer is a set of electronic circuits inside the keyboard housing. All of the other hardware is not part of the computer—it is attached to the computer. The computer regards everything else as an *input-output* device. The computer *communicates* with these devices.

Input-output operations cause data to flow back and forth between the computer and other devices. The word *input* means that data flows into the computer from a device. The word *output* means that data flows out of the computer to a device.

You already know how to do input-output operations. You can input data from the keyboard and output it to the screen. The BASIC words INPUT and GET are used to input data from the keyboard and put it in memory with a variable name. The word PRINT is used to output data from memory to the screen.

The same BASIC words are used for input-output with other devices. When used for input-output with a disk drive or cassette recorder, the symbol # is added to those BASIC words. They become PRINT #, INPUT # and GET #. There is no space between the word and the # symbol.

The procedure to input and output data to other devices is similar to inputting from the keyboard and outputting to the screen.

Opening a Device—Several steps are necessary for communication with any device. The device must be identified, and a path for communications must be provided between the computer and the device. That is called *opening* the device. It is similar to calling Aunt Myrtle on the telephone.

Each device has a device number that serves as its name. For example, the device number for the disk drive is 8.

After communication is established between the computer and a device, then information can be sent in one or both directions. The computer controls that. It tells the device when to talk and when to listen. It also tells the device what is being sent to it and what the device should send to the computer.

The computer is designed so the input-output operations done most often are set up automatically. What you do most often is input from the keyboard and output to the screen. In the immediate mode, the computer always has communication open to the keyboard and the screen.

To do other types of input/output, you must learn how to tell the computer to open and manage communications with input/output devices.

DEVICE NUMBERS

Each device, except the keyboard, has an identification number. When the computer isn't doing something else, it automatically monitors the keyboard.

The screen is device 3. You haven't used a device number for the screen because BASIC does it for you. A PRINT statement automatically opens communication with device 3 even though you are not aware of that step. Device numbers are:

DEVICE	DEVICE NUMBER
Cassette	1
Modem	2
Screen	3
Printer	4
Disk	8

TAPE OR DISK?

To get the most benefit from your computer, you should have a way to store programs and data. A disk drive is the best way because disk storage is faster and more versatile. However, the Commodore cassette recorder works well, costs less and is simpler to use. It is very slow.

Chapter 15 is about using a disk drive. Chapter 16 is about using a cassette recorder. Read the chapter that applies to your equipment—or both if you have both.

15 Using A Disk Drive

This chapter applies specifically to the VIC-1541 Single Drive Floppy Disk unit. Other disk drives that work with a Commodore 64 are similar. This information will generally apply to any disk-storage device that can be used.

If you haven't already done so, turn off the computer and connect the disk drive to the computer as described in the disk-drive user's manual. Read the instructions at the beginning of Chapter 2 about handling disks, saving and loading programs.

If the disk drive is connected, but not turned on, turn off the computer. Then turn the disk drive on. Then turn on the computer again. The turn-on procedure for the computer requires that everything else be turned on *before* you turn on the computer.

A disk drive stores information magnetically on a floppy disk. Putting information on a disk is called *writing to the disk*. Getting information from a disk is called *reading the disk*. Reading a disk does not change what is on it.

Information is stored on a disk in circular tracks, as shown in the accompanying drawing. Each track is subdivided into *sectors*. A sector holds 256 characters. Commodore literature usually refers to a sector as a *block*.

Everything stored on a disk is in a file. A file may occupy several sectors on several tracks on the disk. Each file has a name.

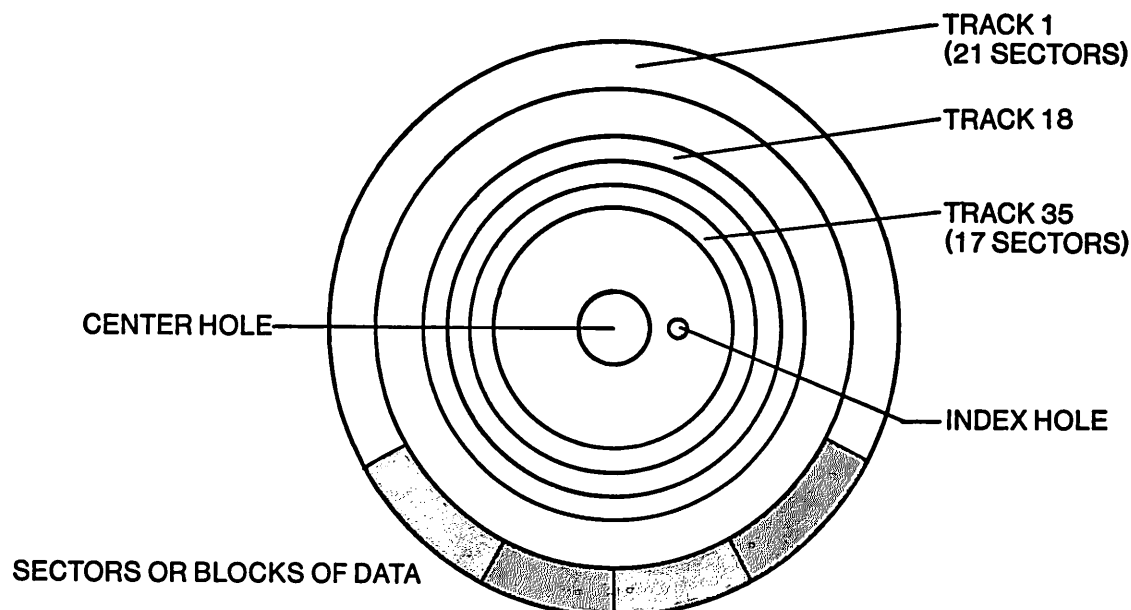
When sending information to a disk, you must specify a filename for that information. To retrieve the information, you must specify the filename that you want to read from the disk back into the computer.

A single item of information in a file, such as a customer's name, is called a *record*. One file may hold many records. One disk can hold up to 144 files.

Disk operations are controlled by a program called the *Disk Operating System (DOS)*. The DOS is stored in read-only memory in the disk-drive unit. Commands that relate to disk operations are sent to the DOS program in the disk drive. There, they are translated into instructions that operate the disk drive.

DOS COMMANDS

The accompanying table is a glossary of DOS commands discussed in this book. They are part of BASIC, and they are used the same way as other BASIC words. Spend a minute or two to become familiar with them.



A floppy disk stores magnetically on 35 circular tracks. Track 35 is the innermost.

Each track is divided into *sectors*, also called *blocks*, which hold 256 bytes. The first two bytes in each sector are used for control information; 254 are available to store data.

Because track 1 is near the outside of the disk, it is physically longer than track 35, which is nearer to the disk center. Therefore, tracks nearer the outside edge of the disk have more sectors than those nearer the center. Track 1 is divided into 21 sectors. Track 35 is divided into 17 sectors.

There are a total of 683 sectors on a disk. Track 18 is used to store the directory of files on the disk, and their locations. This leaves 664 sectors, or blocks, available to store user data.

DOS COMMAND WHAT IT DOES

SAVE	Copies a program from memory and puts it on disk.
LOAD	Reads a program file from disk and puts it in memory.
OPEN	Establishes communication between computer and disk drive. Names the file that will be used.
CLOSE	Ends communication between computer and disk for the file named in the OPEN statement. Closes that file.
PRINT #	Sends data to an open data file. Similar to the action of PRINT with the display.
INPUT #	Gets data from an open data file. Provides a variable name for that data in computer memory. Similar to INPUT as used with the keyboard.
GET #	Gets one or more characters from an open data file. Provides a variable name. Similar to GET as used with the keyboard.
NEW	Prepares a new disk to store data. Also prepares a previously used disk so new data can be stored on it.
COPY	Copies a disk file into another file with a different name on the same disk. With two disk drives, can copy from one disk to another disk.
RENAME	Changes the name of an existing file on disk.
SCRATCH	Erases a file from the disk.
INITIALIZE	Returns the disk drive to normal "power-up" condition. Similar to RUN STOP-RESTORE on the computer.
VALIDATE	Reorganizes files on a disk. Collects unusable or abandoned space and makes it available for use.
VERIFY	Compares the program in memory to a program with the same name on disk to see if they are identical.

TYPES OF FILES

Several types of files are used with BASIC programs.

Program Files—These are the simplest to use. There are only two commands: SAVE puts the program in memory onto the disk, using a filename that identifies the program. LOAD brings the program from disk back into memory, using the same filename.

Sequential Files—This file type is used to store data as a continuous sequence of records—like beads on a string. The only way to get data that is in the middle of the file is to start reading it from the beginning.

Relative Files—This file type is used to store data. You can read or write any record in the file without starting at the beginning. This type of file is like a file drawer filled with individual records in folders. You can open the file and select any record, without having to start at the beginning and work your way to the desired record.

Each record has a number. The disk operating system keeps track of the location of each record on the disk. You specify the record number that you want to read or write.

Random Files—This file type is used to store data. To use this method, you control where segments of the file are placed on the disk. Any segment can be read or written to, but you have to know where it is on the disk. This file type is not discussed in this book. It is discussed in the user's manual packaged with the disk drive.

PREPARING A NEW DISK

The best way to become familiar with disk operations is by the “cookbook” method. I will give you some statements to enter, without explaining everything about them. If you enter them correctly, you will get tasty results. As this chapter progresses, I will explain things in greater detail. By the end, you will understand what you did at the beginning.

Before you can use a disk to store programs and data, it must be prepared by a procedure usually called *formatting* a disk. Formatting erases everything on the disk. Then it puts magnetic “markers” on it, to be used by the disk drive to locate and identify locations where data can be recorded. This chapter includes a program to format disks automatically.

Because this procedure *must be done* with brand-new disks, the command is NEW. This command can also be used to format a previously used disk. It erases everything on the disk, so it can be used again.

If you have been using a disk to store demonstration programs from earlier chapters, continue using it for the programs in this chapter. If you don't have a formatted disk, prepare one using the procedure shown at the beginning of Chapter 2.

OPENING COMMUNICATION WITH THE DISK DRIVE

There are 16 *channels*, along which data may flow to or from the disk-drive unit. They are numbered 0-15.

When you open communication with the disk drive, there are two kinds of information that may be exchanged. One is data being written to the disk or read from the disk. This kind of information flows through a *data* channel. Data channels are channels 2-14.

The other kind of information that flows between computer and disk drive is DOS commands being sent to the disk drive and error information about the disk drive being sent back to the computer. Commands and error information flow back and forth through the *command* channel, which is channel 15.

When an OPEN statement is used to set up communication with the disk drive, the channel is selected by number. A statement that you will use often is OPEN 15,8,15. It establishes communication with the disk drive through the command channel. When that channel is open, DOS commands can be sent to the disk drive through channel 15. Commands relate to operation of the disk drive. Also, DOS error messages can be received from the disk drive through channel 15.

An OPEN statement opens communications with the disk drive. The statement OPEN 15,8,15 opens the command channel, 15. It has three parts following the word OPEN. The first part is the number 15. That is called the *file number*. I will discuss it later. The next part is the *device number*. It

specifies the device being opened for communications. That is number 8, which is the disk drive. The last part is the *channel number*. Command channel 15 is opened at the disk drive by that statement.

CHECKING FOR DOS ERRORS

The DOS program in the disk drive has a set of error messages similar to those displayed by BASIC when program errors occur. I will refer to these as DOS errors. If there is a DOS error, the red light on the disk drive blinks on and off.

DOS errors are reported to the computer through the command channel. If a DOS error occurs, this program will read the error and display it. Enter

```
NEW
8990 REM GET ERROR FROM DOS
9000 OPEN 15,8,15:REM OPEN COMMAND CHAN
9010 INPUT # 15,CODE,MESSAGE$,TRACK,SECT
9020 PRINT CODE,MESSAGE$,TRACK,SECT
```

Line 9000 opens the command channel to the disk drive, so the computer can find out what the error is. At line 9010, INPUT # 15 uses file number 15 to receive four data items from the disk drive, through channel 15.

Those four items describe the error. CODE is the DOS error code number. MESSAGE\$ is a short error message to tell you what went wrong. It relates only to disk operations and is not the same as a BASIC error message to report program errors.

If the error was in reading or writing to the disk, TRACK is the track number on the disk where the error occurred. SECT is the sector or block number in that track. Unless you know a lot about programming, track and sector numbers are not very useful.

Line 9020 displays the error information by printing all four items on the screen. Run the program. Unless there is a DOS error to report, the display shows

```
0 OK 0 0
```

The first zero is the error code. Code 0 means that there is no error. The word OK is the error message, meaning there is no error. The track and sector numbers are both zero, because there was no error on the disk.

Sending an Invalid Command to the Disk—To see a DOS error, let's send a bad command to the disk drive. PRINT # 15, is used to send commands to the disk. The space between PRINT# and the number 15 is optional. A comma after the number 15 is required in a complete statement.

The command to be sent is XXX, which has no meaning to DOS. Enter

```
PRINT # 15,XXX
```

The red light should be blinking on the disk drive. There is a DOS error. To see the error message from DOS, run the program in memory. The display shows

```
31 SYNTAX ERROR 0
0
```

The error code is 31. A syntax error is being reported by DOS. A list of DOS error messages and their meanings is in Appendix D.

Notice that the red light is no longer blinking. When you read the DOS error, it is cleared. To verify that, run the program again. Now it says everything is OK.

To use that program to read DOS error messages, it must already be in memory when the error occurs. That's why the line numbers are so high. If you are writing and testing a program that may produce disk errors, you can load this routine first. Then use line numbers below 9000 for the program. Then if a DOS error occurs, enter RUN 9000 to see what it is.

DOS may report errors with numbers smaller than 20. If so, they are to be disregarded. This routine will display error codes smaller than 20. Later, you will see a routine that doesn't.

A better way to read DOS errors when programming may be on the demonstration disk that was packaged with your disk drive. I will discuss it later.

PROGRAM FILES

Programs that help the programmer do things, such as read DOS errors, are usually called *utility* programs. To put the utility program now in memory onto the disk, you must save it. The format for a SAVE command to the disk drive is

SAVE *filename*,8

The filename is any name that you choose for that program. It must not be the name of another file on that disk. It may have up to 16 characters. Spaces between characters are OK, but they become part of the character count.

The filename is not part of the program. It is just the name of the file holding the program. The filename must be a string. If you type the name, put it in quotation marks. For example, if a filename is FINANCE, you can save it by this statement: SAVE "FINANCE", 8

You can use a string variable to represent the filename, if the filename is in memory as that variable. After executing the statement N\$="FINANCE", you can use N\$ in a SAVE statement instead of "FINANCE". The statement would be SAVE N\$, 8.

The device number is the number at the end. The device number tells the computer where to send the program to be stored. The device number for the disk drive is 8.

Let's use READ ERROR as the filename for the program in memory that reads DOS errors. Enter

SAVE"READ ERROR", 8

The display shows that the DOS is SAVING READ ERROR. The red light on the disk drive glows continuously while the program is being read onto the disk.

If the red light did not blink to indicate an error, the program is probably safely on the disk. List the program in memory. It is still there. Saving a program to disk does not remove it from memory.

DIRECTORY

The disk operating system maintains a directory of all filenames on the disk. A disk may have up to 144 files and 144 filenames in the directory.

The directory is itself a file. You can load it like a program and list it like a program. When you list it, you see the name of the disk and the names of all files on the disk. The name of the disk is the name used when the disk was formatted.

To make it easy to type the name of the directory file, it is just the symbol \$. Enter

LOAD"\$",8

When you see the READY prompt, enter

LIST

The first row at the top of the directory identifies the disk. The number at the left has no significance. The number at the extreme right, such as 2A, is the version of the disk operating system being used. The disk name and ID is "DEMO" 01, which is what you entered when you formatted this disk if you followed the instructions in Chapter 2.

Below the disk identification, all files on the disk are listed. The number at the left is the number of blocks or sectors used by that file. The filename is shown in quotation marks. At the right is a three-letter code showing the type of file that it is. PRG means *program* file.

A formatted disk has 664 blocks or sectors free to store files. This program used 1 block. If it is the only file on the disk, there are 663 left. Each sector holds 256 characters, or bytes. To find out how many bytes an empty formatted disk has free, enter

PRINT 664*256

LOADING PROGRAMS FROM THE DIRECTORY

To see the filenames on a disk, load and list the directory as you just did. To load any program shown, enter a load statement, such as LOAD "READ ERROR",8

Here is a trick. Use the filename already on the screen. Move the cursor up to the row that says "READ ERROR". Overtyping the number at left with the word LOAD. Move the cursor to the right

of "READ ERROR" and type , 8. Move the cursor farther to the right to erase PRG. Now you have a complete LOAD statement on that row, with nothing else except the cursor. It says LOAD "READ ERROR", 8. Press RETURN.

The red light should glow while the READ ERROR program is being loaded. Move the cursor down to a clear area on the screen and list the program. It should be exactly what was saved earlier. Run the program. The error report says everything is still OK.

CHANGING A PROGRAM

Often you will load a program from disk and then change it for some reason. Then you put the revised program back on disk, using the same filename. Please list the program in memory.

Its filename is READ ERROR, but you can't tell that by looking at the program. I normally put the filename at the beginning of a program as a remark. Then, before saving it to disk, I look at the beginning of the program to be sure I am using the correct filename. That prevents me from saving my program once as EXCELLENT and then saving it again later as SUPERB. Enter new line 8980 as shown below.

```
8980 REM FILENAME "READ ERROR"
8990 REM GET ERROR FROM DOS
9000 OPEN 15,8,15:REM OPEN COMMAND CHAN
9010 INPUT# 15,CODE,MESSAGE$,TRACK,SECT
9020 PRINT CODE,MESSAGE$,TRACK,SECT
```

Line 8980 means nothing in the program. It is just a reminder to you, so you will always file it the same way. Line 8990 is a remark to explain what the routine does. The routine begins at 9000, which is an easy number to remember. Now put the program back on disk by entering

```
SAVE "READ ERROR", 8
```

The red light is blinking. Luckily, you have a program in memory that will read the error. Run it. You got error number 63, FILE EXISTS.

SAVING AN EXISTING PROGRAM

The disk operating system protects files on the disk against the possibility that you may inadvertently use the same filename for two different files. If that happened, the second file would replace the first file with the same name and the first file would be lost.

To put a file on disk using a filename that already exists on the disk requires a special version of the SAVE command. It is called *save with replace*. Enter

```
SAVE "@0:READ ERROR", 8
```

The difference between the two SAVE commands is the @0: symbol used inside the quotation marks. It tells DOS that you really want to save that program and replace whatever is on the disk with the same filename.

The revised program file should have replaced the original file on the disk. To be sure, enter

```
LOAD"READ ERROR", 8
LIST
```

OPENING A FILE

Here is the complete format for an OPEN statement:

```
OPEN file n, device n, channel n, filespec
```

The symbol *n* means *number*. File number labels a *path* from the computer to device 8. What the file number is doesn't matter much because it is just a label the computer will refer to later. It will always use the same path, once the path is specified.

The file number must be in the range of 0-255. It is better to use a number from 0-128. It is best to use the same number as the channel number because that makes OPEN statements easier to read and write.

The second item is the device number. The disk drive is device 8.

The third item is the channel number. There are data channels 2-14 and a command channel 15 into the disk drive.

The last item in an OPEN statement is the *filespec*. If an OPEN statement is establishing or opening a *data* file on the disk, the name of that file is part of the filespec. Other information about the data file is also included, as you will see.

The command channel is treated as a file, but it is not necessary for you to give it a filespec. Therefore, the open statement omits the filespec. It is simply OPEN 15,8,15.

That statement says, *open communication using path 15 to device 8 and connect to channel 15*. Because the command channel serves two purposes, it is sometimes referred to by two names. When used to send commands to the disk drive, it is called the *command channel*. When used to receive error messages from the disk drive, it is sometimes called the *error channel*.

CLOSING FILES

A file number cannot be used twice in the same program at the same time. Because a file number is a label for a path to the disk, using the same label twice would confuse the computer.

When file number 15 is assigned to a path to device 8 and channel 15, that file-number label cannot be used for any other purpose until the assignment is canceled. It is canceled by a CLOSE statement. The format for a CLOSE statement is

CLOSE *n*

Symbol *n* means *number*. CLOSE 15 will disconnect the connection made by OPEN 15, *device n*, *channel n*. Then the file number can be used to label another path. I suggest that you reserve file number 15 for use with channel 15, just because it is easier to write and remember.

OPEN AND CLOSE RULES

Data files must be opened before you can use them, and closed before you end the program. If you end a program without closing the files, data may be lost.

If you attempt to open the same file twice, an error message results. To see the message, enter

OPEN 15,8,15

OPEN 15,8,15

The message is ?FILE OPEN ERROR. What kind of error message is that? The question mark identifies it as an error message produced by the BASIC interpreter. The red light on the disk drive is not blinking, so it isn't a DOS error.

Both BASIC and DOS monitor which files are open. If you are careful about opening and closing files, you won't have a problem. If you make a mistake, either BASIC or DOS may tell you about it.

You can close a file that is already closed, and there is no objection. Enter

CLOSE 15

CLOSE 15

The part of an OPEN statement that *can not* be duplicated by another OPEN statement is the file number. It is OK to open the command channel, or a disk file, through two different file numbers at the same time. Enter

OPEN 5,8,15

OPEN 6,8,15

OPEN 7,8,15

Now you have the command channel opened through three different file numbers. Usually, there is no advantage in having a file opened through several different file numbers.

The maximum number of file numbers that can be open at the same time is five.

PRINT#

I mentioned in an earlier chapter that PRINT really means *do it*. PRINT# tells the disk drive to *do whatever follows in that statement*. It may be a command to DOS to open a file, or it may send data

to a file, just as PRINT sends data to the screen. The format for a PRINT# statement is

PRINT# *n*, *command*

Symbol *n* means *number*. The first thing that must follow PRINT# is the file number that labels the path to be used. The space between PRINT# and the file number is optional. The word *command* means whatever you tell the device to do.

The file number must be one that is already open. When the file number is assigned in an OPEN statement, it labels a path to some file in some device. Until it is closed, it is always the path to that file in that device. Because a file number can be assigned to only one path at a time, the file number itself then specifies the destination of a PRINT# command. It specifies which file is to be accessed.

For example, the statement OPEN 3, 8, 11 "FILESPEC" opens a data file named "FILESPEC" using file number 3 to device 8 through data channel 11. After that has been done, a statement using PRINT# 3 automatically goes to the same file in device 8 through channel 11.

GET#

This works the same way. GET# addresses the disk drive. The statement GET# 5, A\$ uses the path specified by file number 5. If that path leads to a file on the disk, it will get one or more characters from that file.

In the earlier demonstrations of GET, you saw that GET takes characters from the keyboard buffer. If a character is there, it gets the character. If a null is there, it gets a null. By running a loop ten times, a GET statement was used to empty the keyboard buffer.

If a loop using GET# is used to get characters from a disk file, the file is a reservoir of characters, similar to the keyboard buffer.

The statement GET# 5, A\$ accepts one character from a file opened using file number 5. The character is placed in memory as A\$.

The statement GET# 5, A\$(1), A\$(2), A\$(3) accepts three characters in sequence from the file. They are placed in memory in the array A\$() as items A\$(1), A\$(2) and A\$(3).

This loop will get and display 1000 characters from a disk file: FOR I=1 TO 1000: GET# 5, A\$: PRINT A\$: NEXT. That loop does not load the file into memory, it just displays it.

Notice that the variable name, A\$, is assigned to each character as it is received from the disk file. That variable name is then used to display the character. Then, the *same* variable name is used to receive the next character from the file. When that loop finishes running, there will be only one A\$ in memory. Its value will be the last value obtained from the file.

This loop will get 1000 characters from a disk file, load them into an array A\$() in memory, and then display them in a column: FOR I=1 TO 1000: GET# 5, A\$(I): PRINT A\$(I): NEXT. To display the characters in horizontal rows on the screen, put a semicolon at the end of the PRINT statement.

INPUT#

INPUT# addresses the disk drive. It is similar to INPUT, used to address the keyboard. INPUT# must be followed by a file number to designate the path, device and channel number. The statement INPUT# 3, A\$ follows the path specified by file number 3. If that path leads to a file on the disk, it will get characters from that file until it encounters a RETURN keystroke.

An INPUT# statement is used in the READ ERROR program in memory. Please list the program. Line 9000 opens file number 15 as the path to device 8 and selects command channel 15 at that device. The command channel is treated as a file even though it is not given a filename. What it holds is error information about the disk drive.

Line 9010 uses file number 15 to address drive 8 and channel 15. It provides a list of four variable names to be input from the disk drive to the computer. When received from the disk drive, those four values are stored in memory with those variable names. Line 9020 then displays the error report by displaying the four variables.

CHANGING A FILENAME

Occasionally, you will have a file on disk and want to change its filename in the disk directory for some reason.

I picked a bad filename for the program in memory. READ ERROR is DOS error message number 20. It means that there was a read error on the disk. Having a filename that is the same as an error message may be confusing. It is best to avoid names that may be confusing. That applies to variable names, filenames, or any kind of names.

DOS has a RENAME command that lets you change filenames on disk. The format is
PRINT # 15, "RENAME0:newname=oldname"

It is used after the statement OPEN 15,8,15 has been executed, so file number 15 is open and leads to the command channel.

Rename this program file ERROR REPORT instead of READ ERROR. Then, when there is a DOS error, running the ERROR REPORT routine will report the details. This command will produce an error. Enter

```
PRINT # 15, "RENAME0:ERROR REPORT=READ ERROR"
```

That produces an error message. The file is not open. Which file is not open? File number 15. Several pages back, you closed it twice.

Suppose you don't know if a file number is open or closed, and you want to send something to that file. You can't send something to a file if it is closed—even if the file is the command channel. Sending data to a closed file produces an error message.

Maybe it's a good idea to open it just to be sure. If it is already open, that produces an error message. That is not a good idea.

A way to avoid an error message, if you are not sure whether a file number is open or not, is to close the file and then open it. Closing a file that is already closed is OK. Try it this way. Enter

```
CLOSE 15
```

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

```
PRINT # 15, "RENAME0:ERROR REPORT=READ ERROR"
```

Load and list the directory to be sure the filename was changed to ERROR REPORT.

Now you are in big trouble. The directory says that the filename is ERROR REPORT but the program itself has a remark at the beginning that says its filename is READ ERROR. The program isn't in memory any more. It was *replaced* by loading the directory. The program is on disk with a mistake in it!

This is what you must do: Load the program from disk, using the correct filename. Change the remark to show the correct filename. Then put the program back on disk using the save and replace command. I'll wait while you do that. The accompanying format table may be helpful.

Please spend a few minutes looking over the format table. Several of the commands can be abbreviated by using only the initial letter.

The NEW command has two uses. If a disk is already formatted, you can use a NEW command to erase only the directory. That frees the entire disk for re-use without formatting it again, which is faster. The old files will be gone, and new files may be written to the disk.

The COPY command copies a file onto the same disk, using a different name. It can be used to combine up to four files into a new filename by listing the four old filenames.

The SCRATCH command erases a file from the disk. Actually, it just removes the filename from the directory. The DOS can no longer find it so it is effectively erased. A new file will be written over it on the disk.

INITIALIZE and VALIDATE are not often used. They were defined earlier in this chapter. Except for these two commands, I think you can learn to use the others just by trying them.

I suggest that you spend a half hour or so using all of the commands on the format table except INITIALIZE and VALIDATE. The main problem will probably be getting the statements typed correctly.

You can't harm the computer or disk drive even if you do something wrong. The most you can lose is the error-report program, which is easy to enter again. If you get into big trouble, format the disk again and start over.

FORMAT FOR DISK-RELATED COMMANDS (Symbol n means number)

Command	Format
LOAD	LOAD "filename",8
SAVE	SAVE "filename",8 (puts new filename on disk) SAVE "@0:filename",8 (save with replace)
OPEN	OPEN file n,8,channel n,"filespec" (opens file) OPEN 15,8,15 (opens command channel to disk drive)
CLOSE	CLOSE file n
PRINT #	PRINT # file n,command or print list
INPUT #	INPUT # file n,command or input list
GET #	GET # file n,command or input list
NEW	PRINT # file n,"NEW0:disk name,ID" PRINT # file n,"NO:disk name,ID" (ID is any two characters) PRINT # file n,"NEW0:disk name" (erases directory of formatted disk)
COPY	PRINT # 15,"COPY0:newfile=0:oldfile" (or) PRINT # 15,"C0:newfile=0:oldfile" PRINT # 15,"C0:newfile=0:f1,0:f2,0:f3,0:f4" (copies up to four oldfiles, f1-f4, into newfile)
RENAME	PRINT # 15,"RENAME0:newname=oldname" (or) PRINT # 15,"R0:newname=oldname"
SCRATCH	PRINT # 15,"SCRATCH0:filename" (or) PRINT # 15,"S0:filename"
INITIALIZE	PRINT # 15,"INITIALIZE" (or) PRINT # 15,"I"
VALIDATE	PRINT # 15,"VALIDATE" (or) PRINT # 15,"V"
VERIFY	VERIFY "filename",8

Begin by copying the ERROR REPORT file onto the same disk using several different names, such as TEST1 and TEST2. Then rename them, combine them by copying, scratch them, load them and list them to see what happened.

Use PRINT# only to send commands to the disk drive using the command channel. Its use to put data on the disk will be discussed later in this chapter.

You can't do much with INPUT# yet except review how it is used in the error-report program. Try inputting less than four items in the error report. Try using different variable names to receive them. The program now uses long variable names for clarity. What the computer actually uses is two-character variable names. They are CO, ME\$, TR and SE.

You can experiment with the GET# statement by using it in the error report program instead of INPUT\$.

PRINT#, INPUT# and GET# will all be used again later, in the discussion of opening and using disk files.

A FORMAT UTILITY

Complaints that casual users have about most disk operating systems is that the commands are complicated, the formats are difficult to remember, and it's difficult to avoid typos when entering the commands. Some people are frustrated by DOS commands.

A trend is to "wrap" the DOS inside another program that is easier to use. The next demonstration program is an example. It formats a disk by asking the user simple questions. It asks if the answers are correct and allows the user to fix typos.

It has error traps to prevent invalid inputs. It decides whether to format the entire disk or just erase the directory. Then the program issues the DOS commands. The user doesn't have to remember or type them.

This program is typical of "user-friendly" programs. It uses a lot of lines to do what can be accomplished by only two DOS statements. When you run it, you may agree that it makes formatting disks seem a lot easier.

The program should explain itself. Read it. Notice that it uses variable names with the NEW command. The variable names are entered at the beginning of the program. At line 310, three strings are used with the NEW command. One of them is a comma. In line 420, CHR\$(34) displays a quotation mark. CHR\$(13) is a carriage return on the screen.

Enter this program and fix typos, if necessary, so it works. To test it, use a new disk or one that you don't mind erasing. Then save this program to disk.

```
NEW
5 REM FILENAME "FORMAT"
10 PRINT CHR$(147):PRINT TAB(12)
   "FORMAT UTILITY":PRINT:PRINT
20 PRINT:PRINT"REMOVE UTILITY DISK."
30 PRINT"INSERT DISK TO BE FORMATTED."
40 PRINT:PRINT"THEN PRESS SPACE BAR."
50 GET A$:IF A$="" GOTO 50
55 PRINT CHR$(147):PRINT"WARNING":PRINT:
   PRINT"DISK TO BE FORMATTED"
60 PRINT"WILL BE ERASED. OK? Y/N?"
65 GET A$:IF A$="" GOTO 65
70 IF A$="N" THEN PRINT CHR$(147):PRINT:
   PRINT"REMOVE DISK.":GOTO 430: REM END
75 IF A$<>"Y" GOTO 65:REM TRAP
80 REM -----
90 REM GET DISK INFO
100 PRINT CHR$(147)
110 PRINT"ENTER DISK NAME USING 1-16 CHA
   RACTERS":INPUT N$:PRINT
120 PRINT"ENTER DISK ID USING 2 CHARACTE
   RS":INPUT ID$:PRINT:PRINT
130 PRINT"ARE NAME AND ID OK? Y/N?"
140 GET A$:IF A$="" GOTO 140
145 IF A$="Y" THEN PRINT:PRINT:GOTO 160
150 IF A$="N" GOTO 100
155 GOTO 140:REM TRAP
160 PRINT"HAS DISK BEEN USED BEFORE?
   Y/N?":PRINT
```

(Program continued on next page.)

```

165 GET A$:IF A$=" "GOTO 165
170 IF A$="N" GOTO 300:REM FORMAT
175 IF A$<>"Y"GOTO 165:REM TRAP
180 REM -----
190 REM ERASE DIRECTORY ONLY
200 GOSUB 500:REM OPEN COMMAND CHANNEL
210 PRINT# 15,"NEW0:"N$:REM DISKNAME
220 GOTO 400:REM END ROUTINE
230 REM -----
290 REM FORMAT DISK
300 GOSUB 500:REM OPEN COMMAND CHANNEL
310 PRINT# 15,"NEW0:"N$","ID$:
    REM DISKNAME,ID
320 REM -----
390 REM END ROUTINE
400 CLOSE 15
410 PRINT CHR$(147)
420 PRINT"DISK "CHR$(34)N$","ID$CHR$(34)
    CHR$(13)CHR$(13)"IS READY TO USE."
430 PRINT"END OF FORMAT UTILITY."
440 PRINT:PRINT"TO FORMAT ANOTHER DISK"
450 PRINT"RUN THIS PROGRAM AGAIN.":END
460 REM -----
490 REM SUBR TO OPEN COMMAND CHANNEL
    AND DISPLAY PROGRAM STATUS
500 OPEN 15,8,15
510 PRINT CHR$(147):FOR I=1 TO 10:PRINT:
    NEXT:PRINT TAB(11)"FORMATTING DISK"
520 RETURN

```

SEQUENTIAL FILES

Sequential files are used by programs to store and retrieve data on a disk. These files are managed by statements in the program.

The main problems in using sequential files are getting them opened and closed correctly, and managing data separators. Many of the demonstration programs in this chapter will cause error messages. This is deliberate. It is done to show you some common problems and their solutions.

OPENING SEQUENTIAL FILES

The format to open a sequential file is

OPEN *file n*, *device n*, *channel n*, "*filespec*"

The symbol *n* means *number*. You have been using the first three items in that statement. The file number labels a path to the file. The device number identifies the device. The channel number affects the type of communication with the device.

When opening a disk file to send data, you cannot use channel 15 because that is reserved for the command channel. Channels 0 and 1 are reserved by DOS to load and save programs.

For data files on disk, you can use channels 2-14. These are the data channels. It doesn't matter which of them you use.

It is convenient to use the same file number and channel number, if the file number is not being

used by another OPEN statement. The statement OPEN 4,8,4, "filespec" opens a file using file number 4 as the path to device 8 and channel 4 to carry data to or from the file.

The word *filespec* means all of the specifications required to establish a file. A filespec includes the filename plus other information. For a sequential file, the complete format is

OPEN *file n, device n, channel n, "O : filename,type,mode"*

Everything in the quotation marks at the end is the filespec for a sequential file. It begins with O: as do some other disk commands. The DOS program is used with other systems. The O: symbol has meaning in other applications, but not with the Commodore 64. Nevertheless, it must be used when DOS expects it to be used.

The filename is a string of 1 to 16 characters. The *type* is the type of file being opened. There are four types:

FILE TYPE	FILE-TYPE DESIGNATION
PRG	Program file
REL	Relative file
SEQ	Sequential file
USR	User file

All three letters of the file-type indicator may be used, or just the first initial. Its function is to tell the DOS what type of file to open. The file type for each file on a disk is shown in the directory for that disk.

The *mode* specification is either *Read* or *Write*. The letters *R* or *W* can be used. A sequential file is opened either to write data into it or read data from it. Both cannot be done simultaneously. If you have a sequential file open in the W mode to write data into it, you must close it and then open it again in the R mode to read it.

OPENING AN EXISTING SEQUENTIAL FILE

The operating system protects sequential files on the disk against the possibility that you may inadvertently use the same file name twice. If that happened, original contents of the file could be lost or altered by accident.

This protection is used only in the Write mode, because that is the only mode that can be used to change the content of a sequential file. To use the Write mode and open a sequential file that *already exists* on the disk, use the prefix @O: ahead of the filename, instead of the O: prefix.

WHICH CHANNEL OPENS A FILE?

An OPEN statement is a DOS command to open a file on the disk. It doesn't transfer data to the disk. Opening a file is a preliminary step, so data can then be sent to the file.

All DOS commands must use channel 15. Therefore, channel 15 must be open before you can issue a DOS command to open or close a file.

At this moment, you may not be sure whether channel 15 is open or not. When you are not sure, close it and then open it. Closing a file already closed does not produce an error message. Enter

CLOSE 15

OPEN 15,8,15

Now, you know that channel 15 is open and DOS is ready to accept DOS commands.

CREATING A SEQUENTIAL FILE ON DISK

A file is *created* when it exists on the disk and appears in the directory of files for that disk. If a file does not exist on the disk, it can be created by an OPEN statement using the filespec—which includes the filename.

The operating system will not allow you to open a sequential file in the Read mode if it does not already exist on the disk. Opening an empty file to read it would be pointless because there is nothing in it to read.

The only way to *create* a sequential file on the disk is to open it in the Write mode. It will then exist on the disk and in the directory, even if you don't write anything in it.

The ERROR REPORT routine will be useful for this demonstration. If it is not in memory, please load it now.

Here are some examples of opening sequential files. This one will cause an error. Enter

```
OPEN 4,8,4,"0:SEQTEST,SEQ,READ"
```

The red light on the disk drive is blinking. To see the DOS error message, run the error report program. The display shows DOS error message 62, FILE NOT FOUND. What that OPEN statement attempted to do was use file number 4 to open a file named SEQTEST. The file type is SEQ and the mode is READ.

DOS won't open a sequential file in the Read mode if it does not exist on the disk. Open it in the Write mode. Enter the identical statement, except change READ to WRITE.

```
OPEN 4,8,4,"0:SEQTEST,SEQ,WRITE"
```

That works OK—if the command channel is open. A sequential file named SEQTEST,SEQ has been created and exists on the disk even though it is still empty.

Notice that the red light on the disk drive remains on without blinking. That means that a data file is open and DOS is ready to write to it or read from it. Opening the command channel does not turn on that light.

Try opening the same file again, while it is still open. Move the cursor up the same OPEN statement that opened it in the Write mode and press RETURN.

The BASIC interpreter detected that error. You can't open a file that is already open. Close the file by entering

```
CLOSE 4
```

The red light went out. Open the file again in the Write mode by entering

```
OPEN 4,8,4,"0:SEQTEST,SEQ,WRITE"
```

Now there is a DOS error. Run the error report program to see what it is. It is error 63, FILE EXISTS. The operating system is protecting the file named SEQTEST that is already on the disk, because you attempted to open it a second time in the Write mode. DOS wants assurance that you really want to do that. Enter

```
OPEN 4,8,4,"@0:SEQTEST,SEQ,WRITE"
```

The only difference is that the symbol 0: is changed to @0: to tell the DOS that you really mean it. The file is open.

What has been demonstrated is that you can't open a sequential file in the Read mode unless it already exists on disk. To create a sequential file, open it in the Write mode using the symbol 0:. When it has been created, the only way to open it again in the Write mode is to use the symbol @0:.

Now that the file exists, you should be able to open it in the Read mode—if you close it first. Enter

```
CLOSE 4
```

```
OPEN 4,8,4,"0:SEQTEST,SEQ,READ"
```

That works. If the file named SEQTEST had data in it, you could read the data. Close the file and open it in the Read mode again. Once a sequential file exists, you can open it in the Read mode as many times as you wish—if you close it before opening it again.

EXAMPLES OF COMMANDS FOR SEQUENTIAL FILES

To create a sequential file whether you write to it or not:

```
OPEN 4,8,4,"0:SEQTEST,SEQ,WRITE"
```

To write to an existing sequential file:

```
OPEN 4,8,4,"@0:SEQTEST,SEQ,WRITE"
```

Using A Disk Drive

To read an existing sequential file:

```
OPEN 4,8,4,"0:SEQTEST,SEQ,READ"
```

To close the file:

```
CLOSE 4
```

ERASING A SEQUENTIAL FILE

To erase a data file, using the SCRATCH command, you must open the command channel if it isn't already open. Enter

```
CLOSE 15
```

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

```
PRINT # 15,"SCRATCH 0:SEQTEST"
```

When the disk drive stops, load and list the directory. SEQTEST is not there.

SEQUENTIAL-FILE POINTER

The disk operating system uses an electronic pointer in sequential files, like a bookmark in a book. When there is nothing in the file, the pointer is at the beginning of the empty file.

As data is written to the file, the pointer moves along so it points to the end of the data in the file. That's how DOS knows where to put the next data item.

Each time a sequential file is opened in the Write mode, the pointer is moved to the beginning of the file, ready to write *an entirely new file*.

Each time a sequential file is opened in the Write mode, *all of the data that was in the file is lost*, because the pointer is moved to the beginning of the file. Then, if you send new data to the file, it overwrites what was there before.

Opening a sequential file in the Read mode does not lose data. The electronic pointer is moved to the beginning of the file and travels along the file as it is being read. Nothing is being written to the file, so no data is overwritten and destroyed.

That's why sequential files are opened either in the Write mode or the Read mode.

MANAGING A SEQUENTIAL FILE

If you have data in a closed sequential file, and want to add more data or change the file in any way, you cannot just open the file in the Write mode. That resets the pointer to the beginning of the file and loses the data.

To change data in a sequential file, you must first open it in the Read mode. Then read the entire file into memory. Close the file. Change the file in memory. Open the file in the Write mode, using the @0: symbol and write the entire file back to the disk.

ERROR CHECKING

It is good practice to monitor the disk drive operation for errors. That doesn't mean that errors happen often. But they will happen due to program or operator errors, disk failures, or maybe a problem in the disk drive itself. If an error occurs, the data may no longer be valid. You want to know that.

You know how to check the command channel for DOS error messages. There is another check that should also be made.

In addition to DOS error checking, the computer itself monitors data transfers between memory and disk. It maintains a variable called STATUS in memory at all times. If everything seems OK, the value of STATUS is zero. If something is wrong with the data, the value of STATUS is not zero. STATUS is usually written as a two-character variable name, ST.

The difference between command-channel error reports and STATUS errors is this: Command-channel errors are DOS errors. They relate to disk operations such as opening and closing files. STATUS relates to the data itself. For example, in a READ operation, if there is no data being read, the STATUS value is changed to indicate that. That may indicate that the disk drive is turned off or not functioning, or all data in the file has been read.

The rule is to check the command channel for errors after every OPEN statement. Check STATUS after every operation that moves data. These operations are PRINT#, INPUT# and GET#.

Checking the command channel is done in a subroutine similar to the error report routine that you have been using. Checking STATUS is done by a single line such as IF ST <> 0 THEN GOTO a subroutine. When ST=0, things are OK.

DETECTING THE END OF A SEQUENTIAL FILE

You can read all data items in a sequential file without knowing how many there are. When the file pointer reaches the end of file (eof) during a read operation, STATUS is set to the number 64. By checking STATUS after each read, you can detect eof.

If you read data out of a sequential file and execute fewer reads than there are data items, all items will not be read. If the program calls for more items than there are in the file, the STATUS value will be changed to 64 as soon as the last data has been read from the file. But, if the program continues to read the file, the program will not stop and no other error will result. What the program gets from continued reads past the end of a sequential file is a null at each read.

The only way to be sure you are getting data from the file is to check STATUS after each read.

WRITING AND THEN READING A SEQUENTIAL DATA FILE

This demo program shows all of the steps and procedures that have been discussed. The program is modular. There are individual program segments to open files, write data, close files, read data and do error checking.

When you have it entered and working, it will be used for the remaining demonstrations about sequential files, just by changing modules in the program.

In this program, the DOS-error subroutine uses two-character variable names ending in Z. This is to make them unique and easy to remember. If you use CODE, MESSAGE\$, TRACK and SECTOR for the variable names, as before, the computer will actually use CO, ME, TR and SE for those variables.

It is not a good idea to use variable names more than once in a program, except for loop counters in FOR-NEXT loops that run to completion. CO, ME, TR and SE are fairly common two-character names. For example, you may want to use CO for company and TR for transaction. Using CZ, MZ\$ and so forth in the subroutine frees those more common variable names for use in the main program.

Read the following program before you enter it. It should be self-explanatory. Notice that the data is entered as D\$ but it is received back from the disk as X\$ and displayed as X\$. That is to be sure you are seeing data from the disk, not D\$ from memory.

Line 1010 interprets any DOS error with an error code less than 20 as no error. Error codes smaller than 20 are not valid errors on the Commodore 64. At line 11000, STATUS is tested. If it is 64, it is not an error. It's just the end of the sequential file. The program ends with a display that says END OF DATA. If ST is not 64, then there is a data error of some kind. Line 11010 responds to that condition. Enter

```
NEW
10 REM WRITE, THEN READ SEQ FILE
20 PRINT CHR$(147)
30 REM -----
40 REM OPEN FILES
50 OPEN 15,8,15:REM COMMAND CHAN
60 OPEN 2,8,2,"@0:SEQTEST,SEQ,WRITE"
70 GOSUB 10000:REM DOS ERROR?
80 REM -----
```

(Program continued on next page.)

```
90 REM WRITE TO FILE
100 INPUT"ENTER DATA";D$
110 PRINT# 2,D$
120 IF ST<>0 GOTO 11000:REM STATUS
130 REM -----
190 REM CLOSE DATA FILE
200 CLOSE 2
210 GOSUB 10000:REM DOS ERROR?
220 REM -----
290 REM OPEN DATA FILE TO READ
300 OPEN 2,8,2,"0:SEQTEST,SEQ,READ"
310 GOSUB 10000:REM DOS ERROR?
320 REM -----
390 REM READ FILE
400 INPUT# 2,X$
410 PRINT:PRINT"DATA IS: "X$
420 IF ST<>0 GOTO 11000:REM STATUS
430 REM -----
490 REM CLOSE FILES
500 CLOSE 2
510 GOSUB 10000:REM DOS ERROR?
520 CLOSE 15
530 END
540 REM -----
9990 REM SUBR CHECK ERROR CHAN
10000 INPUT# 15,CZ,MZ$,TZ,SZ
10010 IF CZ<20 THEN RETURN:REM OK
10020 PRINT"DOS ERROR:"
10030 PRINT CZ,MZ$,TZ,SZ
10040 STOP
10050 REM -----
10990 REM STATUS NOT ZERO
11000 IF ST=64 THEN CLOSE 2:CLOSE 15:
    PRINT"END OF DATA":END
11010 PRINT"STATUS ERROR":STOP
```

Run the program and enter your name. Run again and enter enough Xs to fill the screen row where the cursor starts. And also fill the following screen row. On the third screen row, type your name. Then press RETURN.

Change lines 100 and 110 to enter D rather than D\$. Change lines 400 and 410 to display X, rather than X\$. Run the program and enter 12345. Notice that it prints with a leading space. Enter

PRINT 2*X

Run the program and enter - 12345. Notice that it prints with a minus sign. Run it again and enter your name. It won't accept strings when the program is written using numeric variables.

Change these program lines like this:

```
100 INPUT "ENTER DATA ";A$,B$,C$
110 PRINT # 2,A$,B$,C$
400 INPUT # 2, X$,Y$,Z$
410 PRINT X$ " "Y$ " "Z$
```

The commas in lines 100 and 400 are necessary for program syntax. Line 100 now accepts three data items from the keyboard. Line 400 retrieves three data items from the disk. Line 420 displays three data items separated by single spaces.

Line 110 is a mistake. It attempts to use commas as data separators in the PRINT# statement. The programmer is hoping that line 110 will place A\$, B\$ and C\$ on the disk as three separate records. He is wrong about that.

Commas in a PRINT# statement do the same thing as commas in a PRINT statement. They produce automatic tabs on the screen. The difference is that commas in a PRINT# statement go on the disk and don't affect the display until later, when the data is recovered from the disk and displayed. See what happens when that unfortunate person's ill-conceived PRINT# statement is executed.

Run the program. Enter your first name and press RETURN. The INPUT statement at line 100 displays ?? as a prompt for more data. It needs three items. Enter your middle name and press RETURN. Then enter your last name and press RETURN.

Notice the automatic tabs when the three names are displayed. The automatic tabs are caused by the commas in the PRINT# statement at line 110. They are going on the disk and affecting the display when they come off the disk. Enter

```
PRINT X$
PRINT Y$
PRINT Z$
```

As you can see, all three data items that were entered are in X\$, including automatic tabs. Y\$ and Z\$ are empty. Line 110 is making a *single record* out of A\$,B\$,C\$. The commas are not recognized as data separators. They are taken as part of a single data item in the disk file: A\$,B\$,C\$. Then, the commas cause automatic tabs when the data is retrieved and displayed.

This line will force line 110 to put the three data items on disk as three separate records. Enter

```
110 PRINT # 2, A$ "," B$ "," C$
```

Run it that way and enter three names. That works. The three items are displayed with single spaces between. The spaces are provided by line 410. Enter

```
PRINT X$
PRINT Y$
PRINT Z$
```

Now the data items are three separate records on the disk. There are three data separators that will work in PRINT# statements such as line 110. They are ",", ";" and CHR\$(13), which is a carriage return.

A better way to force individual data items to go on the disk as individual records is to PRINT# each one individually. A PRINT# statement automatically puts a carriage-return symbol at the end of each item. Enter these lines

```
110 PRINT # 2,A$
111 PRINT # 2,B$
112 PRINT # 2,C$
```

Run it that way and enter three data items. Use names, numbers or whatever you like. Instead of using separate PRINT# statements, you can use a loop. This is a very good way to transfer arrays from memory into a sequential file and get them back again. Delete lines 110, 111 and 112. Then, enter these lines

```
100 FOR I=1 TO 3:INPUT"ENTER DATA ITEM";D$(I):NEXT
105 PRINT CHR$(147)
110 FOR I=1 TO 3:PRINT# 2,D$(I):NEXT
```

Line 100 accepts 3 strings from the keyboard and puts them in an array in memory. Line 110 reads the array from memory into the sequential file on disk. Run the program. It works the same as before. If you have a lot of data items, using loops is the best way.

List -500. Line 400 accepts the three data items from the disk, but it does not put them back into an array. It just puts them in memory as individual string variables, X\$, Y\$ and Z\$. Usually, if the data goes onto the disk from an array, you will want it back into an array when it is retrieved from the disk. Running similar loops to read and display the items will do that.

If you know how many items are in the file, you can use a FOR-NEXT loop to read the file into an array or display it. If you don't know how many items there are, don't use a FOR-NEXT loop. Build a loop in the program that checks STATUS for an eof indication and quits inputting from the file when there is no more data. That method will be demonstrated in a minute.

Storing Numbers in the File—As you saw earlier, if you use numeric variables, you can put numbers into the file and retrieve them as numeric variables.

It is often more convenient to use string variable names, as the program is now written. That allows you to store both numbers and strings in the array in memory and also on the disk. You don't have to worry about which is which until you take the data back out of the file. Then, use VAL() to convert the numbers back into numbers.

Run the program and enter three numbers. Then enter

```
PRINT X$
PRINT VAL(X$)
```

I suggest using string variables in this routine.

Using GET#—The GET# statement is a very powerful way to retrieve data. It brings back the file, one character at a time. It gets all characters, including carriage returns. It gives you complete control of the data. You can write your own rules about data separators and carriage returns.

To see how it works, put a different READ FILE segment in the program. List -500 to get your bearings. Then enter

```
400 PRINT:PRINT"DATA IS: ";
410 GET# 2,X$
420 IF ASC(X$+CHR$(0))=13 THEN
    PRINT"[13]";:GOTO 430
425 PRINT X$;
430 IF ST<>0 GOTO 11000:REM STATUS
435 GOTO 410
```

This segment uses a loop that begins at line 420 and ends at 435. Notice that line 430 checks STATUS after each read from the disk file. This routine will run until ST<>0. When ST is not zero it will be 64 because the file ran out of data. When ST=64, the routine at 11000 ends the program. That reads every data item from the sequential file and stops when the last item has been read.

There is an equipment error that can cause STATUS to have a value other than zero. If the disk drive is not connected or not turned on, ST becomes -128. It is likely that you will be aware of that problem before attempting to read the disk because writing to a disk drive that is not present produces a DEVICE NOT PRESENT error. Therefore, it is customary to assume that STATUS is either 0 or 64.

Line 420 uses ASC() to get the ASCII code for each X\$ character as it is brought from the disk. Line 420 is looking for CHR\$(13). When it finds one, it prints the number 13 inside brackets. The brackets are just to call attention to the characters.

If line 420 finds CHR\$(13), it jumps ahead to line 430. That avoids line 425, which prints X\$. Printing CHR\$(13) causes a carriage return on the screen. As you will see, that would spoil the display.

If X\$ is not CHR\$(13), line 425 prints it. List the program and check your typing. Run it. For the three inputs, use 123, then 456, then 789. The display should show this stream of characters: 123[13]456[13]789[13]

Those characters are the entire sequential file, in the same order as they are in the file. The CHR\$(13) symbols were placed in the file automatically by the three PRINT# 2 statements that put the data items into the file.

This routine demonstrated using ST<>0 as a way to stop reading a sequential file with a GET# statement. Use the same method to stop reading with an INPUT# statement. It doesn't matter how you are reading the file. At eof, ST=64.

EXPERIMENT

Put a remark at the beginning that states both the program filename and the data file name. Save this program as SEQDEMO,SEQ so you have a copy of it.

Then, you can change it as much as you like. Spend some time running this program. You may wish to repeat some of the earlier demonstrations and try some experiments of your own.

GETTING A SEQUENTIAL FILE OPENED THE FIRST TIME

The demonstration program you have been running is backward for most practical applications. If a program uses a sequential data file, usually the first thing that happens when you run the program is that the file is read into memory from the disk. Then the data can be displayed or changed and put back on the disk. That procedure is read and then write. The demonstration program you have been running is write and then read.

The techniques are the same, but there can be a start-up problem the first time the program is run.

The DOS will not open a sequential file in the Read mode unless it exists on the disk. If the program has never run before, the file cannot exist. Therefore, if the first action in the program is to open the sequential file to read it, a DOS error results.

This program shows a solution to that problem. A file is opened through the command channel, so the command channel must be opened first. A RUN command closes all files, so a program must open what it needs to have open. Enter

```
NEW
10 REM FIRST RUN DEMO
20 OPEN 15,8,15:REM COMMAND CHAN
30 OPEN 2,8,2,"O:SEQFILE,SEQ,READ"
40 STOP
```

Run the program. The red light blinks to indicate a DOS error. You need to know the number of the DOS error message, but you don't have any way to get it. Entering NEW to start writing the program in memory erased the program lines that were already in memory.

Write a temporary routine beginning at line 100 to read the error channel and display the error. Then delete the temporary lines. You won't need them any more for this demonstration.

The error is DOS error 62, FILE NOT FOUND. You can't open a file in the Read mode unless it

exists on the disk. Getting the file open is a problem only one time—the first time the program runs. After that, the file will exist.

A solution is to use a DOS error-report procedure as a subroutine. Call the routine after line 30 executes, to see if there was an error. If so, solve the problem in the subroutine and return to the main program.

This problem can be solved simply by opening the file in the Write mode. There isn't any reason to display the error report if you fix the problem.

Change line 40 and add lines 50-9010 to the program.

```
10 REM FIRST RUN DEMO
20 OPEN 15,8,15:REM COMMAND CHAN
30 OPEN 2,8,2,"0:SEQFILE,SEQ,READ"
40 GOSUB 9000
50 CLOSE 2: STOP
8990 REM CHECK ERROR CHANNEL
9000 INPUT # 15,CZ,MZ$,TZ,SZ
9010 IF CZ=62 THEN OPEN
      2,8,2,"0:SEQFILE,SEQ,WRITE":RETURN
```

Line 30 will attempt to open the file. It will not succeed on the first run because the file doesn't exist. Line 40 calls the subroutine to check for a DOS error and solve the problem, if possible. Line 50 prevents executing the subroutine twice.

Run the program. You get an error message from BASIC. Both BASIC and DOS watch for errors. What the BASIC interpreter observed was that the program opened a file at line 30 and then opened it again at line 9010. BASIC is not aware of the problem at DOS. BASIC will not allow you to open the same file twice without closing it in between.

To make the error trap at line 9010 work, you must close the file to satisfy BASIC even though it never actually got opened on the disk. Then you can open it again. Change line 9010 like this:

```
9010 IF CZ=62 THEN CLOSE 2:OPEN
      2,8,2,"0:SEQFILE,SEQ,WRITE":RETURN
```

Run it. No error is indicated by BASIC or DOS. To be sure the file was created, load and list the directory. It should show a file on the disk named "SEQFILE" with a file type of SEQ.

As you know, opening a sequential file in the Write mode loses whatever data is in the file. The only time this routine opens the file in the Write mode is when the file does not already exist. If it doesn't exist, it can't have any data in it.

When you write a program that begins by reading a sequential file into memory, a routine like this is essential. Otherwise, you will never get the program to run the first time.

RECORD LENGTH AND NUMBER OF RECORDS

The maximum record length in a sequential file is the longest string of characters you can put into it between carriage returns. If you are entering data from the keyboard, the largest number of keystrokes that can be entered is 80, including the RETURN keystroke. You can assemble a longer string by concatenating two or more strings. The longest string that you can assemble is 255 characters, including the RETURN keystroke.

The maximum number of records is 32767. Usually, it is limited by space on the disk.

WHEN TO USE SEQUENTIAL FILES

To read a sequential file, you must start at the beginning. You can stop reading any time, with no harm to the file. To make changes to a sequential file, you must read it all into memory, change it, and then put it back on disk.

When you are storing data on disk and you will always need all of the data in memory when the program runs, a sequential file is a satisfactory method. If the program doesn't need all of the data

in the file at any one time, then it depends on the length of the data file. If it is short, then it can be moved quickly into memory, and it doesn't matter whether you need all of it there or not. If the file is long, time is wasted bringing data into memory that is not needed by the program.

RELATIVE FILES

Relative files consist of a group of individual records, each with a record number. Any record can be accessed by number, without having to read the file from the beginning.

You can set record length to any value up to 254 characters. The number of individual records depends on available space on the disk. With 254-character records, 720 records will fill a disk with a single relative file. With shorter record lengths, you can use more records.

This file type uses a pointer in a different way than sequential files. The pointer is controlled by the program. It may designate the beginning of any existing record or the beginning of a record about to be written. An essential part of programming with relative files is to keep track of the record-pointer location and control it.

Relative files are opened only one way. When open, they can be read or written to, as you choose. Opening a relative file does not set the pointer to the beginning and does not lose the content of the file.

The input-output commands are the same as sequential files. The main difference is in the OPEN statements and in addressing individual records by record number.

The symbol @O: has no meaning with relative files, and it is not used. The only way to erase an entire relative file is by using a SCRATCH statement. Then it can be replaced.

Individual records can be removed from the file by recording a single space in the records, such as PRINT # 3, " ". That will replace whatever was in the record.

CREATING RELATIVE FILES

A relative file is created when it exists on the disk and in the directory. It is created by a special OPEN statement used only for that purpose. The format is

OPEN *file n, device n, channel n, "filename,L," + CHR\$(len)*

File number, device number, channel number and filename have the same significance as with sequential files. The ,L, symbol after the filename establishes the file type as a relative file.

The last item in this OPEN statement is the record length. It is written in the parentheses of the CHR\$() statement. For example, CHR\$(177) sets a maximum record length of 177 keystrokes, including RETURN.

The computer will always make each record the specified length. If you don't send enough data to the record to use that many characters, the remainder of the record is automatically filled with nulls.

Here is an example of a statement that creates a relative file on the disk:

```
OPEN 3,8,3,"RELDemo,L,"+CHR$(177)
```

That opens a relative file named RELDEMO. The file number is 3 and the channel number is 3. The device number is 8, which is the disk drive. The record length is 177 keystrokes.

When listed in the directory, relative files are identified by REL. In an OPEN statement used to create a relative file, it is identified by the ,L, symbol.

OPENING RELATIVE FILES WHEN THEY EXIST

After being created by the OPEN statement just discussed, relative files may be opened again by a simpler statement. The format is

OPEN *file n, device n, channel n, "filename"*

The ,L, symbol and the record length spec are not required.

Here is an example that opens the file that was just created, assuming it has been closed.

```
OPEN 3,8,3,"RELDemo"
```

The filename is RELDEMO, not RELDEMO,L,.

INPUT-OUTPUT STATEMENTS

When a relative file has been opened, input-output statements use just the file number that was used to open the file. If a relative file was opened using file number 3, the statement `PRINT # 3, data` will address that file.

RECORD NUMBERS

Before a record can be read or written, the record pointer must be moved to that location on the disk. This is done by giving DOS the record number in a *position* statement that I haven't shown yet.

To position the record pointer, DOS must be given two separate numbers, which it combines into a single number used to position the pointer. The two separate numbers are called *high byte* and *low byte*.

Two-Byte Numbers—The mathematics of two-byte numbers is discussed in Chapter 18. To specify record numbers using two bytes does not require understanding the math, but you must understand the procedure.

A single byte in memory can store a number in the range of 0-255. That is the low byte. Because there may be more than that number of records, a second byte must be used to specify record numbers greater than 255. It is the high byte. The range of these two bytes is:

HIGH BYTE (0-1) LOW BYTE (0-255)

The two numbers work like the mileage counter in an automobile. Both begin at 0. Only the low byte changes for the first 255 numbers. For record 255, high byte is 0 and low byte is 255. For record 256, the high byte changes to 1 and the low byte counter starts over again at 0.

Here is a routine to calculate and display high and low bytes for any record number. Enter

```
NEW
5 REM CALC HI & LOW BYTES
10 PRINT CHR$(147)
20 INPUT "ENTER RECORD NUMBER";RN
30 PRINT CHR$(147)
40 PRINT:PRINT "RECORD NUMBER ="RN:PRINT
50 HI=INT(RN/256)
60 LO=RN-(HI*256)
70 PRINT "HI BYTE ="HI, "LO BYTE ="LO
80 PRINT:PRINT:PRINT:PRINT:GOTO 20
```

Run it and enter numbers from 0 to 5. Then enter numbers from 253 to 258. When the record number changes from 255 to 256, you can see the high byte change from 0 to 1, and the low byte start over again at 0.

For records up to 255, high byte is always 0 and low byte is the same as the record number. Break out of the program by pressing `RUN STOP-RESTORE`.

If you see high byte and low byte in a program, you can calculate the record number (RN) by the formula: $RN = (HI * 256) + LO$.

If you have not used two-byte numbers before, spend a few minutes becoming familiar with the procedures. Convert a record number, such as 293, into a two-byte number. Then convert the two-byte number back into a record number.

POSITIONING THE RECORD POINTER AND FIELD POINTER

Positioning the record pointer places it at the beginning of the selected record number on the disk. After that is done, the pointer may be *repositioned* to any desired location within that record. That allows writing and reading segments within a record. The segments are called *fields*.

Even though there is actually only one pointer, it is convenient to think of it as two operations. First position it as the record pointer—which moves it to the beginning of the record. That is done simply by stating the record number, using low byte and high byte. The DOS automatically moves the pointer to the beginning of the specified record number on the disk.

Then position it as the field pointer—which moves it to a location within the record. That is done separately, by specifying a field number. To begin writing or reading with the first character in the record, set the field number to 1. To begin with character 23, set the field number to 23.

Before reading or writing a record in a relative file, position the pointer using a *position statement* in this format:

```
PRINT # file n, "P" CHR$(channel n) CHR$(lo byte) CHR$(hi byte) CHR$(field)
```

The record number is always stated as a two-byte number, even if the high byte is zero. Low byte is stated before high byte.

After the two-byte record number is stated, then the field number is stated. The smallest value for the field number is 1. The largest value is the record length minus one. The last character in a record is CHR\$(13), which is placed there automatically by a PRINT# statement.

Positioning the record pointer is done through the command channel. If the command channel is opened as file number 15, the file number in the position statement will be 15.

The pointer is actually positioned in a relative file on the disk. To designate *which relative file*, specify the channel number used to open that file. Suppose it is channel 3. Here is an example:

```
OPEN 15,8,15:REM OPEN COMMAND CHAN
```

```
PRINT # 15,"P" CHR$(3) CHR$(123) CHR$(0) CHR$(1)
```

In that statement, consider P to mean *Position* the record pointer. CHR\$(3) means that the pointer is being set in the relative file opened using channel 3. The low byte is 123. High byte is 0. That positions the pointer at the beginning of record 123 in the file. Then CHR\$(1) at the very end sets the pointer to the first character in that record. It sets the field number. If that specification were CHR\$(17), reading or writing would begin with the 17th character in that record.

In a position statement, the *file number* is associated with the command channel. The *channel number* is associated with the relative file where the pointer is being positioned.

Examples— Assume that the command channel is opened as file number 15.

PRINT # 15,"P" CHR\$(3) CHR\$(1) CHR\$(0) CHR\$(1) positions the pointer to record 1, character 1 of the file opened using channel 3.

PRINT # 15,"P" CHR\$(4) CHR\$(255) CHR\$(0) CHR\$(6) positions the pointer to record 255, character 6 of the file using channel 4.

PRINT # 15,"P" CHR\$(6) CHR\$(0) CHR\$(1) CHR\$(99) positions the pointer to record 256, character 99 in channel 6.

PRINT # 15,"P" CHR\$(7) CHR\$(1) CHR\$(1) CHR\$(135) positions the pointer to record 257, character 135 in channel 7.

Two things are relative in a relative file. The record number is relative to the beginning of the file. Record 17 is the 17th record from the beginning of the file. Field numbers are relative to the beginning of the record. If the field number is 56, reading or writing begins with character 56 in that record.

OMITTING THE FIELD NUMBER

If you omit the field number from a position statement, the pointer remains at the beginning of the record. Reading or writing will begin with the first character in the file.

MANAGING RECORD NUMBERS

The ability to read and write individual records and individual fields within a record gives great flexibility in writing programs. But it also requires you to keep track of record numbers that have been used, and what is in those records.

If you are using more than one field in a record, you must also remember where the individual fields begin, so you can set the pointer exactly at the beginning of each field.

This sometimes requires you to use two files. One is a relative file with records in it. The other is an index of the relative file. If you want the record on Jones, look up the record number in the index file. Then use that record number to get the record on Jones from the relative file. The index file can be either sequential or relative.

If you are using more than one field in a record, you must remember where the individual fields begin, so you can set the pointer exactly at the beginning of each field. *Remember* means write it down—as a remark in the program.

When reading or writing to a relative file, you are entirely on your own. If you write to a record that already has data in it, the new data will replace the old data in that record. Be sure that is what you intend to do.

When you *write* a record, space on the disk is provided automatically for every smaller record number, whether the smaller numbers have been used or not. If you write to record 200, disk space is immediately allocated for every record up to number 200. *Reading* record 200 does not allocate space for all records with smaller numbers.

If you have not written to every record up to the highest record number in the file, you don't know what is in the records that your program has not written to. If the disk has been used before, those records will contain whatever was left in those locations by programs that used the disk before.

If you read a relative record that has not been written to, you get what programmers call *garbage*. It may be anything. Reading garbage into your computer can cause problems. For example, it may clear the screen.

EXAMPLES OF STATEMENTS USED WITH RELATIVE FILES

Open command channel:

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

Create relative file named RELDEMO using file number 3 and channel 3. Record length is 177 characters:

```
OPEN 3,8,3,"RELDEMO,L,"+CHR$(177)
```

Close RELDEMO.

```
CLOSE 3
```

Open RELDEMO again after it has been created and closed:

```
OPEN 3,8,3,"RELDEMO"
```

Position the record pointer in RELDEMO to record 123. Position statement uses file number of command channel and channel number of relative file. Omit field number, so reading or writing begins with first character in file.

```
PRINT # 15,"P" CHR$(3) CHR$(123) CHR$(0)
```

Position the record pointer in RELDEMO to record 123, field 22. Position statement uses file number of command channel and channel number of relative file. Field number is 22, so reading or writing begins with character 22 in file.

```
PRINT # 15,"P" CHR$(3) CHR$(123) CHR$(0) CHR$(22)
```

Put data in the file, using file number of relative file:

```
PRINT # 3,data
```

ERROR CHECKING WITH RELATIVE FILES

The DOS error-checking procedure is the same as with sequential files. There is one DOS error message used with relative files that you should pay particular attention to. It is error 50, RECORD NOT PRESENT. That error means that the record pointer has been positioned beyond the end of the relative file.

If the pointer is placed at the *first* record number beyond the end of the file, error 50 does not result. That allows adding to a file without causing error messages. For example, if records 1-15 are in the file, the pointer can be placed at record 16 without producing DOS error 50. Positioning it at record 17 will produce the error.

An error message is not produced if the program reads one record past the end of the file even

though that record has not been written to by the program. Reading two or more records past the end of file will cause that DOS error message.

DOS errors do not stop program execution, but you can detect them by reading the error channel after each input-output operation to a relative file and do whatever is appropriate.

Reading or Writing Beyond the Last Record—If a program positions the record pointer to a record number that is more than one record beyond the end of the relative file, DOS error 50 occurs. This error is produced by positioning the pointer, not later input-output statements. If your purpose is to increase the length of the file by writing a new record, ignore the error and write the record.

If you allow the program to *read* a record that has not been written to, and data is there, it will be garbage. If data is not there, a BASIC error occurs and stops the program. The error is *string too long*. It will be demonstrated.

Reading or Writing to Empty Records Within the Existing File—The record pointer may be set to any record *within* the file without producing DOS error 50, whether that record has previously been written to or not.

If the purpose is to write to it, do it. If the record has data, it will be replaced. If the record is empty, it will receive the data being written.

If the purpose is to read it, be sure the program has previously written to that record number. If data is there, but it was left there by earlier use of the disk, it will be garbage. If the record is empty, a BASIC *string-too-long* error occurs. This stops program execution.

Checking STATUS—Checking STATUS is not done to find the end of a relative file. If you check it, and it is not zero, there is a problem at the disk drive. It may be turned off or disconnected. The error is DEVICE NOT PRESENT, which will be reported by BASIC. There is usually no need to check STATUS with relative files.

USING A RELATIVE FILE

This demonstration will create a relative file, put records into it and read records from it. It can serve as a framework to write other programs. It is modular. By changing modules, you can cause it to work in a variety of ways. Enter

```

NEW
5 REM FILENAME "RELDemo"
10 PRINT CHR$(147)
20 OPEN 15,8,15:REM COMMAND CHAN
30 REM -----
90 REM CREATE REL FILE
100 OPEN 3,8,3,"RELFILE,L,"+CHR$(25)
110 GOSUB 10000:REM DOS ERROR?
120 CLOSE 3:REM TO DEMO REOPENING
130 GOSUB 10000:REM DOS ERROR?
140 REM -----
190 REM OPEN CLOSED FILE
200 OPEN 3,8,3,"RELFILE"
210 GOSUB 10000:REM DOS ERROR?
220 REM -----
290 REM POSITION POINTER TO WRITE
    NINE RECORDS.
300 FOR R=1 TO 9
310 PRINT # 15,"P"CHR$(3)CHR$(R)CHR$(0)
    CHR$(1)
320 GOSUB 10000:REM DOS ERROR?

```

(Program continued on next page.)

```
330 REM -----
390 REM WRITE RECORD NUMBER R
400 PRINT # 3,"CONTENT OF RECORD"R
410 NEXT R
420 REM -----
490 REM SELECT RECORD TO DISPLAY
500 PRINT CHR$(19):FOR J=1 TO 10:PRINT:
    NEXT J:REM LOCATE CURSOR
510 PRINT"ENTER RECORD NUMBER"CHR$(13)
    "    OR Q TO QUIT";:INPUT C$
520 PRINT CHR$(147):PRINT
530 IF C$="Q" THEN PRINT"END OF PROGRAM"
    :GOTO 800:REM CLOSE FILES & END
540 RN=VAL(C$):REM RECORD NUMBER
550 REM -----
590 REM POSITION POINTER TO READ
600 PRINT # 15,"P"CHR$(3)CHR$(RN)CHR$(0)
    CHR$(1)
610 GOSUB 10000:REM DOS ERROR?
620 IF CZ=50 THEN CZ=0:GOTO 500:
    REM PREVENT READ PAST END
630 REM -----
690 REM GET AND DISPLAY RECORD
700 D$=" ":INPUT # 3, D$
710 PRINT"THIS IS RECORD"RN":":PRINT
720 PRINT D$
730 GOTO 500
740 REM -----
790 REM CLOSE FILES & END
800 CLOSE 3
810 GOSUB 10000:REM DOS ERROR?
820 CLOSE 15
830 END
840 REM -----
9990 REM SUBR CHECK DOS ERROR
10000 INPUT # 15,CZ,MZ$,TZ,SZ
10010 IF CZ=50 THEN PRINT MZ$:RETURN
10020 IF CZ<20 THEN RETURN
10030 PRINT"DOS ERROR:"
10040 PRINT CZ,MZ$,TZ,SZ
10050 STOP
```

List the program in segments and check for typos. The remarks explain most of it. After each DOS operation—open, close or position the pointer—subroutine 10000 is called to check for DOS errors.

List 9990-. When this subroutine is called, line 10010 checks the error code, CZ, to see if it is 50—RECORD NOT PRESENT. If so, the subroutine displays the error message, MZ\$, and returns to the calling routine. The main program then jumps back to get another record number from the keyboard. If the error is not 50, but is a code less than 20, line 10020 returns to the calling routine without doing anything. Error codes below 20 are not valid errors.

If the error is not 50 and not below 20, the subroutine displays the DOS error report and stops the program.

List -400. The program creates a relative file named RELFILE and then closes it. It is not necessary to close it in order to use it. It is closed just to demonstrate the shorter OPEN statement used to reopen it after it has been created.

Line 200 opens the file again. Line 300 runs a loop with R as the counter to put records 1-9 in the file. Line 310 positions the record pointer for each of those record numbers, R. Line 320 checks for a DOS error caused by positioning the pointer. If error 50 or an error below 20 is detected, nothing is done. This routine is intended to write these records.

List -600. Line 400 puts the same data into each record, except for the record number. The routine at line 500 gets a record number from the keyboard for display.

List -700. Line 600 positions the pointer at the record number to be read. Line 610 checks for a DOS error. If the error is code 50, line 620 resets CZ to 0 in memory and then jumps back to line 500 to get another record number. This procedure is intended to prevent reading records beyond the end of the file. It will actually allow reading one record past the end of the file.

List the program. Line 700 clears D\$ in memory by setting it to a null. This is to be sure that what is displayed actually comes from the disk. Then it gets the content of the specified record number, using an INPUT# statement. Line 720 prints D\$. Line 730 jumps back to line 500 to allow another record number to be input from the keyboard.

Run the program. No RECORD NOT PRESENT errors are displayed because each new record being put into the file is one beyond the last existing record.

Display records 1-9. They should be the same as shown on line 400, except for the record number. Then press Q to quit. If the program works as it should, save it. If not, fix it and then save it.

Run the program again. Select record 0 for display. You get record 1. There is no record 0. Select record 11 for display. The subroutine finds error 50 and the read routine does not attempt to read that record. It jumps back to line 500 to get another number.

Records 1-9 have been placed in the file by this program. Enter number 10. There is no DOS error 50 to prevent reading that record even though it is beyond the file. You see whatever is on the disk. If you see a pi symbol, that code marks empty sectors on the disk. Enter Q to quit.

The way to avoid reading past the last record is to keep track of record numbers in the program and don't read records that are beyond the file. Another way is to test the content of the record. If it isn't what it should be, reject it.

DISK BUFFER

While running this program, you may notice that the disk drive seems to get records from the disk and put them on the disk when it isn't running.

The disk drive has a small memory, called a *buffer*. Each time data is read from the disk, the buffer is filled. This may be more data than the program asked for. But the next data requested by the program may be in the buffer. When the program asks for that data, the disk drive can deliver it from the buffer, without running the disk.

When data is sent to the disk, it is accumulated in the buffer. When the buffer is full, the data is put on disk. That reduces the number of times the disk drive must operate. To the computer, data in the buffer is exactly the same as data on disk. It can be read back into the computer memory when needed.

Some program operations cause the disk drive to empty the buffer onto the disk. One is to fill up the buffer. It automatically dumps its content to the disk. Another is closing files. If a program ends without closing files, data in the buffer may never be put on disk.

VARIATIONS

These program changes will show several ways of using relative files. List -500. Enter a new line 395:

```
395 IF R=5 THEN PRINT # 3," ":GOTO 410
```

On the fifth pass through the loop, line 395 will put a single blank space in record 5. It is necessary to put something in the record or an error results. The computer will fill up the rest of the record with nulls. Nothing will print in that record. Run it and display the records.

Change line 395 to a remark by inserting REM at the beginning. Then change line 400 to read

```
400 PRINT # 3,"123456789 RECORD"R
```

That will replace every record in the file with a slightly different record. Run it and display the records.

List -700. Change line 600 to read

```
600 PRINT # 15,"P"CHR$(3)CHR$(RN)CHR$(0)
      CHR$(5)
```

That sets the field number to 5 when positioning the pointer to read the records. The write pointer is not changed, so the records are the same as before. The read operation will begin with character 5 of each record. Run it and display the records. Each record is read from character 5 to the end of the record.

Now, remove REM from line 395 so it will execute. Run it again. Display records 1-5. At record 5, BASIC reports STRING TOO LONG ERROR IN 700.

List -700. The problem is that the field number causes the program to start reading each record at character 5. But, line 395 causes record 5 to have only one character—a space at the very beginning.

When the program starts reading record 5 at the fifth character, it finds a string of nulls. This provokes the error message. Sometimes, error messages don't describe the problem very well. Delete line 395 and run it again. Display the records.

The BASIC error message STRING TOO LONG results when the computer reads a relative record that is empty or at least empty from the pointer to the end of the record. Because it stops the program, it must be avoided. The way to avoid it is not reading an empty record.

EXPERIMENT

That demonstrates the fundamentals of relative files. What you need now is some experience setting them up and using them. There are some things that the demonstration program does not do. I suggest that you revise the program or write a new program so it does the following things.

These modifications may not be easy to do. Using relative files seems very complicated and difficult when you first try. The only way to learn to use relative files is to write some programs that work. It may require persistence and some hard thinking. When you have done it a few times, it becomes easier. It is worth the time and effort.

- 1) Use record numbers larger than 255. This requires adding a line to convert the record number to a two-byte number. Then plug the two-byte numbers, as variable names, into the position statements.
- 2) Create an array in memory. Read the array into a relative file. Then bring it back into an array.
- 3) Use several fields in each record. Write to each field. Then read and display each field.
- 4) Write a routine that allows bringing a record from disk, changing one or more of the fields, and then putting it back.

Figure out a way to keep track of each record that has been written. This is to prevent reading records that have not been written to by the program. A simple method is to create an array in memory dimensioned for the maximum number of records that will be in the file. When a record is written to, put a symbol such as Y in that element of the array.

That makes a sort of index to the file. You can put information about each record into the index, if you wish. Then, before reading a record, check the array element with the same number to see if the record has been written to. Before ending the program, save both the relative file and its index to disk, using two different filenames. When you run the program again, bring the index array into memory.

BACKUPS

You should have a backup copy on disk of every important program you have. If your only copy of a program is on a disk that fails, you have lost the program.

There are several ways to duplicate files on the same disk. If you have a program in memory, you can save it to disk twice, using two different names such as PROGRAM and PROGRAM BK. The symbol *BK* means *backup*.

You can duplicate both program and data files on the same disk by using the COPY command and different filenames. Copy the program to make a backup. Then copy the data file to make a backup for it.

That is pretty good protection. When a disk fails, usually most of it is still OK. If you have duplicate files, usually one of them can still be read.

Sometimes, a catastrophe happens. If your dog eats the disk, it doesn't matter if there were duplicate files on it. You may hope that he has duplicate indigestion.

The COPY command cannot be used to copy files to a different disk in the same drive. If you have two drives, you can copy files from one drive to the other.

You can duplicate *program* files on a different backup disk. Load the file into memory from one disk. Then change disks. Then save the program onto the second disk.

You cannot duplicate data files by that method because data files cannot be loaded by a LOAD command and saved by a SAVE command. LOAD and SAVE are used only with program files.

If you have a printer, you can write your programs so printing the data on paper is an option. That makes a reference copy of the data, if you ever need it. I think it is a good idea to print reference copies of both the program and the data.

CLOSING FILES

It is essential to close files before ending a program. DOS does its "bookkeeping" as each file is closed. That means it makes a record of the sectors or blocks on the disk used by that file. This information is recorded on the disk, in a Block Allocation Map (BAM) stored on track 18, with the disk directory. If the BAM is not brought up to date, DOS cannot locate the file on the disk. The data is lost.

Closing a data file also causes the disk buffer to put whatever it holds onto the disk during the close operation.

Because the command channel is used to open and close disk files, the command channel must be opened first and closed last. If a program closes the command channel with data files still open, DOS will close all of the data files. However, BASIC will not "know" that those files are closed because no BASIC statement closed them directly. If the program continues to run and send data to the files, it will not go onto the disk. There is no error message for that problem from BASIC or from DOS.

When a BASIC error causes a program to stop execution, all files are closed in the BASIC interpreter but not in DOS. If that happens, Commodore recommends using the INITIALIZE

command from the immediate mode to restore the disk drive to the power-up condition. Enter

OPEN 15,8,15,"INITIALIZE"

If you see an asterisk beside the type designation of a sequential file in the directory, that file was not closed before the program ended. The remedy is to run the program again, check and replace data in the file if necessary, then close the file and end the program.

DOS WILDCARDS

In **LOAD** commands, DOS will recognize symbols for portions of program filenames. A question mark can be used instead of a single character. The command **LOAD "T?ST"** will load the first program found using the letters T ST as the filename with any character in the space occupied by the question mark. For example, **TEST**.

An asterisk may be used to represent all of the remaining characters in a program filename. For example **LOAD "P*"** will load the first program found with a filename beginning with P.

The command **LOAD "*"** will load the last program loaded. If no program has been loaded, it will load the first program in the directory.

UTILITIES ON THE DISK-DRIVE DEMO DISK

With your disk-drive unit, you may have received a test/demo disk. If so, load and list the directory. If there is a **HOW TO USE** program shown, load and run it. It will explain the other programs on the disk.

Look for a program entitled **C-64 WEDGE**. If it is listed, load and run it. This program changes DOS to provide methods to view the directory and check the DOS error channel without losing a **BASIC** program that you may have in memory.

After running **C-64 WEDGE**, you can check for DOS errors by entering **@** or **>**. You can see the disk directory by entering **@\$** or **>\$**. Neither of these affects the program in memory. These are very useful commands when you are programming.

Also, you can load a disk program by using the **/** symbol to represent the word **LOAD**. Some other interesting and useful utility and demonstration programs are on that disk.

16 Using A Cassette Recorder

The Commodore 64 uses a special cassette-tape unit to store programs and data files. If your cassette unit is not connected to the computer, turn the computer off. Connect the cassette unit as shown in the instruction booklet for the recorder. Then turn the computer back on. If you haven't done so already, read the cassette instructions at the beginning of Chapter 2.

The cassette unit draws electrical power from the computer. It uses ordinary audio-tape cassettes. By turning the cassette over, you can use both sides of the tape. Storage capacity is determined by the length of the tape. Commodore recommends using cassettes with a maximum playing time of 30 minutes—15 minutes per side—to avoid mechanically overloading the cassette unit.

The computer is designed so tape commands are the simplest way to store and retrieve data. Because tape is played from beginning to end, data is stored on tape as a sequence of characters. This storage method is called *sequential*.

From the computer's point of view, sequential files must be recorded and read from the beginning. There is no way to enter a file at a point past the beginning and start reading or recording from there.

The cassette recorder doesn't know that. It will start recording or playing back at any location on the tape if you press the keys on the cassette machine to do that. When using a cassette recorder, you must set the tape to the correct location to do whatever the program in the computer requires.

Two kinds of files may be stored on tape. One is program files. These are copies of programs. They may be saved to tape or loaded from tape into the computer, so they can be run.

The other kind of file is a sequential data file. Data files are created and managed by statements in a program. They are used to store data, such as a list of names, used by the program.

The tape commands and their purposes are shown in the table on the next page.

USING THE CASSETTE UNIT

To avoid having the wrong keys depressed on the cassette unit, form the habit of pressing the STOP key at the beginning of each operation. That resets all keys. Then press the correct keys for the operation to be done.

Mechanically, it is better to press the STOP key at the end of each operation. That releases the record/play head from the tape and "relaxes" the machine. It doesn't harm the machine if you press STOP after each operation and before the next—just to be sure.

TAPE COMMAND	WHAT IT DOES
SAVE	Copies a program from memory and puts it on tape.
LOAD	Reads a program file from tape and puts it in memory.
OPEN	Establishes communication between computer and a data file on tape. Names data file.
CLOSE	Ends communication between computer and data file named in OPEN statement. Closes that file.
PRINT #	Sends data to an open data file. Similar to action of PRINT with the display.
INPUT #	Gets data from an open data file. Provides a variable name for that data in computer memory. Similar to INPUT as used with the keyboard.
GET #	Gets one or more characters from an open data file. Provides a variable name. Similar to GET as used with keyboard.
VERIFY	Compares program in memory to a program with the same name on tape to see if they are identical. Also used to find end of last program on tape.

It is usually a good idea to begin operations with the tape fully rewound. The cassette unit will start recording or playing back at any point along the length of the tape when you press PLAY to play or RECORD-PLAY to record. Unless you have rewound to the beginning, you may not know where the record/play head is located on the tape.

If you start recording in the middle of a tape, you may damage a file already on the tape. If you start playing, you will probably not play back a complete file. Unless stated otherwise, the following discussion assumes that you have rewound the tape.

SAVING AND LOADING PROGRAMS

Saving and loading programs is done in the immediate mode, by commands from the keyboard. Programs are usually saved with an identifying name called a *filename*.

The program to be saved must be in memory. The simplest way to put it on tape is to enter the command SAVE. If the tape is rewound, that puts the program at the beginning of the tape, *without a filename*.

To bring that program back into memory, rewind and enter LOAD. That command will load the first program on the tape, whether or not it has a filename.

Saving and loading without filenames is OK for temporary storage. If you are writing a program and want to take a break, that's a good way to save it. For permanent program storage, save programs with filenames. The format is

SAVE "*filename*"

The filename should name or describe the program. It must not be used by any other program on that tape. It may use up to 16 letters or numbers. Spaces are OK. Examples of filenames are FINANCE and MAIL LIST. The filename identifies that program on the tape, so you can later load it even if other programs are on the tape.

To load a program that has been filed with a filename, the format is

LOAD "*filename*"

After you have entered LOAD or SAVE commands, the computer will display operating instructions for the cassette unit. For example, after you enter a LOAD command, the display will show PRESS PLAY ON TAPE.

The computer does not control whether the tape is rewound or if the record/play head is at the right location on the tape. That is up to you. That's why it is good practice to begin most tape operations with the tape rewind.

VERIFYING SAVES

When you have saved a program to tape, you may want to be sure that it is on the tape without errors. The **VERIFY** command compares what is on tape to the program in memory. If they are not identical, the computer displays a message that says **VERIFY ERROR**. If that happens, save and verify again. To demonstrate that, enter this short program.

```
NEW
10 REM TEST SAVE AND VERIFY
20 PRINT "THIS IS A SAVE AND VERIFY TEST"
```

Run the program to be sure it works. Put a cassette into the cassette unit. Be sure nothing valuable is on that side of the tape. It will be overwritten by this program when it is saved. Rewind the tape to the beginning. If the tape counter is not set to 000, press the reset button beside the counter. Now the tape is rewound and the counter set to 000. Enter

SAVE

Follow the instructions on the screen: **PRESS RECORD & PLAY ON TAPE**. That means press both keys simultaneously. Watch the tape counter and the adjacent red **SAVE** light. As soon as the cassette unit starts saving the program, the red light glows.

The cassette unit begins by recording a standard audio signal that precedes a program. When the tape counter reaches 5, approximately, the red light blinks off and on. That signals the beginning of the actual program. At about 7 on the counter, the red light turns off and the cassette unit stops. The program should be on tape. To verify that, press the **STOP** key. Rewind the tape and press the **STOP** key again. Then enter

VERIFY

Follow the instructions on the screen. Press **PLAY**. Watch both the screen and the tape counter. The screen will become blank until the counter shows 5. Then the cassette unit stops temporarily and the screen display tells you that the file has been found.

Then the cassette unit runs again to play the program so it can be verified. When the counter shows 7, the cassette unit stops again and the screen shows **VERIFYING OK**, followed by the **READY** prompt symbol.

Verification was successful. To be sure, rewind and enter

LOAD

Follow the instructions on the screen. When the display shows **FOUND**, loading begins. When it shows **READY**, the program is loaded. Run it. List it. All of that was done without a filename, which is OK for temporary storage.

To load a program that is on tape, the record/play head must be positioned at the beginning of the program. When the program is at the beginning of the tape, that is easy to do. Just rewind before loading.

If the program is not at the beginning of the tape, the computer will find it for you—if you used a filename to put the program on tape and if you load it using that filename. Even so, it is usually best to start at the beginning just to be sure that the tape is not set beyond the start of the program.

INTERRUPTING CASSETTE OPERATIONS

Occasionally, you will have the cassette recorder set incorrectly for whatever you intended to do. When you realize what happened, stop the cassette unit by pressing the **RUN STOP** key on the computer. Then operate the keys on the cassette to set it correctly and begin again.

AUTOMATICALLY RUNNING THE FIRST PROGRAM ON TAPE

If there is only one program on a tape, or if you want to run the first program on the tape, there

is a shortcut method. Just press SHIFT-RUN STOP. Shifting a key with two labels normally causes the key to perform whatever the top label says. The top label on this key says RUN. That's what it does—but only with the cassette unit and only with the first program on the cassette. Rewind the tape and try it.

USING FILENAMES

This will demonstrate saving, verifying and loading using a filename. First, you must choose a filename that will help you identify the program stored using that name. Use VERIFY TEST as the filename.

It's a good idea to show the filename at the beginning of a program, as a remark, so you always file it the same way. Enter

```
5 REM FILENAME "VERIFY TEST"
```

List the program. The filename is not part of the program, it is just the name that will be used to file it on tape. Rewind and use the following sequence of commands to put the program on tape with a filename, verify it, load it, and run it. Be sure to rewind before each operation.

```
SAVE"VERIFY TEST"
```

```
VERIFY"VERIFY TEST"
```

```
LOAD"VERIFY TEST"
```

```
RUN
```

Now that program is on tape with a filename to distinguish it from other programs that you may put on the same tape.

BACKUP COPIES

You should have a copy of every important program on a separate tape in case the original tape becomes damaged. To do that, save the program once on the primary cassette and save it again on a backup cassette.

If the backup cassette is plainly labeled as backup, you can use the same program names on the backup. I prefer to use a symbol, such as BK, in the filename of backup copies of programs. You could save VERIFY TEST as VERIFY TEST BK, for example.

Some people put backups on the same tape. That isn't the best backup protection but it makes it easier to load a backup copy if the primary copy doesn't load correctly for some reason. Of course, you can do both.

FINDING THE END OF THE LAST FILE ON TAPE

The first program on the cassette is VERIFY TEST. Suppose you write another program and want to save it on the same tape or you want a backup copy of the same program on the same tape.

To save the new program without overwriting the first program, you must advance the tape past the end of the first program.

One way to do that is to write down the names of programs that are on the tape with the starting and ending tape-counter numbers. To add a new program, advance the tape past the end of the last program already on the tape. Then save the new program.

Another way is to VERIFY the last filename on the tape. To do that, you must know what the last filename on the tape is. If that program is still in memory, verification may be successful. If not, it will fail. Either way, the tape stops at the end of the program verified. Then you can save the next program.

To demonstrate that, let's save a backup copy of VERIFY TEST using the new filename VERIFY TEST BK. The program should still be in memory. Rewind the tape and enter

```
VERIFY "VERIFY TEST"
```

The computer will find the program on tape and verify it. The cassette unit stopped at 7, which you know is the end of the first program on that tape. You don't care if the verification was successful or not. You just used it as a way of finding the end of that program.

List the program in memory and change the remark at line 5 like this:

```
5 REM FILENAME "VERIFY TEST BK"
```

Now you can identify which version it is. With the tape counter still at 7, press STOP. Enter

```
SAVE"VERIFY TEST BK"
```

When the cassette unit stops, notice that the tape counter is at about 14. There should be two programs on the tape, with different filenames.

Practice—If you are not accustomed to cassette operations. Try loading each of the two files now on tape.

Use the verify trick to position the tape between the files. Use it again to position the tape at the end of the last file. Clear memory and verify one of the files to see the report when verification is not successful.

COMPLETE SAVE COMMAND

A complete save command has this format:

```
SAVE "filename", device n, command n
```

Symbol *n* means *number*. Every device that the computer communicates with has a device number. The device number for the cassette unit is 1.

The command number is 1, 2 or 3. Command 1 is used by advanced programmers, usually with another programming language. Command 2 instructs the computer to place a special *end-of-tape marker* at the end of the file being saved. Command 3 combines commands 1 and 2 so the computer does both.

The only command that you are likely to use until you are an advanced programmer is command 2. If you use it at the end of a file, it prevents the cassette unit from running past that point while searching for a filename. If you have other files on the tape, behind the end-of-tape marker, the computer will never find them if it starts searching from the beginning of the tape.

You can fool the computer by advancing the tape past the end-of-tape marker and then beginning a search.

Default Values—If you don't supply the device number and a command number, the computer uses *default* values. It assumes that the device is the cassette and it *does not* put an end-of-tape marker at the end of the file.

To specify a value for command 2, you must also supply a device number. To save a program with an end-of-tape marker, the save command would be SAVE "filename",1,2. Unless you have a reason to do that, I suggest that you use the default values and just use SAVE"filename" to save your programs.

SEQUENTIAL DATA FILES

Sequential data files are put on tape and retrieved from tape by statements in a program. They are used to store data used by the program. Doing that requires several steps.

A sequential file is opened by an OPEN statement using a filename for the sequential file. That prepares the computer to place data in that file.

When open, data is sent back and forth between computer memory and the file on tape by input/output statements. These are PRINT#, INPUT# and GET#. The # symbol is used to distinguish them from PRINT, INPUT and GET, which are used as input/output statements for the keyboard and display.

PRINT takes data from memory and displays it on the screen. PRINT# works similarly except that it takes data from memory and puts it in a file on tape.

INPUT and GET accept data from the keyboard and put it in memory with a variable name. INPUT# and GET# are similar except that they accept data from a file and put it in memory with a variable name.

OPENING A SEQUENTIAL FILE

The format for an OPEN statement is

OPEN *file n, device n, command n, "filename"*

Symbol *n* means *number*.

File Number—The file number is a reference number used to identify that file. It can be in the range of 1 to 255. I suggest using small numbers, such as 2 or 3.

Each file that is open must use a different file number. You may have a maximum of 10 data files open at the same time.

Device Number—For the cassette unit, the device number is always 1.

Command Number—The command number specifies whether the file is opened for reading or writing to the file. Command 0 opens the sequential file so it can be read—data will flow from the file to the computer.

Command 1 opens the file so it can be written to—data will flow from the computer to the file. Command 1 causes a special *end-of-file* marker to be placed at the end of the file. When the computer is reading data from the file, the end-of-file marker tells it where to stop.

Command 2 opens the file so it can be written to and places a different marker at the end. Command 2 puts an end-of-tape marker at the end of the sequential data file. An end-of-tape marker is “stronger” than an end-of-file marker. The computer can read past an end-of-file marker, to find another file. It cannot read past an end-of-tape marker.

Unless you have a special reason to want an end-of-tape marker at the end of the sequential data file, use command 1.

Open Statements—Here are the two OPEN statement formats that you will normally use:

OPEN *file n, 1, 0, "filename"* opens the file to read it.

OPEN *file n, 1, 1, "filename"* opens the file to write data into it.

Examples—Assume the filename for a data file is NAMES. It could be names in a mailing list or address book stored on tape. Assume that you decide to open this file as file number 5. To open the file so you can read data from it into the computer, use OPEN 5,1,0,“NAMES”. To open it for writing, so you can send data to it from the computer, use OPEN 5,1,1,“NAMES”.

CLOSING FILES

When a file has been opened to read, it must be closed before it can be opened again to write. If opened to write, it must be closed before it can be opened to read. The close statement is very simple. The format is

CLOSE *n*

The number *n* is the file number used to open the file. To close the NAMES file opened as file number 5, enter the statement CLOSE 5. Suppose a file named FINANCE was opened for reading using file number 3. The open statement would be OPEN 3,1,0,“FINANCE”. The close statement would be CLOSE 3.

SEQUENTIAL-FILE POINTER

Sequential files are opened either for reading or writing. For sequential files, the computer maintains an electronic pointer that works like a bookmark.

When a file is being read, the pointer starts at the beginning and moves along to keep track of what has been read. Reading a file does not affect its content.

When a file is being written to, the pointer moves along the file to show where new data can be placed without writing over existing data.

Opening a sequential file for writing always moves the pointer to the beginning of the file, ready to write a *completely new file*. If the new file is written at a different location on the tape, there will be two files on the tape. They may have different content.

Usually, you should write the new file at the same location on the tape as the old file because two files with the same filename may cause confusion—especially if the contents are different. If the new file is written at the same location on the tape, it will replace the file that was already there.

HOW TO ADD DATA TO AN EXISTING SEQUENTIAL FILE

To add data to an existing sequential file, first open the file for reading and move the entire file into memory. Add data or change the data while it is in memory. Then write it back to the file.

RECORDS

A file is composed of data records, sometimes called *data items*. A record is the smallest amount of data that can be retrieved from the file as a single item. When you put data into the file, you establish the length of each record by placing a data separator between one record and the next. The following paragraphs explain how to do that.

HOW TO SEND DATA TO A FILE

A PRINT# statement sends data to a file. Do not type a space between PRINT and #. The format is

PRINT# *file n, variable name or print list*

A space between PRINT# and the file number is optional. Using the file number sends the data to the filename that was opened with that file number. A single variable name or list of variable names specifies what data is to be sent to the disk. You can also enclose characters in quotation marks and they will go into the file.

For example, PRINT# 2, A\$ will put the characters represented by A\$ into the file. When those characters are read back, they should be given a string variable name because they were put into the file as a string.

PRINT# 2, 12345 will put that number into the file. It should be read back into a numeric variable name.

PRINT# 2, A\$ B\$ C\$ sends all three of those variables into the file as a single record.

INSERTING DATA SEPARATORS

Data separators are commas, semicolons, and CHR\$(13)—a carriage-return symbol. The separators *must be strings*. You can use “,” or “;” or CHR\$(13).

CHR\$(13) is the easiest to use, because that symbol is placed automatically at the end of each data item placed in the file by a PRINT# statement.

The statement PRINT# 4, A\$ will put A\$ into the file and automatically put a CHR\$(13) behind it. A statement such as PRINT# 4, A\$ B\$ C\$ puts a CHR\$(13) symbol at the end of the three strings. No data separator is between the strings, so they will be written and read as a single record.

To treat those strings as individual data records, you can put them in the file separately, each with a separate PRINT# statement. That is the simplest way. Or, you can insert separators between them. This program demonstrates using data separators. Enter

```
NEW
10 OPEN 1,1,1,"TESTFILE":REM WRITE
20 A$="APPLE":B$="BANANA":C$="CHERRY"
30 PRINT# 1,A$ B$ C$
40 CLOSE 1
50 PRINT CHR$(147)"REWIND TAPE"
60 STOP
100 OPEN 1,1,0,"TESTFILE":REM READ
110 INPUT# 1, A$
120 PRINT CHR$(147)
130 PRINT A$
140 CLOSE 1
```

Using A Cassette Recorder

Rewind the tape and run the program. When it stops at line 60, rewind the tape. Then, enter CONT to execute the read part of the program. Tape operations are slow. Be patient.

The INPUT# statement at line 110 got all three of those strings because there were no data separators. Enter

```
30 PRINT#1,A$,B$,C$
```

Rewind and run it that way. The INPUT# statement at line 110 got all three of those strings as one record because there were still no data separators. The commas were taken as part of the record. They acted as automatic tabs on the screen. Enter

```
30 PRINT#1,A$ "," B$ "," C$
```

Rewind and run it. List the program. The data separators worked because they were written as strings. Line 110 got only the first record. Enter

```
110 INPUT# 1,A$,B$,C$
```

```
130 PRINT A$ B$ C$
```

The commas in line 110 are required by BASIC. Otherwise, it would be taken as a single variable name A\$B\$C\$, which is not a legal name. Rewind and run it. That gets all three records separately. To be sure, enter

```
PRINT A$
```

```
PRINT B$
```

```
PRINT C$
```

Probably the best way to separate individual records in the disk file is to PRINT# each one separately. List the program and change lines 30-34 and 130-134 like this:

```
10 OPEN 1,1,1,"TESTFILE":REM WRITE
```

```
20 A$="APPLE":B$="BANANA":C$="CHERRY"
```

```
30 PRINT#1, A$
```

```
32 PRINT#1, B$
```

```
34 PRINT#1, C$
```

```
40 CLOSE 1
```

```
50 PRINT CHR$(147)"REWIND TAPE"
```

```
60 STOP
```

```
100 OPEN 1,1,0,"TESTFILE":REM READ
```

```
110 INPUT# 1,A$,B$,C$
```

```
120 PRINT CHR$(147)
```

```
130 PRINT A$
```

```
132 PRINT B$
```

```
134 PRINT C$
```

```
140 CLOSE 1
```

Rewind and run it. Automatic carriage-return symbols are placed in the file by lines 30-34 just as they are on the screen by lines 130-134.

Numeric variables are handled in a similar way. Make these changes:

```
30 PRINT#1, 12345
```

```
110 INPUT# 1,A ,B$,C$
```

```
130 PRINT A
```

Rewind and run it. Notice that the number prints with a leading space. There is also a trailing space, but you can't see it.

STATUS

The computer maintains a variable named STATUS in memory. It is usually written ST in program lines. This variable is used to monitor data input-output operations. It is used in programs to detect errors and to detect the end of a file.

For the cassette unit, a value of 0 is normal. If an end-of-file marker is encountered, ST is set to 64. If an end-of-tape marker is found, ST is set to -128. Enter

PRINT ST

It should be 64 because the demonstration program reads all of the data in the file, each time it runs. That is because three records are put in the file and three are read.

Monitoring the value of ST allows you to read all records from a file without knowing how many there are. After each read from the tape, check the value of ST. If it is 0, keep reading. If it is not 0, or specifically if it is 64 or -128, stop reading. Normally, it is sufficient to test ST to see if it is 0.

If you continue inputting from a file when it has no more data, BASIC displays an error message. To see the message, change line 110 to ask for one more item, such as D or D\$, and run the program.

The STRING TOO LONG error message occurs when no data is found in a file. Sometimes, error messages are not very descriptive of the problem.

GET#

A GET statement addresses the keyboard buffer. It can be used to get a single character. In a loop that runs 10 times, it can empty the keyboard buffer.

When GET# is used with a tape file, the file is a reservoir of characters, similar to the keyboard buffer.

GET# does not recognize data separators. Instead, it gets them from the file. You should have a data file named TESTFILE on tape from the last demonstration program. Let's use a GET# statement to retrieve it. Enter

```
NEW
10 OPEN 1,1,0,"TESTFILE"
20 GET# 1,A$
30 PRINT "[ " A$ " ] ";
40 IF ST=0 GOTO 20
50 CLOSE 1
```

That routine runs a loop to get characters until ST=0. Then it closes the file and ends the program. At line 30, each A\$ is printed in brackets so the display will be easier to interpret. Rewind the tape and run the program.

Now you can see the leading and trailing spaces that print with the number 12345. They are captured inside the brackets. At the end of each line is the first half of a pair of brackets. A carriage-return code is inside the brackets. The second half of the brackets is at the beginning of the next line. CHR\$(13) does that. A carriage return is not a printable character, but its effect is visible.

To see the ASCII codes for these characters, change line 30 like this:

```
30 PRINT "[ " ASC(A$+CHR$(0)) " ] "
```

Including CHR\$(0) with A\$ prevents a BASIC error if A\$ is a null. Rewind and run it. This time, CHR\$(13) doesn't produce carriage returns because CHR\$(13) is not being printed—its ASCII code number is being printed. Printing *the number 13* doesn't cause a carriage return.

GET# is a very powerful way to examine tape files to see exactly what is in them.

SUMMARY

What has been shown so far is that a program already *in the computer* can use a data file on tape—provided the tape is rewound to the beginning of the file each time the program needs to use

the file. Some good people learn the hard way that rewinding is necessary — by forgetting to do it. I'm sure you didn't do that.

A more practical demonstration would show how a program *on tape* can control a data file on tape. That raises the question of where to put the program. If the data file is to be used more than once, it must be possible to rewind to the beginning of it without error, every time. The best place for a data file is at the beginning of the tape.

Therefore, the best place for the program is *behind* the data file.

In the real world, a cassette user must store programs on tape. The programs must send data to a tape file and get it back. Usually, the program will send data to the file, get it back, change it, and send it to the file again. For example, if you were storing and then updating a mailing list, it would be necessary to do that. The next demonstration program does all of those things.

FORMAT FOR TAPE COMMANDS	
Symbol <i>n</i> means <i>number</i>	
COMMAND	FORMAT
LOAD	LOAD "filename"
SAVE	SAVE "filename"
OPEN	OPEN file <i>n</i> , 1, 0, "filespec" (to read) OPEN file <i>n</i> , 1, 1, "filespec" (to write)
CLOSE	CLOSE file <i>n</i>
PRINT #	PRINT file <i>n</i> , command or print list
INPUT #	INPUT # file <i>n</i> , command or input list
GET #	GET # file <i>n</i> , command or input list
VERIFY	VERIFY "filename"

A CASSETTE INDEX PROGRAM

It is sometimes useful to have an index of files on a tape. The tape itself doesn't keep a list of what is on it. You can keep the list separately, on paper, if you wish. Another way is to keep it on the tape as an index file.

You have to write in the filenames either way. An index on the tape is not likely to become separated from the tape. Also, this index program provides a handy way to load files from the tape. The method is simple enough to be used by anybody, without typing filenames or load commands.

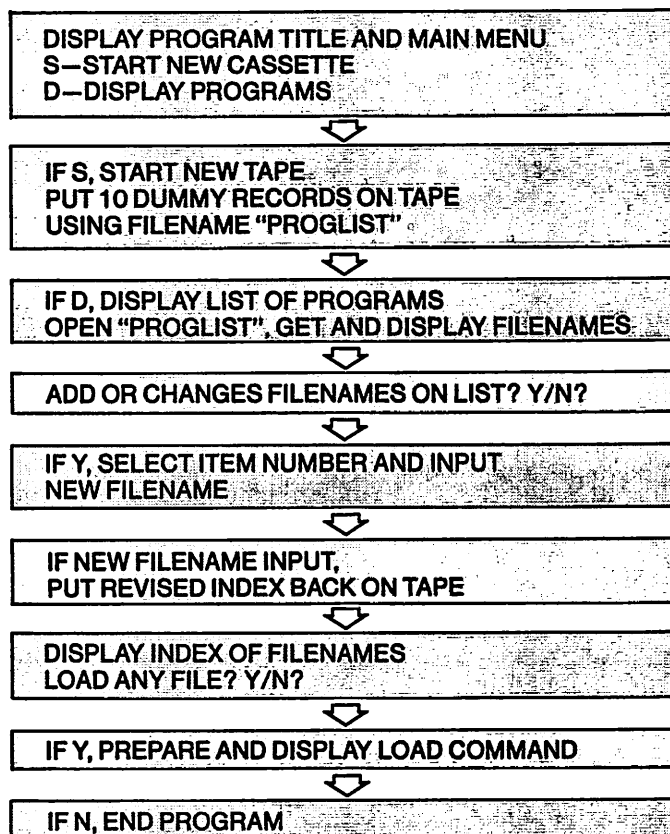
The main disadvantage of this program is that it runs slowly because tape operations are slow. I suggest that you enter and run it for its educational value. Then, you can use it or not, as you choose.

The program is modular. Each module begins with a remark that says what it does. That makes it easy to understand and change programs. It is not the best way to write programs for tape because there are a lot of remarks and spaces in the lines. Longer programs take longer to load and save. Later, I will discuss compressing programs so they load faster.

On page 231 is a flow chart for this program, showing the modules and what they do. Please look it over, so you know what to expect as you enter it. Enter

```
NEW
5 REM FILENAME "INDEX"
10 PRINT CHR$(147):PRINT
20 PRINT TAB(8)"CASSETTE INDEX PROGRAM":
  PRINT:PRINT
30 PRINT TAB(8)"S - START NEW CASSETTE"
40 PRINT TAB(8)"D - DISPLAY PROGRAMS"
50 GET A$:IF A$="" GOTO 50
60 IF A$="D" GOTO 200
70 IF A$<>"S" GOTO 50:REM TRAP
80 REM -----
```

FLOWCHART FOR CASSETTE INDEX PROGRAM



A flowchart is a form of program plan. It helps when planning a program and also as part of the program documentation—so another person can understand the program more easily.

This flowchart shows the major segments of the Cassette Index program, in the same order as the program.

That segment gets the program started by displaying a menu. To test, run the program. Pressing D should produce a program error because the lines for that choice are not written. Test the error trap at line 50 by pressing a key that is not S or D.

Line 130 in the following segment has 16 periods between quotation marks. Enter

```

90 REM START NEW TAPE
100 PRINT CHR$(147):PRINT:GOSUB 900:
    PRINT:REM TAPE INSTR
110 OPEN 1,1,1,"PROGLIST":PRINT
120 FOR I=1 TO 10
130 PRINT # 1,"....."
140 NEXT:CLOSE 1:PRINT CHR$(147)
150 REM -----
  
```

That segment begins by calling subroutine 900, which asks the user to rewind the tape. It will be used often. Then the program opens a data file called PROGLIST, which is a list of programs on the tape. It puts 10 dummy records on the tape. Each dummy record is a string of 16 periods. Each can later be replaced by a program filename. If you want to index more programs, use more than 10 dummy records.

The program you are entering is in memory. It has not yet been put on tape. When you run it to test, it will put the data file on tape with the dummy records in it. Because the tape will be rewound, the dummy file becomes the first file on the tape. That has two purposes. It creates the file on the

Using A Cassette Recorder

tape. It holds space at the beginning of the tape for that file. This program will be put on tape immediately after the PROGLIST data file.

Before running the program again, enter the subroutine that begins at line 900, so you can test it. Enter

```
890 REM SUBR TAPE INSTRUCTIONS
900 PRINT
910 PRINT"REWIND TAPE AND PRESS RETURN"
920 GET A$:IF A$=" "GOTO 920
930 RETURN
940 REM -----
```

The GET routine at line 920 just stops the program until the user rewinds the tape and presses RETURN. To test, enter a temporary STOP at line 151:

```
151 STOP
```

Then run the program. At the menu, press S. Follow the instructions. The cassette recorder should run. It is putting the dummy PROGLIST file on tape. The next segment of this program will check to see if it is there. When the program stops, delete line 151. List -150. Then enter

```
190 REM DISPLAY INDEX
200 PRINT CHR$(147):PRINT:GOSUB 900:
    REM TAPE INSTR
210 OPEN 1,1,0,"PROGLIST"
220 C=1:REM ARRAY COUNTER
230 INPUT# 1,NDX$(C)
240 IF ST=0 THEN C=C+1:GOTO 230
250 CLOSE 1
260 FLAG=0:REM SET IF CHANGE ITEM
270 GOSUB 1000:REM DISPLAY INDEX
280 REM -----
```

Line 210 opens PROGLIST to read each record into an array NDX\$() in memory. The variable C is used to count array locations in NDX\$(). Line 230 inputs records from the tape file, one at a time, and puts them in NDX\$(). When the counter C is 1, the first record goes into NDX\$(1), and so forth. This array is not dimensioned because it holds only 10 items.

Line 240 checks STATUS to see if the file still has data. If so, it jumps back to line 230 to get the next record and put it in the array. When all records have been read, ST will equal 64. Line 240 will not execute. Line 250 closes the file.

Line 260 establishes a FLAG that will be tested later to see if the user has changed any of the elements in NDX\$(). Of course, the user must change them to get program filenames into the array and into the tape file. Testing the FLAG tells the program when the list of programs has been changed, so it will be put back on the tape to replace the preceding list. You'll see how that works in a minute.

Because the list of programs is displayed frequently, it is done by a subroutine at line 1000. Let's enter the subroutine next. Enter

```
990 REM SUBR DISPLAY INDEX
1000 PRINT CHR$(147):PRINT:PRINT
    "ITEM" TAB(8)"PROGRAM":PRINT
1010 FOR I=1 TO C:PRINT I TAB(5)"—"
    TAB(8)NDX$(I):NEXT:PRINT
1020 RETURN
```

Now you can test again. List -280. Put a temporary STOP at line 281. Run the program. Select D. When the display asks you to PRESS PLAY ON TAPE, it is executing line 210, which opens the file to read it. The message PRESS PLAY ON TAPE is put on the display by the computer. Then the program reads the file and builds the array NDX\$(). The subroutine displays the array and the program stops at line 281. Delete line 281.

You should see the 10 dummy records from the file, displayed on the screen. Now the program offers the user the opportunity to change any of those items. That means, type in program filenames that are on the tape. List -280 and enter

```

290 REM LOOP TO GET FILENAMES
300 PRINT:PRINT"ENTER FILENAME? Y/N?"
310 GET A$:IF A$="" GOTO 310
320 IF A$="Y" GOTO 400
330 IF A$="N" AND FLAG<>0 GOTO 500:
    REM ITEM CHANGED IN THIS LOOP
340 IF A$="N" AND FLAG=0 GOTO 600:
    REM NOT CHANGED
350 GOTO 310:REM TRAP
360 GOTO 900:REM TAPE INSTR
370 REM -----

```

The user can choose to enter a filename, either to replace one of the dummy records in PROGLIST, or to change a record to hold a different filename. If he says *yes* to the question at line 300, line 320 jumps ahead to a routine at 400.

The routine at 400 gets the new filename and processes it. It adds 1 to the value of FLAG, so it is no longer zero. Then the routine at 400 loops back to line 300 to ask the question again. That allows the user to enter more than one filename at a time.

This part of the program is in a big loop from 300-460. Any line in this loop may execute more than once. At line 330, if A\$ is N, the user has chosen not to enter a new filename on that pass through the loop. But, he may already have entered a new filename on an earlier pass.

Therefore, line 330 tests both the value of A\$ and the value of FLAG. If A\$ is N and FLAG is not 0, one or more new filenames were entered. Because there were changes, a new PROGLIST file must be put on tape to replace the old file. Therefore, line 330 jumps ahead to line 500, which does that.

At line 340, if A\$ is N but FLAG is still 0, no changes were made. Line 340 jumps ahead to line 600, which does not replace PROGLIST on tape.

To test some of this, use line 600 for a temporary STOP. Run the program and select S at the menu. When you see ENTER FILENAME? Y/N?, press Q. The trap at line 350 should work, and that input should be ignored.

Then press N. Because FLAG is still 0, the program should jump to 600 and stop. That doesn't test the other N option, but it will be tested by the next segment. Leave the STOP at 600.

List -370 and hold down the CTRL key while the program is being displayed. At line 320, if the user presses Y, he intends to enter a new filename. Line 320 jumps to line 400, which is the next segment. Enter

```

390 REM INPUT NEW FILENAME
400 INPUT"WHICH ITEM NUMBER";L
410 PRINT:PRINT"ENTER NEW ITEM"L
420 INPUT NI$:REM NEW ITEM
430 NDX$(L)=NI$:REM PUT IN ARRAY
440 FLAG=1:REM ITEM CHANGED

```

(Program continued on next page.)

Using A Cassette Recorder

```
450 GOSUB 1000:REM DISPLAY NEW INDEX
460 GOTO 300:REM DO AGAIN?
470 REM -----
```

This routine will accept new filenames, put them in the array NDX\$() and display the revised list. It will do it until the user presses N at the keyboard to indicate that no new filenames are to be entered.

The item number is designated at line 400 as the variable L. A new item is entered at line 430 as NI\$. Then line 430 puts the new item into array NDX\$() at location L. If item 5 was selected for a new filename, it goes into NDX\$(5).

Line 440 sets the FLAG. When the program leaves this loop, it tests FLAG to decide whether or not to put the array NDX\$() back on the tape.

To test, you still have a STOP at line 600. Run the program. Select D at the menu, to display programs. Dummy filenames on the tape will be displayed. Press Y to enter a new filename. Select item number 1. Enter INDEX because that will be the first program on this tape.

The revised list of filenames should be displayed, with INDEX shown as item 1. This list will not actually go on tape yet, so you can put anything you want in it. Press Y again to enter another filename. Put your name at item 2. Put somebody else's name at item 3. The program loops until you press N, indicating that you have finished entering new filenames. Press N.

List -470. The bottom of the big loop that changes filenames is at 460. The program has two exits from this loop. One is at line 330, the other at 340. The user must press N to get out of the loop. The exit route is determined by the value of FLAG. If a filename was changed, FLAG will no longer be 0. Line 330 will execute and jump ahead to line 500, where the list of filenames will be put back on the tape, to replace the old list. Enter

```
490 REM IF CHANGED, PROGLIST TO TAPE
500 GOSUB 900:PRINT:REM TAPE INSTR
510 OPEN 1,1,1,"PROGLIST"
520 FOR I=1 TO 10:PRINT# 1,NDX$(I):NEXT
530 CLOSE 1
540 REM -----
```

The way to test that routine is run the program and enter some new filenames. The program will stop at the temporary STOP at line 600, but the revised PROGLIST should already be on the tape. Then, run it again and select D to display the filenames. If you select S, a new dummy file will be created.

Test by doing that. It doesn't matter what you enter for new filenames. They will be changed later, anyway. See if the revised list is displayed correctly. If not, look for typos in the program.

The program is complete now, as an index. The remaining two routines make it easy to load a program from the index. Enter

```
590 REM LOAD OR END
600 GOSUB 1000:REM NEW DISPLAY
610 PRINT"LOAD ANY PROGRAM? Y/N?"
620 GET A$:IF A$="" GOTO 620
630 IF A$="N" THEN END
640 IF A$<>"Y" GOTO 620
650 REM -----
```

If the user presses N, the program ends. If he presses Y, the next routine executes to get the number of the program to be loaded. Enter

```
690 REM GET ITEM NUMBER TO LOAD
700 GOSUB 1000:REM NEW DISPLAY
710 INPUT"ENTER ITEM NUMBER";P
720 REM -----
```

The item number is taken as P. It is the program number on the index list and also the element number of that program in the array NDX\$(). The next routine loads it. Enter

```
790 REM DISPLAY LOAD COMMAND
800 GOSUB 900:REM TAPE INSTR
810 PRINT CHR$(147):PRINT:PRINT:
    PRINT"PRESS RETURN":PRINT
820 PRINT TAB(6)"LOAD "CHR$(34)NDX$(P)
    CHR$(34)CHR$(145)CHR$(145)CHR$(145)
830 END
840 REM -----
```

That's the end of the program, except for the subroutines that are already entered. This routine does a little trick. It prints a load command on the screen. For example, if the user selected item 1 on the list, and item 1 is INDEX, load command placed on the screen is LOAD "INDEX".

Then it moves the cursor up a few lines and ends the program. When a program ends, the READY prompt appears and the cursor appears on the following screen row. Before ending the program, the cursor is moved up enough that it ends on the same line as the LOAD command, after the program ends.

To enter that load command, all the user must do is press RETURN. It is already typed on the screen, and the cursor is on the same line.

Line 800 calls subroutine 900 to get the tape rewound again, to be sure that the computer can find the selected program. Line 810 displays an instruction to the user.

Line 820 does the trick. It displays the load command. The selected filename is NDX\$(P). That filename is wrapped in quotation marks by a CHR\$(34) on each side of it. CHR\$(145) moves the cursor up one row on the screen. It is the ASCII code for a cursor-up keystroke. Line 830 ends the program with the LOAD command all set up, waiting for a RETURN keystroke to enter it. Because the program has ended, the computer is in the immediate mode.

To test the complete program, you need a program file on the tape, so you can load it.

PUTTING A PROGRAM FILE ON TAPE WITH A DATA FILE

The INDEX program should go on tape immediately after the data file used by that program. To do that, run the program again. Select D to display the existing data file. When the program stops with the list of programs displayed, press N twice. That ends the program without running the tape again.

Look at the tape counter. It should be at 7 or 8. That is the end of the data file. Press the STOP key on the cassette unit. Enter

```
SAVE"INDEX"
```

Follow the instructions on the screen. The tape counter should stop at about 26. Then, put a backup copy of the INDEX program on tape, immediately behind the copy you just saved. Press STOP on the cassette unit. Enter

```
SAVE"INDEX BK"
```

Follow instructions. The tape counter should stop at about 44. Now, you have a data file and two program files on the tape. You should run the INDEX program to put the two program filenames into the index.

Run the program. Start over by selecting the S option. Then put INDEX on the list as item 1 and INDEX BK on the list as item 2. Later, if you put more programs on this tape behind INDEX BK, add them to the list.

When the program asks if you want to LOAD ANY PROGRAM, reply no. Rewind the tape.

USING THE INDEX PROGRAM

To run the first program on a tape, press SHIFT-RUN STOP. Do that. It will take a minute or so because the INDEX program is being loaded from tape. When you ran it before, it was already in memory. The screen will flicker on and off, which is normal. Eventually, the menu should be displayed.

Press D to display the programs. Don't enter a new filename. Load program number 1. When it is loaded, list it.

IS THIS PROGRAM PRACTICAL?

It takes about two minutes to load everything, display the index, and get a LOAD command ready to enter by pressing RETURN. For a person who is unfamiliar with computers, it makes finding and loading cassette programs very simple and easy to do.

Also, it keeps an index of the tape in a way that makes it very hard to lose or misplace—unless you lose or misplace the cassette.

On the other hand, if you know that the program you want to run is on the tape and don't mind typing a load command, you can have the program up and running a lot faster.

DOES THE DATA FILE HAVE TO PRECEDE THE PROGRAM FILE?

No. A disadvantage of putting the data file at the beginning of the tape is that you must use dummy records to hold space on the tape. Also, the length of the data file is limited to the number of dummy records that were originally put on the tape.

If you put the program at the beginning and the data file after the program, it will work OK. After the program is loaded, the tape player stops and waits for the next command. When the program opens the data file, the tape will resume running from where it stopped. The data file will be found and read into memory.

If the data file is behind the program file, there is no need for dummy records. You can make the data file as long as you wish. You can make it longer each time the program runs, if you add data to the file—provided it doesn't run into another file behind it on the tape.

If the data file is behind the program file, and the data file is read only once, that is probably the best way. If the data file must be accessed more than once by the program, then there is the problem of rewinding past the beginning of the data file so it can be read again, or written to.

The most reliable way to assure that is to rewind all the way to the beginning of the tape. Another way is to observe the setting of the tape counter at the beginning of the data file and rewind past that setting. If you rewind the tape all the way to the beginning, the tape must then run past the entire program before it can read the data file, which slows down the operation. Rewinding just past the beginning of the data file is better, but it requires more attention and cooperation from the operator.

If you are willing to use one cassette for one program, consider putting the program on one side and the data file on the other. After loading the program, turn the cassette over. Then the data file can be at the beginning of the tape. It can grow longer during successive runs of the program. It can be accessed quickly, and rewinding back to the beginning is simple and automatic.

WHAT IF THERE IS MORE THAN ONE DATA FILE ON THE TAPE?

You can have several programs on a cassette, each with one or more data files. The only problem is locating the tape to read the correct data file. Probably the surest way to do that is rewind the tape each time the data file is to be read or written to.

If a program calls for a data file by name, and the tape is rewound past the beginning of that file, the computer will automatically find and read that data file. If the data file is a long way from the beginning of the tape, it will take a long time.

Tape cassettes are inexpensive. Consider putting one program with its data files on each side. If you do that, you don't need an index program. You can write the name of the program on the cassette itself.

CLOSING FILES

It's important to close all files before ending the program. If you don't, some of the data being sent to tape may not actually be recorded on the tape.

MODIFYING THE INDEX PROGRAM

The index program demonstrated what you need to know to use a sequential data file from a program stored on tape.

It can be modified easily to keep a list of nearly anything. For example, you can put names and addresses in it. They can be changed when necessary. It will be slow to look up a name if it is near the end of the list, but may be faster than finding it another way.

If you don't want to use dummy records, put the data file behind the program file on the tape. If you have a printer, you can print out the list of names, or whatever you have in the data file, and use the printout for reference. When you make changes to the list on tape, print out a new copy.

PRACTICE

Using the index program as a guide, make a program to hold and display a list of names and phone numbers. Put it behind the program file on the tape, so it can become longer and dummy records are not needed. Keep the list in alphabetical order.

When you read the data file into memory, read it into an array. Provide a way to make changes to the list by changing array elements while the file is in memory. Provide a way to add names and phone numbers to the list and put them in alphabetical order using an insert sort. Then write the file back to tape, so the changes are permanent.

Even though a sequential file must be read from the beginning to get it into memory, it doesn't have to be displayed from the beginning. Once it is in an array in memory, you can get and display any element of the array.

Provide a way to scan through the entire list, one screen load at a time. Provide an alternate choice of searching the list and displaying only the desired name and phone number.

Doing some of those things may not be easy if you haven't done them before. After you have done them, you will know how to use tape files.

17 Using A Printer

This chapter applies specifically to the Commodore VIC 1525 Graphic Printer. It applies generally to any printer that can be used with the Commodore 64.

The Graphic Printer is a type called *dot-matrix printer*. The printed characters are made of small dots on paper, arranged in a pattern to make the character. The space for each character is a rectangle that is 6 dots wide and 7 dots tall. If all dots are printed to make a solid rectangle of dots, there will be 42 dots.

As the print head travels across the paper, it prints one character and then leaves a small amount of space between that character and the next. This separates characters horizontally. The maximum number of characters in a horizontal line is 80.

Normally, the printer makes 6 lines of characters per inch, from top to bottom of the page. When the paper is advanced to make the next line, a small space is left between lines. This separates lines on the page.

A standard page is 8-1/2 inches wide and 11 inches tall. This provides room for 66 lines of characters on a page, with no margins at top and bottom. Normally, there are margins at top and bottom, and on both sides.

The printer uses standard continuous-form paper with tear-off sprocket holes on the sides and perforations between pages. Holes on each side of the paper fit on sprockets in the printer. Sprockets are called *tractors* because they pull the paper through the printer.

CONNECTING THE PRINTER

If your printer is not connected, turn off the computer. Plug the printer cable into the serial port on the back of the keyboard unit. If you have a disk drive, the cable from the disk drive is already plugged in there. In that case, plug the printer cable into one of the serial ports on the back of the disk-drive unit.

Plug the AC power cord into an AC outlet. Follow the startup and self-test instructions in the booklet packaged with the printer. The Device Selector switch on the back of the printer is marked T 4 5. T is the self-test setting. Set it to 4. This causes the printer to become device 4.

Load paper into the printer if it isn't already loaded. Feed it from the back. Slide it downward into the slot behind the tractor. It will emerge in front of the tractor, between the print head and the platen.

Set the tractors to the correct width by sliding them to the left or right. Open them and place the sprocket holes of the paper over the sprocket teeth of the tractors. Close the tractor covers. Slide them left or right as necessary to pull the paper taut between them. Slide both together to position the left margin of the paper.

Turn on all accessories, then turn on the computer.

WHAT CAN THE PRINTER DO?

A printer can print copies of a program in memory. It will use up to 80 characters for each program line. A program line that uses two lines on the screen will use only one line on paper.

The printer can make all of the characters that can be displayed on the screen. It has both upper-case-and-graphics and upper-and-lower-case modes, the same as the display screen. It can print any of the standard characters in either normal or double-width. It can print reversed characters.

PRINTER FONTS

Code	Upper	Wide	Lower	Wide	Code	Upper	Wide	Lower	Wide
32					96				
33					97				
34		(QUOTATION MARKS)			98				
35	#	#	#	#	99				
36	#	#	#	#	100				
37	#	#	#	#	101				
38	#	#	#	#	102				
39	#	#	#	#	103				
40	#	#	#	#	104				
41	#	#	#	#	105				
42	#	#	#	#	106				
43	#	#	#	#	107				
44	#	#	#	#	108				
45	#	#	#	#	109				
46	#	#	#	#	110				
47	#	#	#	#	111				
48	#	#	#	#	112				
49	#	#	#	#	113				
50	#	#	#	#	114				
51	#	#	#	#	115				
52	#	#	#	#	116				
53	#	#	#	#	117				
54	#	#	#	#	118				
55	#	#	#	#	119				
56	#	#	#	#	120				
57	#	#	#	#	121				
58	#	#	#	#	122				
59	#	#	#	#	123				
60	#	#	#	#	124				
61	#	#	#	#	125				
62	#	#	#	#	126				
63	#	#	#	#	127				
64	#	#	#	#	128				
65	#	#	#	#	129				
66	#	#	#	#	130				
67	#	#	#	#	131				
68	#	#	#	#	132				
69	#	#	#	#	133				
70	#	#	#	#	134				
71	#	#	#	#	135				
72	#	#	#	#	136				
73	#	#	#	#	137				
74	#	#	#	#	138				
75	#	#	#	#	139				
76	#	#	#	#	140				
77	#	#	#	#	141				
78	#	#	#	#	142				
79	#	#	#	#	143				
80	#	#	#	#	144				
81	#	#	#	#	145				
82	#	#	#	#	146				
83	#	#	#	#	147				
84	#	#	#	#	148				
85	#	#	#	#	149				
86	#	#	#	#	150				
87	#	#	#	#	151				
88	#	#	#	#	152				
89	#	#	#	#	153				
90	#	#	#	#	154				
91	#	#	#	#	155				
92	#	#	#	#	156				
93	#	#	#	#	157				
94	#	#	#	#	158				
95	#	#	#	#	159				

(Continued on next page.)

PRINTER FONTS (continued)

Code	Upper	Wide	Lower	Wide	Code	Upper	Wide	Lower	Wide
160					208	┌	┌	┌	┌
161	■	■	■	■	209	●	●	●	●
162	■	■	■	■	210	●	●	●	●
163	■	■	■	■	211	●	●	●	●
164	■	■	■	■	212	●	●	●	●
165	■	■	■	■	213	●	●	●	●
166	■	■	■	■	214	●	●	●	●
167	■	■	■	■	215	●	●	●	●
168	■	■	■	■	216	●	●	●	●
169	■	■	■	■	217	●	●	●	●
170	■	■	■	■	218	●	●	●	●
171	■	■	■	■	219	●	●	●	●
172	■	■	■	■	220	●	●	●	●
173	■	■	■	■	221	●	●	●	●
174	■	■	■	■	222	●	●	●	●
175	■	■	■	■	223	●	●	●	●
176	■	■	■	■	224	●	●	●	●
177	■	■	■	■	225	●	●	●	●
178	■	■	■	■	226	●	●	●	●
179	■	■	■	■	227	●	●	●	●
180	■	■	■	■	228	●	●	●	●
181	■	■	■	■	229	●	●	●	●
182	■	■	■	■	230	●	●	●	●
183	■	■	■	■	231	●	●	●	●
184	■	■	■	■	232	●	●	●	●
185	■	■	■	■	233	●	●	●	●
186	■	■	■	■	234	●	●	●	●
187	■	■	■	■	235	●	●	●	●
188	■	■	■	■	236	●	●	●	●
189	■	■	■	■	237	●	●	●	●
190	■	■	■	■	238	●	●	●	●
191	■	■	■	■	239	●	●	●	●
192	■	■	■	■	240	●	●	●	●
193	■	■	■	■	241	●	●	●	●
194	■	■	■	■	242	●	●	●	●
195	■	■	■	■	243	●	●	●	●
196	■	■	■	■	244	●	●	●	●
197	■	■	■	■	245	●	●	●	●
198	■	■	■	■	246	●	●	●	●
199	■	■	■	■	247	●	●	●	●
200	■	■	■	■	248	●	●	●	●
201	■	■	■	■	249	●	●	●	●
202	■	■	■	■	250	●	●	●	●
203	■	■	■	■	251	●	●	●	●
204	■	■	■	■	252	●	●	●	●
205	■	■	■	■	253	●	●	●	●
206	■	■	■	■	254	●	●	●	●
207	■	■	■	■	255	●	●	●	●

You can type at the computer keyboard and print out what you type. That can be done in the immediate mode or when a program is running.

The printer can be controlled by a program. It will print what the program tells it to.

In all of the operations just described, the printer prints standard characters, such as letters, numbers and graphics symbols.

The printer also has a graphics mode. In that mode, you can control individual dots. This allows you to create characters that are not on the keyboard or represented by ASCII codes. It allows you to print graphics that you design.

All of those operations are discussed in this chapter, except the graphics mode. The graphics mode is discussed in Chapter 19, along with screen graphics.

OPENING THE PRINTER

Communication is established with the printer by an OPEN statement, similar to that used with a disk drive or cassette unit. The form is

OPEN *file n*, *device n*, *mode n*

Symbol *n* means *number*. The file number can be any number from 1 to 255 that is not being used to open another device such as a disk drive. I suggest that you use file number 4 for the printer.

The device number is set by the switch marked T 4 5 on the back of the printer. Setting that switch to 4 causes the printer to become device 4.

Mode number 0 selects upper-case-and-graphics characters. Mode 7 selects upper-and-lower-case. These are the same two character sets that are used on the display screen.

A difference is that switching modes on the screen causes all characters being displayed to change to the selected character set—including characters that were displayed originally in the other mode. Switching modes at the printer does not change characters that have already been printed.

Here are examples of the two OPEN statements using file number 4:

OPEN 4,4,0 opens the printer in the upper-case-and-graphics mode.

OPEN 4,4,7 opens the printer in the upper-and-lower-case mode.

CLOSING THE PRINTER

After opening and using the printer, you should close it with a CLOSE statement. This frees the file number for use by another device, if needed. The close statement uses the same file number that was used to open the printer. If opened as file number 4, the close statement is CLOSE 4.

SENDING DATA TO THE PRINTER

When the printer is open, a PRINT# *n* statement sends data to the printer, to be printed. Symbol *n* is the file number used to open the printer.

If the printer was opened as file 4, PRINT# 4, "HELLO" will cause the printer to print HELLO. A comma must be used after the file number. A space after the comma is optional.

This command is the same one used to send data to a disk file or cassette file. The computer regards the printer as a device that can be treated as a file. When data is sent to the printer by a PRINT# statement, it is printed, not actually filed. Therefore, no filename is used when sending data to the printer by that method.

I will refer to this method by the phrase *printing as a file*. The procedure is to open the printer and print whatever you wish, using PRINT# statements. Then close the printer. Enter

```
OPEN 4,4,0
```

```
PRINT# 4, "HELLO"
```

```
CLOSE 4
```

You will find it much easier to type PRINT#4, rather than PRINT# 4. The computer accepts either form. I used PRINT# 4 in this book to make it clear that PRINT# is one word.

SUBSTITUTING THE PRINTER FOR THE SCREEN

Another way to print on paper is to divert the characters that would ordinarily be displayed on the screen and send them to the printer instead. That is done by the statement CMD *file n*, in either the immediate or program mode.

The procedure is to open the printer and then issue the CMD command. After that, anything that would have been displayed on the screen will be printed by the printer instead.

The CMD command causes the word PRINT to have a different meaning. A PRINT statement following a CMD statement prints characters on paper instead of on the screen.

When you divert characters from the screen to the printer, TAB() statements and commas for automatic tabs do not work the same on the printer as on the screen. On the printer, a TAB() statement has the same effect as a SPC() statement. TAB(3) and SPC(3) both move the print head three columns to the right of the last character printed. A comma used for an automatic tab has the same result as SPC(10).

There is a way to set tabs on the printer. It will be discussed later.

To switch back to the screen, instead of the printer, use the command PRINT# *n*. A comma is not required. If the printer was opened as file number 4, the command is PRINT# 4. That statement sends a blank line to the printer. Instead of printing a blank line, the result is to switch the characters back to the screen. Following PRINT statements will cause a display on the screen.

When the printer has been substituted for the screen, it displays READY prompts, just like the screen does.

Demonstration—In the immediate mode, enter

```
OPEN 4,4,0
CMD 4
PRINT"NOW IS THE TIME"
PRINT# 4
PRINT"NOW IS THE TIME"
CLOSE 4
```

After the printer was opened in the upper-case-and-graphics mode, CMD 4 switched characters to the printer. The printer printed READY. The first PRINT statement printed on the printer instead of the screen. Because the computer is in the immediate mode, the printer printed READY again.

The PRINT# 4 statement switched following characters back to the screen. The screen displayed READY. The next PRINT statement printed on the screen. The screen displayed READY again. The CLOSE statement closed the printer and released the file number used to open the printer.

That same sequence of events can be used in a program, without READY prompts. As review, the sequence is

- 1) OPEN the printer.
- 2) Use CMD to send data to the printer. Otherwise, this data would appear on the screen.
- 3) Use PRINT# *n* to switch back to the screen.
- 4) Switch back and forth as much as you wish.
- 5) CLOSE the printer.

LISTING A PROGRAM

The same method is used to list a program in memory. To have a program to list, enter

```
10 REM DEMO LISTING
20 PRINT"THIS PROGRAM LISTED"
30 END
```

Run it and list it on the screen. To list it on the printer, enter

```
OPEN 4,4,0
CMD 4
LIST
PRINT# 4
CLOSE 4
```

Program listings on paper are useful as a permanent record of the program. When writing a long program, a listing is a great help in keeping track of what has been entered and in fixing problems when they occur.

PRINTING THE DISK DIRECTORY

If you have a disk drive, you can print the directory of each disk on paper. If you have a lot of disks, printed directories are helpful.

The disk directory loads like a program and prints like a program. The numbers at the left, in the directory, are actually the number of blocks used by each program on the disk. When the directory is loaded as a program, those numbers become line numbers in the "directory program."

To print the directory on a disk, enter

```
LOAD "$",8
OPEN 4,4,0
CMD 4
LIST
PRINT # 4
CLOSE # 4
```

PRINTING FROM A PROGRAM

In the immediate or program mode, you can print by substituting the printer for the screen or print as a file. This program demonstrates doing it both ways in a program. Enter

```
NEW
10 REM DEMO PRINTING
20 PRINT CHR$(147)
30 PRINT"THIS IS ON SCREEN."
40 PRINT"NOW SWITCHING TO PRINTER."
50 OPEN 4,4,0
60 CMD 4
70 PRINT"THIS IS ON PRINTER"
80 PRINT "1","2","3","4","5","6","7","8"
90 PRINT"012345678901234567890123456789"
100 PRINT SPC(5)"A"
110 PRINT TAB(5)"A" TAB(6)"B"
```

Run the program. Notice that commas move the print head 11 columns from wherever the last character was printed. SPC() statements work correctly. TAB() works the same as SPC().

List the program. The reason it lists on the printer instead of the screen is that the printer is still substituted for the screen. Enter

```
PRINT # 4
CLOSE 4
```

Now list the program on the screen.

TABBING THE PRINT HEAD

A special command is used to tab the print head, using the left margin as a reference. It requires two-digit numbers. To tab 6 spaces, you must use 06 as the number of columns to be tabbed.

The command is CHR\$(16) "n", in which *n* is the number of columns to be tabbed. This command must be part of a PRINT# statement. It is usually part of a list of items to be printed. Enter

```
120 PRINT # 4:REM SWITCH BACK TO SCREEN
130 PRINT"NOW ON SCREEN AGAIN"
```

(Program continued on next page.)

```
140 PRINT # 4,"A PRINT # COMMAND ALWAYS ";
150 PRINT # 4," SENDS DATA TO PRINTER IF PR
INTER IS OPEN."
160 PRINT # 4
170 PRINT # 4,CHR$(16)"05" "C";
180 PRINT # 4,CHR$(16)"06" "D"
```

Line 120 switches characters back to the screen again. But that merely restores the normal meaning of the word PRINT. A PRINT statement will now print on the screen.

Switching characters back and forth between screen and printer by CMD and PRINT # *n* statements changes the meaning of PRINT, but does not change the meaning of PRINT#. It always sends something to the printer.

Line 130 prints on the screen. The screen is now active. The printer is no longer substituted for the screen.

Line 140 uses a PRINT# statement to send data to the printer. A PRINT# statement always sends data to the printer, whether or not the printer has been substituted for the screen. The semicolon at the end of line 140 holds the print head in that line on the paper.

Line 150 finishes the sentence, on the same line. Line 160 prints a blank line on paper. If the printer were substituted for the screen, it would switch output back to the screen. At this point in the program, the printer is not substituted for the screen. PRINT # 4 just prints a blank line on paper.

Lines 170 and 180 demonstrate that CHR\$(16), followed by a two-digit number in quotation marks, tabs the print head.

Run the program. When it stops printing, rotate the black knob at top right on the printer away from you. That advances the paper so you can see the last item printed. Never use that control to move the paper in the other direction.

Instead of putting the tab number in one set of quotation marks and the following item to be printed in another set, you can put both in the same set. Line 170 would work the same if it were written PRINT # 4,CHR\$(16)"05C". That's why you must use two digits for the number of columns to be tabbed. CHR\$(16) gets the next two characters as a tab value even if they are included in a string to be printed.

Change line 170 to read that way and run the program.

List the program. This time, it lists on the screen because the printer is not substituted for the screen. There is one more thing to do. Enter

```
190 CLOSE 4
```

Now you have a complete program. The printer was opened and used. Part of the time, it was substituted for the screen. Then it was unsubstituted. It was used again to print as a file. Then the file was closed so that file number is released.

SUMMARY OF PRINTER COMMANDS

OPEN <i>file n, device n, mode n</i>	Mode 0 is upper case and graphics Mode 7 is upper and lower case
CLOSE <i>file n</i>	Closes printer and releases file number.
CMD <i>file n</i>	Substitutes printer for screen.
PRINT# <i>file n</i>	Switches characters from printer back to screen. If printer has been substituted for screen. Otherwise, prints blank line on paper.
PRINT# <i>file n, string or numeric</i>	Prints characters on paper.

SETTING TOP OF FORM

If you are printing on a page and want the lines to begin near the top of the page, you must position the paper in the printer before starting to print. Do that by rotating the black knob at top right to advance the paper. Do not attempt to move the paper backward.

A good setting is to place a continuous-form horizontal perforation just below the tractors. This is called setting *top of form*.

SKIPPING PERFS

If you print a document with more than 66 lines, it will not fit on one page. Usually, you will prefer not to print across the perforations that divide one page from another. Instead, you will leave a bottom margin on the first page, advance the paper past the *perfs*, advance a line or two on the following page, and then resume printing.

The method of doing that is identical to the method used to control scrolling on the screen. Begin by setting top of form. Then use the program to count each line on the paper as it is printed, including blank lines.

When the line count is getting close to 66, perhaps at line 62 or 64, stop printing characters. If you stop at line 62, print six blank lines. That will leave four blank lines on the first page and two blank lines at the top of the second page. Then reset the line counter and resume printing characters and counting lines. That is called *skipping the perfs*.

When page 2 has been printed, repeat the "skip perfs" operation. You can continue that indefinitely. The top and bottom margins of page 2 and all following pages will be identical.

NUMBERING PAGES

If you use a routine to skip over the perfs, you can also print page numbers on each page, using a page-number counter in the program. You can print the page number at the bottom on one page, before advancing to the next page. Or you can print it at the top of each page.

PRINTER BUFFER

The printer has a buffer that holds characters to be printed. If you interrupt program execution with characters in the buffer, they will be printed as part of the next group of characters printed.

You can prevent that by turning off the printer and then on again before printing. Turning off the printer empties the printer buffer.

PRINTER CONTROL CODES

There are ASCII control codes to control the printer. They are shown in the accompanying table.

PRINTER CONTROL CODES (These codes are executed by a PRINT# n statement.)	
Code	Effect
CHR\$(8)	Enter graphic mode
CHR\$(10)	Line feed
CHR\$(13)	Carriage return (line feed plus cursor fully left)
CHR\$(14)	Select double-width characters
CHR\$(15)	Select standard-width characters
CHR\$(16)	Tab print head (followed by "tab number")
CHR\$(17)	Select upper and lower case
CHR\$(18)	Select reversed characters
CHR\$(26)	Repeat graphics dot pattern
CHR\$(27)	Used with dot address
CHR\$(145)	Select upper case and graphics
CHR\$(146)	Cancel reversed character setting

The graphic mode selected by CHR\$(8) is dot graphics. CHR\$(26) and CHR\$(27) relate to the dot-graphics mode. These control codes are not discussed in this chapter. The other control codes will be demonstrated.

The printer must be opened either in the upper-case-and-graphics mode or the upper-and-lower-case mode. After it is open, you can use control codes 17 and 145 to switch back and forth between those modes.

The Commodore printer booklet refers to CHR\$(17) as the "Cursor Down" mode because printing CHR\$(17) moves the cursor down one line *on the screen*. Cursor Down has no meaning when applied to the printer. On the printer, CHR\$(17) selects the upper-and-lower-case character set.

The Commodore printer booklet refers to CHR\$(145) as the "Cursor Up" mode because printing CHR\$(145) moves the cursor up one line *on the screen*. The name Cursor Up has no meaning when applied to the printer. On the printer, CHR\$(145) selects the upper-case-and-graphics character set.

DISPLAYING PRINTER FONTS

In the printing industry, a character set is called a *font*. The printer has four fonts. One is upper case and graphics. Another is upper and lower case. They are the same as the two character sets that can be displayed on the screen. Fonts 3 and 4 are similar to the same as the first two, except that they are double-width.

In addition, the printer can be set to print reversed characters. Black and white are reversed in each character rectangle. Perhaps that makes eight fonts.

This program will print all four fonts, not reversed. Those are all of the standard fully formed characters that can be printed. It will also skip the perfs and number the pages. Enter

```
NEW
10 REM PRINT FOUR FONTS
20 PRINT CHR$(147)
30 PRINT "PLEASE SET TOP OF FORM,"
40 PRINT "THEN PRESS RETURN."
50 GET A$:IF A$=" "GOTO 50
90 REM ----- PRINT HEADER
100 OPEN 4,4,0:REM UPPER CASE
110 PRINT # 4,CHR$(16)"15 PRINTER FONTS"
120 PRINT # 4
130 PRINT # 4, "CODE" CHR$(16)"08";
140 PRINT # 4, "UPPER" CHR$(16)"19";
150 PRINT # 4, "WIDE" CHR$(16)"28";
160 PRINT # 4, "LOWER" CHR$(16)"39";
170 PRINT # 4, "WIDE"
180 LC=9:REM SET LINE COUNTER
185 PC=1:REM SET PAGE COUNTER
190 REM ----- PRINT CHARACTERS
```

Test that segment by running the program. It should print the title and four column heads. Enter

```
200 PRINT # 4
205 FOR C=32 TO 255
210 REM EXCEPTIONS
215 IF C=34 THEN PRINT # 4,C,
    " (QUOTATION MARKS)":GOTO 260
```

(Program continued on next page.)

```
220 IF C=145 THEN PRINT # 4,C:GOTO 260
```

```
225 IF C=146 THEN PRINT # 4,C:GOTO 260
```

Line 205 sets up a loop to print the characters. The ASCII code numbers recognized by the printer are 0 to 255. Some are printer-control codes rather than printable characters. By starting the loop at code 32, control codes below 32 are omitted. This avoids the problem of whatever they do. For example, CHR\$(13) would put an undesired carriage return in the table being printed.

CHR\$(34) is a problem because it prints quotation marks and puts the computer in the insert mode. Rather than try to trick it into not doing that, the program just prints a notice that CHR\$(34) is quotation marks.

CHR\$(145) and CHR\$(146) are control codes that are in the range of printable characters. Lines 220 and 225 just print the code numbers and jump to the bottom of this routine. This prevents those control codes from being executed and causing problems in printing the table. Now, the program can start printing characters. Enter

```
230 PRINT #4,C;:REM CODE NUMBER
```

```
235 PRINT # 4,CHR$(16)"10" CHR$(C);:
```

```
    REM UPPER CASE
```

```
240 PRINT # 4,CHR$(14)CHR$(16)"20"
```

```
    CHR$(C);:REM WIDE UPPER
```

```
245 PRINT # 4,CHR$(15)CHR$(17)
```

```
    CHR$(16)"30"CHR$(C);:REM LOWER CASE
```

```
250 PRINT # 4,CHR$(14)CHR$(17)
```

```
    CHR$(16)"40"CHR$(C);:REM WIDE LOWER
```

```
255 PRINT # 4,CHR$(15)CHR$(145):
```

```
    REM RETURN TO UPPER CASE STD WIDTH
```

```
260 IF LC/64-INT(LC/64)=0 THEN GOSUB 500
```

```
270 LC=LC+1
```

```
280 NEXT C
```

```
290 END
```

Line 230 prints the ASCII code number in the first column. Lines 235-250 switch to the correct font and print the remaining four columns of the table. Each time, the value of C is printed as CHR\$(C). Line 255 resets the upper-case-and-graphics font with standard character width so the next line on the table will begin with that font. Test by running the program until it prints the headers and eight or nine lines of the table itself. Then break out of the program. Enter the subroutine that skips perfs and numbers pages.

```
490 REM ----- SUBR SKIP & NUMBER
```

```
500 FOR SKIP=1 TO 4:PRINT # 4:NEXT SKIP
```

```
510 PC=PC+1:REM PAGE COUNT
```

```
520 LC=2:REM RESET LINE COUNT
```

```
530 IF PC>1 THEN PRINT # 4,CHR$(16)"65"
```

```
    "PAGE"PC:LC=LC+1
```

```
540 RETURN
```

Run the program. If it seems to be running correctly, I suggest that you let it print the complete table. It will be useful later, when you want to look up an ASCII code number to see what it prints.

IMPROVEMENTS

This table would be more useful if it listed all of the code numbers, instead of starting at 32. Then you could use it to look up both control codes and printable codes. For the experience, rewrite the program to do that.

For codes that do nothing, leave the table blank. For control codes, print a message that says what they do. This will be similar to what the program does at code 34.

When a table uses several pages, it is usually a good idea to repeat the column heads on each page. This can be done in the subroutine. Instead of printing just the page number, rewrite the subroutine so it prints the column heads and the page number.

Notice that this program prints near the left margin of the page. You may wish to center the table on the page, so it will look better. If you put it in a notebook, moving the table to the right will allow room to punch holes in the pages to fit a ring binder.

A well-written program will not stop in the middle of a page when the program ends. It should advance to the top of the next page. Then, if the user prints something else, it begins on a new page, rather than ruining the last page of the data just printed. Figure out a way to do that and write it into the program.

EXPERIMENTAL PROGRAMMING

When you use a command or statement that you have not used before, there are often surprises. You haven't used CHR\$(18) yet. It sets the printer to make reversed characters.

Save the program in memory. Then make a new program that displays the four fonts with reversed characters. Begin by changing the title to REVERSED PRINTER FONTS. That's the easy part.

Use CHR\$(18) to print the title and column heads in reversed type. Then print the entire table in reversed type. When you test, you will observe that the printer returns to normal type when you don't expect it to. Force it back into the reversed mode by putting PRINT # 4,CHR\$(18) into the program as often as necessary. You will see what is resetting the printer to make normal type.

LOCKUP PROBLEMS

When you are programming and testing programs with the printer turned on, you may occasionally lock up the computer. The cursor may disappear or stop blinking. The computer will not respond to keyboard commands.

If this happens, you have probably sent a command to the printer that it cannot execute. The computer is waiting for the printer to execute an impossible command.

You have two choices. You can turn off the computer and start over again, or you can turn off and then turn on the printer. I prefer to turn off the printer.

USING THE PRINTER AS A TYPEWRITER

The printer is a typewriter without a keyboard. A keyboard is on the computer. They are connected by a cable. It is sometimes handy to type on the keyboard and print what you type.

Here is a way to do that. Save the program in memory, if you wish. Then enter

```
NEW
10 REM TYPEWRITER
20 PRINT CHR$(147)
30 PRINT CHR$(14):REM U&LC
40 OPEN 4,4,7:REM U&LC
50 S$=" ":INPUT S$
60 PRINT # 4,S$
70 GOTO 50
```

That's a bare-bones program, but it works. It doesn't display instructions to the user. It's an endless loop. To get out of it, press RUN STOP-RESTORE. It doesn't skip over the perfs and number pages. A program like that might be OK for a programmer to use to type reminders to himself.

A common way to get out of a loop that has an INPUT statement is to ask the user to enter a special word to end the program. For example, end it when the input is the word *end*.

Run the program. Enter this reminder: Fix this program!

REVIEW

After you have used a printer for a while, you will think of it as an essential part of a computer system. Printers are useful to list programs—especially long programs that don't work and need fixing.

Sometimes, the best way to find a problem in a program is to print out the values of variables as the program runs. Then, you have a record of what happened. That makes it easier to figure out what went wrong.

A printer is useful to print out reports and documents from a program and to serve as a typewriter.

This chapter demonstrated most of the things that people do with printers: listing programs, printing in the immediate mode, printing from a program, using a variety of fonts.

I suggest that you review this chapter and look over the demonstration programs to be sure you understand them. If you didn't do the program modifications suggested, this is a good time to do it. The best way to learn to use a printer is to use it.

18 Binary And Logical Operations

This chapter discusses the numbering system used inside the computer—binary numbers. Then it shows how these numbers are used to execute program statements such as comparisons. It shows ways to manipulate and change binary numbers in memory. This is an advanced and powerful way to control the computer.

This chapter is mainly mathematics. It prepares you for the following two chapters, which discuss advanced graphics and sound. The information in this chapter is not essential unless you want to go beyond ordinary BASIC programming.

NUMBERING SYSTEMS

Three numbering systems are commonly used with computers: ordinary decimal numbers, *binary* numbers and *hexadecimal* numbers. To understand the last two, you must first understand the decimal-numbering system. The problem is that most of us know the decimal-numbering system so well that we don't know it at all.

DECIMAL NUMBERING SYSTEM

Decimal means *ten*. There are 10 numbers in the decimal system, 0 through 9. Because the decimal system is based on 10 numbers, we say that the number 10 is the *base* for the decimal numbering system.

In the number 1234, each digit has a value determined by two things: the value of the digit itself and its position in the number. I will refer to these as the *digit value* and the *position value*.

The number 1234 has an implied or actual decimal point after the 4. From the decimal point to the left, the position values are: 1, 10, 100, 1000 and so forth. You probably learned position values in school as units, tens, hundreds, thousands and so forth.

The *total value* of each digit is its digit value multiplied by its position value. The total value of the rightmost digit is 4 times 1, which is 4. The total value of the next digit is 3 times 10, which is 30.

The total value of the number 1234 is: one thousand plus 200 plus 30 plus 4. Everybody knows that without even thinking about it. When we think about it, it becomes difficult.

This table shows how the total value of the entire number is determined using digit values and position values.

Digit Value	x	Position Value =	Total Value
4		1 (rightmost digit)	4
3		10	30
2		100	200
1		1000	1000
TOTAL VALUE OF NUMBER =			1234

The total value of the entire number is the sum of the total values of each of the digits. The total value of each digit is its digit value multiplied by its position value.

The number 1234 has four positions to the left of the decimal point. The positions are numbered. The first position to the left of the decimal point is *position zero*. Position numbers are shown here.

NUMBER 1234
POSITIONS 3210

Soon, you will see why position numbers are important.

The phrase *powers of ten* means that the number 10 is raised to several different powers or *exponents*. The notation is $10 \uparrow X$, in which X is the power or exponent. For example, $10 \uparrow 2$ means 10 to the second power, or 10 squared, or 10 times 10. The exponent is 2.

Here are several powers of 10. The base number is 10. Notice that the numbers follow a pattern:

$10 \uparrow 4 = 10 \times 10 \times 10 \times 10 = 10000$, which is 1 followed by 4 zeros.
 $10 \uparrow 3 = 10 \times 10 \times 10 = 1000$, which is 1 followed by 3 zeros.
 $10 \uparrow 2 = 10 \times 10 = 100$, which is 1 followed by 2 zeros.
 $10 \uparrow 1 = 10$, which is 1 followed by 1 zero.
 $10 \uparrow 0 = 1$, which is 1 followed by 0 zeros.

You probably agree with the first three lines in this example with no mental difficulty. You must agree with the bottom two as an act of faith—because they fit the pattern. Or else, you can accept these mathematical definitions: $10 \uparrow 1 = 10$ and $10 \uparrow 0 = 1$.

Those definitions apply to any number. Here are two general definitions, using N to represent any number. The definition $N \uparrow 1 = N$ says that any number raised to the power of 1 is not changed in value. For example, $177 \uparrow 1 = 177$.

The definition $N \uparrow 0 = 1$ says that any number raised to the zero power has a value of 1. For example, $177 \uparrow 0 = 1$. What follows is based on those two general definitions.

In a numbering system using position values, the value of each position is the base number raised to the position number. The following table illustrates that, using 10 as the base. It shows position value in two ways—as powers of ten and as ordinary numbers.

POSITION NUMBER	4	3	2	1	0
POSITION VALUE	$10 \uparrow 4$	$10 \uparrow 3$	$10 \uparrow 2$	$10 \uparrow 1$	$10 \uparrow 0$
POSITION VALUE	10000	1000	100	10	1

This brings us back to the beginning of the discussion of the decimal numbering system. From the left, position values are units, tens, hundreds and so forth.

What has been done so far is to show the method of determining position values and the method determining the total value of a number in the decimal system.

BINARY NUMBERING SYSTEM

Individual memory elements in a computer can have only two conditions—full or empty. We think of these conditions as on and off, or 1 and 0. Computer elements cannot count from 0 to 9, as in the decimal system, but they can count from 0 to 1.

A numbering system that uses only two numbers, 0 and 1, is called *binary*. The binary numbering system is “natural” for computers.

Binary numbers look like this: 10110111. The digits can be only zeros or ones.

Using exactly the same method as before, here are position values for a binary numbering system using 2 as the base. All numbers in this table are decimal numbers. I am using decimal

numbers to describe the binary numbering system.

POSITION NUMBER	4	3	2	1	0
POSITION VALUE	2↑4	2↑3	2↑2	2↑1	2↑0
POSITION VALUE	16	8	4	2	1

To find the total value of any digit, multiply the digit value by the position value. Because position values are decimal numbers, this expresses the total value of a binary digit as its decimal-number equivalent. That makes it understandable to people.

Here is an example. It converts the binary number 1101 into its decimal-number equivalent.

Digit Value	x	Position Value =	Total Value
1		1 (rightmost digit)	1
0		2	0
1		4	4
1		8	8
DECIMAL VALUE OF 1101 =			<u>13</u>

The computer recognizes binary numbers without converting them to decimal. The computer sees a difference between 1101 and 1110. We can see the difference also, but we feel better knowing that 1101 is the same as decimal 13 and 1110 is decimal 14.

Magic Numbers—The position values for binary numbers are all multiples of 2, except the first one. The series is: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 and so forth. Each of these numbers is the sum of all smaller numbers in the series, plus one.

HOW TO ADD BINARY NUMBERS

Adding binary numbers is done in a way similar to adding decimal numbers. With decimal numbers, if the total of a column exceeds 9, we “carry” a digit to the next column to the left. For example,

$$\begin{array}{r} 8 \\ +2 \\ \hline 10 \end{array}$$

With binary numbers, $0+0=0$, $1+0=1$, and $1+1=0$ with a carry. Here is an example:

$$\begin{array}{r} 0001 \\ +0001 \\ \hline 0010 \end{array}$$

Binary 0010 is equivalent to decimal 2.

By starting with binary 0000 and adding 0001 repeatedly, you can generate the first 16 binary numbers. Please do the indicated additions in the accompanying table mentally. Notice how the pattern of digits changes each time 0001 is added to the preceding value. It will be helpful to learn to recognize these patterns and mentally translate them into their decimal equivalents.

BITS AND BYTES

Digits in a binary number are called *bits*. A binary number with 8 bits is called a *byte*.

A four-bit binary number can range from 0000 to 1111. Using all possible combinations of four binary bits, you can count from decimal 0 to decimal 15, which is 16 different values.

Because four binary digits are half of a byte, some people call it a *nybble*.

Using pencil and paper, calculate the position values for an eight-digit binary number. Add them up. The result is 255. All possible combinations of eight bits can produce 256 different values. They are 0 to 255.

HOW KEYBOARD CHARACTERS ARE STORED IN MEMORY

In computer memory, each tiny memory element can hold one bit. Eight of these elements are connected together as a unit to hold eight bits. This eight-bit unit is called a *memory location*. Each memory location holds one byte.

Every character on the keyboard is represented by an ASCII code number ASCII code

ADDING BINARY NUMBERS	
Binary	Decimal Equivalent
0000	0
<u>+0001</u>	
0001	1
<u>+0001</u>	
0010	2
<u>+0001</u>	
0011	3
<u>+0001</u>	
0100	4
<u>+0001</u>	
0101	5
<u>+0001</u>	
0110	6
<u>+0001</u>	
0111	7
<u>+0001</u>	
1000	8
<u>+0001</u>	
1001	9
<u>+0001</u>	
1010	10
<u>+0001</u>	
1011	11
<u>+0001</u>	
1100	12
<u>+0001</u>	
1101	13
<u>+0001</u>	
1110	14
<u>+0001</u>	
1111	15

numbers range from 0 to 255, in decimal. Each ASCII code number, and therefore each character, is converted to binary by the computer and stored in one byte at one location in memory.

The largest decimal number that can be stored in one byte is 255, which is binary 11111111. That's why there are no ASCII code numbers greater than 255.

USES OF BINARY

If you understand binary numbers, you can do more with your computer. For internal operations, the computer uses some memory locations to control eight different things. Each location holds one byte and each byte has eight bits. One bit controls one thing.

If the computer itself uses a certain bit to control something, and you want to control it from the keyboard or a program, you must find a way to change that one bit. That is called *bit addressing*, *bit mapping*, *bit-wise programming* and similar names.

DO YOU NEED BINARY?

You can do a lot of good programming without knowing anything about binary or how to use it. Controlling individual bits in the computer is mainly for fancy things such as making your own screen displays. If you are satisfied with the characters and graphics symbols you have already seen in this book, you don't need to know much about binary numbers.

At this point, you can make a decision. If you really want to learn binary numbers, spend some time with pencil and paper to become thoroughly familiar with them. Write down binary numbers and convert them to decimal until you can do it quickly and without difficulty. Memorize position values up to 128.

If you are not sure, gather the general ideas now and come back to them later, if you become more interested in binary and bits.

BINARY-DECIMAL CONVERSION

Converting a binary number to decimal, using pencil and paper, is tedious. It's easy to make an error. Here is a routine that will do it for you. It uses the same method—adding up the position values for each position where the bit value is 1. Enter

```
NEW
8000 PRINT CHR$(147):REM BINARY TO DEC
8010 INPUT"ENTER 8-BIT BINARY NUMBER";N$
8020 IF LEN(N$) < > 8 GOTO 8000
8030 D=0:REM DECIMAL VALUE
8040 X=7:REM EXPONENT
8050 FOR I=1 TO 8:REM READ BITS
8060 IF MID$(N$,I,1)="1" THEN D=D+2^X
8070 X=X-1:NEXT
8080 PRINT:PRINT"DECIMAL  ="D:END
```

Line 8020 is an error trap. Line 8030 initializes the variable D to zero. It will be used to hold the decimal value of the binary number. Line 8040 initializes X to 7. X is the exponent used to determine position values in the binary number.

Line 8050 sets up a loop to read bits in an 8-bit binary number, one bit at a time. Line 8060 uses a MID\$() statement to do that. Bits are read from N\$ one character at a time, starting at the left. When I is 1, MID\$(1,1) is the leftmost character of N\$. Variable X is initialized at 7. If the leftmost bit is 1, then $D = D + 2^7$.

Line 8070 decrements X to 6, sets I to 2 and jumps back to run the loop again. If the second bit from the right end of N\$ is 1, then $D = D + 2^6$, and so forth.

Variable I counts from 1 to 8, to read the characters in N\$ from left to right. Variable X counts from 7 down to 0. This provides the correct position value for the bits in N\$.

Run the program. Enter 11111111. The program displays the decimal equivalent. Enter a variety of binary numbers, 8 bits at a time. Enter 00001101 and notice that the decimal equivalent is 13.

DECIMAL-BINARY CONVERSION

When any decimal number is divided by 2, the remainder is either 1 or 0. To convert a decimal number to its binary equivalent, divide repeatedly by 2 and use the remainders to form the binary equivalent, starting with the rightmost bit.

Here is an example using the decimal number 13. When 13 is divided by 2, the result is 6 with a remainder of 1. Use the remainder to form the rightmost character of the binary equivalent. Use the 6 for the next division by 2.

The complete conversion is shown here:

13/2 is 6 with a remainder of 1	Rightmost digit is	1
6/2 is 3 with a remainder of 0	Right two digits are	01
3/2 is 1 with a remainder of 1	Right three digits are	101
1/2 is 0 with a remainder of 1	Right four digits are	1101
0/2 is 0 with a remainder of 0	Right five digits are	01101
0/2 is 0 with a remainder of 0	Right six digits are	001101
0/2 is 0 with a remainder of 0	Right seven digits are	0001101
0/2 is 0 with a remainder of 0	Right eight digits are	00001101

Make a few conversions from decimal to binary, using other decimal numbers, until you are sure it works. Convert decimal 197 to binary. If you get 11000101, you are doing it correctly.

Here is a routine to do decimal-binary conversions using the same method. Enter it without typing NEW so the preceding routine remains in memory.

```

10000 PRINT CHR$(147):REM DEC TO BINARY
10010 INPUT"ENTER DEC NUMBER, 0-255";N%
10020 IF N%<0 OR N%>255 GOTO 10000
10030 FOR I=1 TO 8
10040 HALF%=N%/2
10050 BIT$(I)=STR$(N%-(2*HALF%))
10060 N%=HALF%
10070 NEXT
10080 BINARY$=""
10090 FOR I=8 TO 1 STEP -1
10100 BINARY$=BINARY$+BIT$(I)
10110 NEXT
10120 PRINT:PRINT"BINARY ="BINARY$:END

```

Line 10020 is an error trap. The loop at line 10030 repeatedly divides by 2. Line 10040 divides N% by 2 and names the result HALF%, which is an *integer variable*. If N% is 11, HALF% will be 5.

Line 10050 finds the remainder and names it BIT\$(I). The remainder is found by the expression (N%-(2*HALF%)). If N% is 11, HALF% is 5 and (11-(2*5)) is 1. The remainder is 1.

Remainders are placed in an array, using the array name BIT\$(I). BIT\$(1) is the *first* remainder produced by the routine, and it is the *rightmost* bit in the binary equivalent. BIT\$(8) is the last bit formed by this routine but it is the leftmost bit in the binary number.

After all eight bits have been formed, the loop at line 10090 reverses their order while concatenating them to form the variable BINARY\$. This loop counts from 8 down to 1. BINARY\$ starts out as "". The first bit concatenated onto it is BIT\$(8). The last bit to be concatenated is BIT\$(1). This puts the bits in correct order in the binary number. Test by entering 255.

USING THE CONVERSION ROUTINES

Now you have two conversion routines in memory beginning at lines 8000 and 10000. The line numbers are memory aids. When you need to convert a binary number, run 8000 and enter 8 bits. To convert a decimal number, run 10000 and enter the decimal number.

These routines can help you write programs requiring you to make conversions between decimal and binary. Begin with these two routines in memory. Then write the program at lower line numbers, beginning at line 10.

When the program has been written, delete these routines, unless you actually used them in the program itself. Keep these routines in memory. You will use them later in this chapter.

LEFT AND RIGHT SHIFT

Compare the binary numbers 00000001 and 00000010. The difference is that the bit with value 1 has been moved one place to the left. That is called *shifting* a bit to the left. For this discussion, it doesn't matter how it is shifted.

The value of 00000001 is 1 and the value of 00000010 is 2. Shifting a bit 1 place to the left multiplies the value of the binary number by 2, which is 2^1 . Shifting a bit 2 places to the left multiplies the number by 2^2 . Shifting 5 places to the left multiplies the number by 2^5 . Shifting any number of places to the left, using N to represent the number of places, multiplies the number by 2^N .

Please verify that by experiment. Write down some binary numbers, shift all of the bits one or more places to the left and compare the decimal equivalents.

Shifting bits any number of places to the right divides the number by 2^N . Verify that also.

A very important left shift is 8 places. $2^8 = 256$. Shifting a binary number 8 places to the left multiplies its value by 256.

Here are 16 binary digits: This is two bytes placed end to end. The space between is just to help you see where one byte ends and the other begins—00000000 00000001. The left byte has no value. Consider those 8 zeros as place markers. The value of the right byte is 1.

Now, let's do an 8-bit left shift. The number becomes 00000001 00000000. By the left-shift rule, the value of the left byte is 256 times what it was before it was shifted.

Now I am going to take out the space, so it is all one binary number: 0000000100000000. Using pencil and paper, calculate position values and find the value of that number. It is 256.

STORING A NUMBER IN TWO BYTES

The largest number that can be stored in one byte is decimal 255, which is 11111111. Adding binary 1 to that number makes it 100000000, which is the same as 0000000100000000. You just evaluated that as 256.

That 16-bit number cannot be stored in one byte. It is stored in two bytes. Here is the space again, to divide the bytes: 00000001 00000000. When a number is stored in two bytes, the bytes are called *low byte* and *high byte*. In this example, 00000001 is the high byte and 00000000 is the low byte.

Evaluating the low byte, in decimal, is fairly easy. You have been doing it in this chapter. The routine at line 8000 does that.

How do you evaluate the high byte? The easy way is to evaluate it as if it were a low byte and then multiply by 256.

When you have evaluated low byte and high byte, as though both were low bytes, you have separate values for the two bytes. The total value of a two-byte number is the sum of the values of the two bytes. If you evaluated the high byte as though it were a low byte, that value is too small. The correct value is 256 times as great. Here is a handy recipe to evaluate a two-byte number:

$$\text{Value} = \text{Low byte} + (256 \times \text{High byte})$$

Of course, you can always evaluate two bytes as one 16-bit binary number. Calculate position values for all 16 bits and proceed in the usual way.

It is much easier and more useful to use the recipe. Let's find the largest number that can be stored in two bytes. That number in binary is 11111111 11111111. The low-byte value is 255. The high byte, if it were a low byte, is also 255. Using the handy recipe,

$$\begin{aligned}\text{Value} &= \text{Low byte} + (256 \times \text{High byte}) \\ &= 255 + (256 \times 255) \\ &= 255 + 65280 \\ &= 65535\end{aligned}$$

The largest number that can be stored in two bytes is 65535. With two bytes, there are 65536 different numbers. They are 0 to 65535.

As a programmer, you sometimes have to do it the other way. You know the decimal value of a number to be stored and need to know the individual values for low byte and high byte so you can

poke them into memory locations. Here is a method of doing that:

High byte = $\text{INT}(\text{Value}/256)$

Low byte = $\text{Value} - (256 \times \text{High byte})$

The high byte is the number of times that 256 divides evenly into the value to be stored. The low byte is the remainder. If you don't see that immediately, do it with pencil and paper. Example:

Value to be stored = 555

High byte = $\text{INT}(555/256)$

High byte = 2

Low byte = $555 - (256 \times 2)$

Low byte = 43

In decimal, high byte is 2 and low byte is 43. To get the total decimal number represented by those two bytes,

Value = Low byte + $(256 \times \text{High byte})$

= $43 + (256 \times 2)$

= 555

To see what those two bytes look like as binary numbers, use routine 10000 to convert each byte in decimal to its value in binary. With high byte on the left, the two bytes are 00000010 00101011

HEXADECIMAL NUMBERS

This is similar to the binary numbering system. It is a way of writing a binary number using fewer digits. I am not going to ask you to learn very much about it.

A measure of efficiency of a numbering system is the number of digits needed to state a value. In that sense, binary is the least-efficient system possible. The only system less efficient than binary would be a *unary* system. It has only one number, such as zero. It doesn't work at all, except to count to zero.

Decimal is more efficient than binary because decimal has 10 numbers. In binary, it takes four digits to count to 15—they are binary 1111. In decimal, four digits can count to 9999.

The *hexadecimal* numbering system uses base 16—*hexa* means six, *decimal* means 10. Because the base is 16, it has some of the same position values as binary, such as 256 and 65536. It is a close cousin to binary and therefore suitable for communicating with computers in a language they can understand.

The main problem in hexadecimal is finding 16 symbols to represent the 16 numbers. For people, this is startling. The 16 hexadecimal numbers are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

In hexadecimal, the decimal-number equivalent of A is 10. B is 11, and so forth. The theory is the same as decimal or binary. The total value of any digit is its digit value multiplied by its position value.

Position values are 16^0 , 16^1 , 16^2 , 16^3 , 16^4 and so forth. In decimal, those position values are 1, 16, 256, 4096, 65536 and so forth.

As you can see, hexadecimal gets up to big numbers in a hurry. Hexadecimal numbers are used by programmers, usually when writing programs in *assembly language*. It is shorthand for binary. For example, the number 65535 in binary is 1111111111111111. That number in hexadecimal is FFFF.

LOGICAL OPERATORS

Commodore BASIC has three logical operators: AND, OR and NOT. Logical operators are sometimes called *Boolean operators* after a mathematician named Boole.

AND and OR are used in IF-THEN statements according to their meanings in English. You have seen examples of that.

LOGICAL AND

Logical operators are also used with binary numbers to do a sort of arithmetic on them. The AND operator works with two bits at a time, following a set of rules that are illustrated by this table:

1 AND 1 = 1
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0

That table is called a *truth table*. The number 1 means *true* and 0 means *false*. It may help to remember that if you think of 0 as an empty promise—which is false.

The first line in the table says, *true AND true = true*. If you are considering two things together, one must be true AND the other must be true, for their combination to be true. The next line says *if both are false, their combination is false*. The last two lines say, *if either is false, their combination is false*. Combining two bits, using AND, is called *ANDing the bits*.

Your computer will show you the truth table for AND. Enter

```
PRINT 1 AND 1
PRINT 0 AND 0
PRINT 1 AND 0
PRINT 0 AND 1
```

Because true and false are represented by bit values, the effect of logical AND produces a result that is a bit value. But, it can mean true or false. The essential fact about ANDing two bits is that the result is 1 only when one bit AND the other bit are both 1.

LOGICAL OR

When OR is used with two bits, the idea is the same but the rules are different. Here is a truth table for the OR operator:

1 OR 1 = 1
0 OR 1 = 1
1 OR 0 = 1
0 OR 0 = 0

The essential fact about ORing is that the result is 1 if one bit OR the other is 1. The result is 0 only if both bits are 0. Verify the truth table for OR by entering statements from the keyboard similar to those you used to see the AND truth table.

TRUE/FALSE OPERATIONS

In addition to the logical operators, the computer has a built-in lie detector. It is used to execute IF-THEN statements using relational operators to compare two values. For example, if a statement uses the expression IF $X > 3$, the computer must determine if $X > 3$ is true or false. Is X larger than 3 or not larger than 3?

Simple Comparisons—Some IF-THEN statements make only one test, using a relational operator, to decide whether or not to execute the THEN statement. If a statement says, IF $X > 3$ THEN GOTO 180, the computer must first decide if $X > 3$. Based on that determination, it then decides whether or not to jump to line 180.

You have seen IF-THEN statements like that work in programs. It works because BASIC makes it work. This discussion shows how the computer makes that decision. Let's refer to the expression $X > 3$ as a *proposition*. It may be true or false.

To test that proposition, the computer uses information in memory to determine if X is larger than 3, or not. Then it assigns a true/false value to the proposition. If it is *false*, the computer assigns the true/false value 0 to the proposition. If it is true, the true/false value -1 is assigned.

If you ask, the computer will show you the true/false value of any proposition. To establish a value for X in memory, enter

```
X=5
```

Then enter

```
PRINT X>3
```

That's a peculiar PRINT statement. What is to be printed is not in quotation marks and it is not a variable name. It is a *complete relational expression*.

The statement PRINT X>3 means *print the true/false value of the proposition X>3*. It is true, so the computer displays -1. In memory, the computer holds the value -1 as testimony that the proposition is true.

The complete program statement is, IF X>3 THEN GOTO 180. The computer now has a true/false value for X>3. The computer decides whether or not to GOTO 180, based on the true/false value it has found for the proposition. If the true/false value is -1, the statement executes and the program jumps to line 180. If the true/false value is 0, the statement does not execute.

In true/false determinations, *any number that is not zero is considered true*. In the truth tables shown earlier, the number 1 is used to signify true. Because any number that is not 0 means true, 1 and -1 have the same meaning when used as true/false indicators.

To execute a statement with simple comparisons, such as X>3 or X=3 or X\$="ABC", the computer takes two steps. First, it makes a true/false evaluation of the proposition. Then based on the resulting true/false value, it executes the THEN statement or it doesn't.

Complex Comparisons—Not all IF-THEN statements use a single test or comparison. The statement IF X<3 AND Y\$="ABC" THEN PRINT "YES" has two *propositions*. One is that X>3. The other is that Y\$="ABC". If both are true, the statement will execute and print YES. If either is false, the statement will not execute.

When an IF-THEN statement has more than one proposition, using relational operators, they must be connected logically by a logical operator—AND or OR.

To make a complex comparison of that type, the computer first looks at the proposition X>3 and assigns it a true/false value, based on the information the computer has in memory. Then the computer tests the next proposition, Y\$="ABC", and assigns it a true/false value.

The resulting true/false numbers, 0 or -1, are combined by the logical AND operator, following the rules for ANDing. A final true/false value is produced by the AND operation. The final value is used to decide whether or not to execute the THEN statement. The following section shows how that is done.

LOGICAL OPERATIONS

The statement IF X>3 AND Y\$="ABC" THEN PRINT "YES" has two relational operators and a logical operator, AND.

The logical operator AND operates on two true/false values, produced by testing the two propositions. It follows the rules for ANDing. The true/false values are either 0 or -1, reflecting the truth or falsity of the two propositions.

If the first proposition is true and the second false, the statement says IF (0 AND -1) THEN PRINT "YES". The parentheses enclose what is still to be evaluated.

The AND operator will determine the final true/false value by evaluating the expression in the parentheses, following the rules for ANDing. To see what the true/false value will be, enter

```
PRINT 0 AND -1
```

The result is 0, which means false. The *entire* IF statement has been judged false because one of the propositions is false. The THEN statement will not execute.

Suppose both statements are true. Enter

```
PRINT -1 AND -1
```

The result is -1, which means true because it isn't zero. The entire IF statement is judged true, so the THEN statement will execute.

If both propositions were false, the THEN statement would not execute. Enter

PRINT 0 AND 0

What has been shown is that relational operators generate 0 or -1 to indicate true or false. In an IF-THEN statement with more than one proposition is to be tested, logical operators combine those true/false numbers. The truth-table rules are then used to determine a final true/false value. The computer executes or doesn't execute the THEN statement, depending on the final true/false value.

Write an IF-THEN statement using OR and explore it the same way.

"DOES IT EXIST" TEST

Logical and relational operators operate on *operands*. Normally, there are two operands for each operator. In the statement IF X=Y, the relational operator is =. The operands are X and Y.

There is a special relational operation that omits both the operator and the second operand. Enter

X=1

IF X THEN PRINT "YES"

In that statement, IF X is treated as though it were IF X <> 0. I consider statements such as IF X to say, *if X exists*, meaning if it has some value other than zero. Set X to 0 and test again to see if X exists.

A statement such as IF X THEN may be confusing because it looks incomplete. It makes the program a little shorter so it will take less space in memory.

If a program is too long, you should certainly shorten it. Otherwise, it may be better to use the long form so anybody can understand it.

BIT OPERATIONS

Another use of logical operators is to change individual bits at specified locations in memory as a way of controlling actions of the computer. Some things can be done only by controlling individual bits.

When logical operators are used this way, they are not performing true/false evaluations, they are changing bits following truth-table rules.

Even though you may intend to change only one bit at a memory location, you must send an entire eight-bit byte to that location.

There are several ways to identify the bits in an eight-bit byte. The *least-significant* bit is at the right, at position 0. The least-significant bit has a position value of 1.

The *most-significant bit* is at left. It is at position 7 and has a position value of 128. The most-significant bit contributes more value to the binary number than the least-significant bit.

When dealing with bits, *bit value* means the same thing as *position value*. That's because the bit can be only 1 or 0. If it is 1, the bit value is the position value. If 0, its value is zero. Bit 7 has a bit value of 128—if it is 1.

We usually say *bit number* instead of *bit-position number*. It's easy to become confused when counting bits, because we sometimes forget that the bit at the right is bit 0. To avoid that difficulty, remember that bit 7 is the one with a bit value of 2⁷. Bit 0 has a bit value of 2⁰. The accompanying table illustrates those definitions.

CHANGING ALL BITS IN A BYTE

Suppose you want to set bit 4 at memory location 3333 to 1 and all other bits to zero. The bit pattern you must send to that location is 00010000, which is one byte. Remember that the rightmost bit is bit 0.

Sending a bit pattern to a memory location is done by a POKE statement. Even though the computer munches on binary numbers in private, it wants decimal numbers from the keyboard. POKE the decimal number equivalent to the binary number 00010000. To make the conversion,

ANATOMY OF A BYTE								
BIT POSITION NUMBER	7	6	5	4	3	2	1	0
BIT NUMBER (SAME AS EXPONENT)	7	6	5	4	3	2	1	0
POSITION VALUE (AS POWER OF 2)	2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
POSITION VALUE (AS DECIMAL)	128	64	32	16	8	4	2	1
BIT VALUE (IF BIT IS 1)	128	64	32	16	8	4	2	1

run 8000 and enter the binary number. The decimal equivalent is 16.

The statement `POKE 3333,16` will put the bit pattern 00010000 into location 3333 because 16 is the decimal equivalent of that bit pattern.

USING LOGICAL OR TO SET BITS TO 1

In the preceding demonstration, bit 4 was set to 1 by sending an entire byte to a memory location. All other bits except bit 4 were zero in the byte that was sent to memory. If those bits in that memory location were ones before, they are zeros now.

Sometimes you will want to set one or more bits to 1 at a memory location and *not change the other bits*. You may not know what the other bits are, but you don't want to change them—whatever they are.

That is done with an OR operation. Figure out a bit pattern that will have the desired result and then OR that bit pattern with the bit pattern already in memory. Here is an example that sets bit 4 to 1, assuming that the bit pattern already in memory is 10000001:

BIT PATTERN IN MEMORY	10000001	(DECIMAL 129)
BIT PATTERN TO OR	00010000	(DECIMAL 16)
RESULT	10010001	(DECIMAL 145)

In the bit pattern in memory, bit 4 is 0. In the bit pattern to be ORed, bit 4 is 1 and all other bits are 0. That will change bit 4 in memory to 1 and not change any of the other bits.

An OR operation requires two operands. One is the bit pattern in memory. The other is the bit pattern to be ORed.

The OR operation takes one bit from each operand, at the same positions, and ORs them to get the result. The rule for a logical OR operation is that the result is 1 if one bit OR the other bit OR both bits are 1.

Mentally OR those two operands to verify the result shown. Starting at the right, $1 \text{ OR } 0 = 1$. Then, $0 \text{ OR } 0 = 0$. Continue ORing pairs of bits through the entire byte. You can start at either end and the result is the same.

If you use an OR operation to set a bit to 1, and the bit already has a value of 1, it will remain set to 1 because $1 \text{ OR } 1 = 1$.

Demonstration—To demonstrate using OR to set a bit to 1 at a memory location, we will use location 3333. That is in the memory area used to store programs, so it will do no harm to experiment at that location.

To begin, put the binary number 10101010 at that location by a POKE statement. Poking is done with decimal numbers, so you need to know the decimal equivalent of that binary number. It is 170. Use the routine at 8000 to verify that. Enter

```
POKE 3333,170
```

Verify that location 3333 now holds the desired binary number by entering

```
PRINT PEEK(3333)
```

Decimal 170 is shown. What is actually at that memory location is the 8-bit binary number 10101010. Now, let's change bit 4 of that binary number to 1. It is now 0. Remember that bit 0 is the rightmost bit.

An OR operation is used to set bits to 1. The operand to set bit 4 to 1 is 00010000, which has a decimal value of 16.

To write the OR operation as a program statement, you must use the decimal equivalent of the bit pattern to be ORed. Enter

```
POKE 3333, PEEK(3333) OR 16
```

The PEEK statement has the value stored at 3333. The number 16 is ORed with that value. Then the result is poked back into location 3333. If that worked, the bit pattern at location 3333 is now 10111010, instead of 10101010. Enter

```
PRINT PEEK(3333)
```

Decimal 186 is at that location. To see that value in binary, run 10000 and enter 186. Bit 4 has been set without changing any of the other bits.

In decimal, location 3333 was changed from 170 to 186. In binary, it was changed from 10101010 to 10111010.

Here are some practice exercises:

Set location 3333 to binary 11111111.

Set location 3333 to binary 00000000.

Set bits 3, 4, and 5 at location 3333 to 1 without changing the other bits. The result should be decimal 56 at that location. Verify that.

USING LOGICAL AND TO SET BITS TO 0

Logical AND is used to set bits to 0 without changing any other bit. With the AND operator, the result is 1 only if *both* operands are 1. If either bit is 0, the result is 0. By ANDing any bit in memory with 0, the result is 0, no matter whether the bit was originally 1 or 0.

ANDing 1 with any other value does not change the other value because the truth table for AND says that $1 \text{ AND } 1 = 1$. The truth table also says that $0 \text{ AND } 1 = 0$.

If a number is 10010001, and you want to set bit 4 to 0, use 11101111 as the operand in an AND operation. In that operand, the 0 is at the location where you want to produce a 0 in the result. All other digits are 1. Example:

NUMBER TO BE CHANGED	10010001	(DECIMAL 145)
OPERAND	AND 11101111	(DECIMAL 239)
RESULT	10000001	(DECIMAL 129)

Mentally AND those two numbers to verify the result.

Demonstration—Perform that AND operation using memory location 3333. Set a bit pattern and then zero one of the bits. Enter

```
POKE 3333,145
```

That puts binary 10010001 at location 3333. Enter

POKE 3333, PEEK(3333) AND 239.

That ANDs decimal 239 with the content of location 3333. Enter

PRINT PEEK(3333)

The decimal value is 129. Run 10000 and enter 129. Bit 4 has been set to 0 without changing any other bit at that location.

HOW TO SWITCH BITS FROM 1 TO 0 AND BACK AGAIN

In some programs, it is necessary to control individual bits at memory locations. The program must switch a bit back and forth between 0 and 1. You need a way to do that reliably, without knowing the value of the bit to be switched.

AND 0 will set any bit to 0 no matter what the original value of the bit.

OR 1 will set any bit to 1 no matter what the original value of the bit.

OR 0 will leave a bit set to zero if it is already zero.

Here is a way to switch a bit back and forth between 0 and 1, without worrying about its value before it is switched:

Begin by setting the bit to 0, using an AND operation, even if it is already 0. Then use an OR operation to set it to the desired value, 1 or 0. If a bit is first set to 0, OR 1 will change it to 1 because $0 \text{ OR } 1 = 1$. If a bit is first set to 0, OR 0 will leave it set to 0 because $0 \text{ OR } 0 = 0$.

When a bit has been set to 0 first by the AND operator, OR 0 and OR 1 *have the effect* of setting it to 0 or 1. I will refer to this as the AND/OR method of switching bits.

Demonstration—Set the value at location 3333 to binary 00000000 by poking decimal 0 into that location. Verify by peeking.

Let's switch bit 0 at that location back and forth between 0 and 1. Begin by deciding what the OR operand should be to switch the bit to a value of 1. It should be 00000001, which is decimal 1.

The AND operand to set the same bit to 0 is the same binary number with each bit reversed. Reverse means to change each 1 to a 0 and each 0 to a 1. The AND operand is binary 11111110, which is decimal 254. Enter

POKE 3333, (PEEK(3333) AND 254) OR 1

AND 254 sets bit 0 to 0. OR 1 sets it to 1. Enter

PRINT PEEK(3333)

Bit 0 has been set to 1. The OR operand to set it back to 0 is 00000000, which is decimal 0. Enter

POKE 3333, (PEEK(3333) AND 254) OR 0

AND 254 sets bit 0 to 0. OR 0 doesn't change it. In effect, OR 0 sets it to 0. Enter

PRINT PEEK(3333)

Now, switch bit 0 back to 1. Enter

POKE 3333, (PEEK(3333) AND 254) OR 1

PRINT PEEK(3333)

Switching Any Bit—Use the conversion routines at 8000 and 10000 as much as you wish while reading this discussion.

The preceding demonstration switched bit 0. Suppose you want to switch bit 3. The OR operand to set bit 3 to a value of 1 is 00001000, which is decimal 8. The AND operand to set that bit to 0 is 11110111, which is decimal 247. To set bit 3 to 1 at location 3333, enter

POKE 3333, (PEEK(3333) AND 247) OR 8

To set bit 3 back to 0, enter

POKE 3333, (PEEK(3333) AND 247) OR 0

Binary And Logical Operations

Switching More Than One Bit—This method works with a group of bits, regardless of their values before they are switched. Suppose you want to switch bits 3, 4 and 5. The OR operand to set bits 3, 4 and 5 to ones is 00111000, which is decimal 56. The AND operand to zero those bits is 11000111, which is decimal 199.

Begin by poking 0 into location 3333. Then enter

```
POKE 3333,(PEEK(3333) AND 199) OR 56
PRINT PEEK (3333)
```

Decimal 56 is stored at location 3333. Use routine 10000 to convert 56 into binary. Bits 3, 4 and 5 have been set to 1. To zero those bits, enter

```
POKE 3333,(PEEK(3333) AND 199) OR 0
```

When using this method, the bits don't have to be in sequence. You can switch any combination, such as bits 1, 4 and 7.

COMPLEMENTARY BINARY NUMBERS

When one binary number is obtained from another by reversing the values of all bits, the second number is the *complement* of the first and the two numbers are *complementary* to each other.

NUMBER	00111100	DECIMAL VALUE	60
COMPLEMENT	11000011	DECIMAL VALUE	195
SUM	11111111	SUM	255

The word *complement* is a form of the word *complete*. Notice that a binary number and its complement, when added together, form a "complete" number—meaning all ones. When the decimal equivalents are added together, the sum is 255. A binary number added to its complement is always equal to the maximum value that can be stored in that number of binary digits.

For 8-digit binary numbers written as decimals,

NUMBER + COMPLEMENT = 255

COMPLEMENT = 255 - NUMBER

This provides an easy way to calculate the AND operand when you are using the AND/OR method to switch bits. First, determine the OR operand needed to set the bits to 1. Suppose it is 00111100, which has a decimal value of 60.

The AND operand to set those bits to 0 is the complement of the OR operand. It is binary 11000011, which is decimal 195. When you know the decimal value of the OR operand, you can find the decimal value of the AND operand by calculation.

AND OPERAND = 255 - OR OPERAND
= 255 - 60
= 195

MASKING

Preceding demonstrations have obtained the number stored at a memory location by peeking. A PEEK statement does not change the data stored at that location. It just reads it. PEEK(3333) is a *variable name*. Its value is the content of location 3333.

Sometimes, you will want to "look at" a single bit at some location in memory to see if it is 1 or 0 and take some action depending on the value of that bit. You don't want to change the bit, you just want to know what it is.

After an entire byte is copied from memory by a PEEK statement, logical AND is used to select which bit or bits is to be examined. For example, ANDing a byte with 00001000 causes every byte to become 0 except bit 3, which is not changed. That does not affect the value stored in memory at the location peeked. You are operating on the value of PEEK(), not the actual content of the memory location peeked at.

Using logical AND to select bits to be examined is called *masking*. AND 00001000 masks all bits

except bit 3. AND 00000001 masks all bits except bit 0. Enter

```
POKE 3333, 255
```

That sets all bits to 1. Examine bit 3 by entering

```
PRINT PEEK(3333) AND 8
```

The result is decimal 8. The mask caused the binary number stored as PEEK(3333) to become 00001000. The decimal value of that number is 8. If you are looking for a *binary* 1 at position 3, peeking will find either decimal 8 or 0. Peeks and pokes use decimal numbers.

Here is a way to avoid the mental error of looking for 1 and finding 8. Enter

```
IF PEEK(3333) AND 2↑3=2↑3 THEN PRINT "Y"
```

The value 2↑3 is both the position that you want to examine in the binary number and the decimal value of a bit at that position.

To examine more than one position, do this:

```
IF PEEK(3333) AND 2↑1+2↑5+2↑7=2↑1+2↑5+2↑7  
THEN PRINT "Y"
```

That statement will execute if bits 1, 5 and 7 are all binary ones.

What has been demonstrated are methods of examining and controlling individual bits and groups of bits in memory. You will use these techniques in the next chapter.

LOGICAL NOT

The NOT operator works with only one operand. When used in a true/false determination, NOT reverses the true/false sense of the operand. Enter

```
PRINT NOT 0
```

```
PRINT NOT -1
```

If you experiment with NOT, you will see results that may be difficult to understand. That's because, mathematically, NOT performs a *twos-complement* operation—which is beyond the scope of this book.

SAVE THE CONVERSION ROUTINES

Save the conversion routines starting at lines 8000 and 10000. You can keep them in memory or save them to disk or on tape. The routines are used in the next chapter.

REVIEW

Learning to program a computer, in any language, isn't easy. It's not something you can pick up in half an hour by reading a magazine article or skimming through a book. It takes time and a lot of mental effort.

As with many activities, I think the reward is proportional to the amount of effort. When I was learning BASIC, I had difficulty with three things: arrays, DEF FN and everything in this chapter. After a long time and a lot of mental effort, I began to understand them.

The good news is that you can program very well without knowing everything about BASIC. You must understand arrays, but DEF FN is not essential for most programming. The information in this chapter is not essential unless you want to go beyond ordinary BASIC programming.

If you had difficulty with this chapter, or anything discussed earlier, don't be discouraged. Write programs with what you know and understand. Programming teaches programming. One day, you will find that you are using programming methods and ideas that you don't remember learning.

This chapter is a buffet. I suggest that you look through it again and take as much as your appetite suggests. Later, you can always come back for another serving.

BASIC words used in this chapter are AND, OR and NOT. Look them up in Appendix B.

19 Sprites And Bit Graphics

This chapter shows how to create screen displays by controlling individual bits at memory locations. By this method, you can draw shapes on the screen and create animated displays.

SPRITES

Sprites are small shapes on the screen created by a special, simplified technique. When you see the method, you may not think it is simplified, but it is.

The following demonstration is not the best way to program sprites, but I think it is the best way to understand how to program them. After the demonstration, I will discuss a better way to write programs using sprites.

A sprite is any shape or figure that you can make by controlling individual dots in a rectangle. The rectangle has 21 rows with 24 dots in each row. It can be placed anywhere on the screen and moved around under program control.

Making a sprite in the sprite rectangle is something like arranging checkers on a checkerboard to create a shape or draw a smiling face.

To specify the shape of a sprite, you must control individual bits in a *memory map* of the sprite. If a sprite bit in the memory map is set to 1, the corresponding dot on the screen is illuminated. A dot is the smallest area on the screen that can be individually controlled—such as turning it on or off. That tiny area is called a *picture element*, shortened to *pixel*. *Pixel* and *dot* mean the same thing.

If a bit in a sprite map is 0, that area of the sprite is transparent—you can see the screen background color through that part of the sprite. If something else is on the screen, such as alphanumeric characters, it can be seen through a hole in a sprite.

Bits that determine the shape of a sprite are stored in memory in groups of 8, as bytes. One memory location holds one byte.

The sprite rectangle has 24 dots on each row. Considering the top row as row 1, it is divided into three bytes. From left to right, they are bytes 1, 2 and 3. Row 2 of the sprite is specified as bytes 4, 5 and 6. The bytes are stored serially in memory, in that order.

Because a sprite has 21 rows, each requiring 3 bytes, the total number of bytes needed to specify a sprite is 21×3 , which is 63 bytes. To see how the shape of a sprite is specified using a memory map, enter the following demonstration program.

You should still have the binary-decimal-binary conversion routines in memory at lines 8000 and 10000 or have saved them to disk or tape. Keep them in memory while entering these lines at

lower line numbers—beginning at line 0. To enter line 11, type 24 ones between the quotation marks. Enter

```
0 PRINT CHR$(147):REM SPRITE DEMO
1 DIM S$(63):C=1:GOTO 11
10 REM LINES 11-31 DEFINE SPRITE 0
11 R$="111111111111111111111111":GOSUB4
```

Line 1 dimensions an array S\$() to hold 63 strings. It holds the 63 bytes necessary to specify the shape of a sprite. C is a counter used to fill the array. The name S\$ means *Sprite zero*. There can be as many as 8 sprites in a program, numbered 0 to 7.

This program assembles a map of a sprite on program lines in the program itself. Then it transfers that map into computer memory as a series of 63 bytes and displays the resulting sprite figure.

Line 11 represents the top row of the sprite rectangle. All of the 24 pixels in this row are turned on because each is set to 1.

Because there are 21 rows in a sprite rectangle, there must be 21 similar lines in the program. They will be lines 11-31. To relate line number to row numbers in the sprite rectangle, subtract 10 from the line number. Line 11 is sprite row 1.

Make lines 12-31 the easy way by overtyping line numbers. Except for line numbers, they will all be identical. When you have done that, the program will look like this:

```
0 PRINT CHR$(147):REM SPRITE DEMO
1 DIM S$(63):C=1:GOTO 11
10 REM LINES 11-31 DEFINE SPRITE 0
11 R$="111111111111111111111111":GOSUB4
12 R$="111111111111111111111111":GOSUB4
13 R$="111111111111111111111111":GOSUB4
14 R$="111111111111111111111111":GOSUB4
15 R$="111111111111111111111111":GOSUB4
16 R$="111111111111111111111111":GOSUB4
17 R$="111111111111111111111111":GOSUB4
18 R$="111111111111111111111111":GOSUB4
19 R$="111111111111111111111111":GOSUB4
20 R$="111111111111111111111111":GOSUB4
21 R$="111111111111111111111111":GOSUB4
22 R$="111111111111111111111111":GOSUB4
23 R$="111111111111111111111111":GOSUB4
24 R$="111111111111111111111111":GOSUB4
25 R$="111111111111111111111111":GOSUB4
26 R$="111111111111111111111111":GOSUB4
27 R$="111111111111111111111111":GOSUB4
28 R$="111111111111111111111111":GOSUB4
29 R$="111111111111111111111111":GOSUB4
30 R$="111111111111111111111111":GOSUB4
31 R$="111111111111111111111111":GOSUB4
```

All of the bits in this sprite map are set to 1, so it will be a rectangle on the screen. Later, you can change some of them to make a shape.

Each of these lines creates a string variable named R\$, which means *row*. Each has 24 characters, all ones. Characters will be poked into memory, one byte at a time.

The next step is to divide each group of 24 characters into three bytes, so each byte can be poked into memory.

Program lines 11-31 each jump to a subroutine at line 4, which divides R\$ into three smaller strings with 8 bits in each string. When the subroutine has finished operating on the R\$ provided by line 11, it will return to line 12. Line 12 provides another R\$ and jumps to the subroutine at line 4 where that R\$ is subdivided into three strings. It doesn't matter that each R\$ is the same as the others. Later, they won't be the same.

When all 21 rows on the sprite map are processed by the subroutine, 63 bytes have been derived from the sprite map and placed in the array S0\$().

The reason the subroutine is at line 4 is so lines 11-31 will each occupy only one row on the screen. The lines say GOSUB4, which fits on one row. That makes a bit map that will fit on the screen when you display lines 11-31 of the program. GOSUB 50 or any number with more than one digit would cause each of lines 11-31 to occupy two screen rows.

Now you can enter the subroutine. List lines 0-11. Notice that S0\$ uses the number zero, not letter O. Enter

```
2 REM -----
3 REM SUBR MAKES R$ INTO 3 BYTES. PUTS
  EACH BYTE INTO S0$( ) TO BUILD SPRITE 0
4 FOR I=1 TO 17 STEP 8
5 S0$(C)=MID$(R$,I,8)
6 C=C+1
7 NEXT I
8 RETURN
9 REM -----
```

List lines 0-15. Be sure you entered S0\$, not SO\$.

Line 4 sets up a loop to take three bytes out of each R\$. To see what it does, use the immediate mode to enter

```
FOR I=1 TO 17 STEP 8:PRINT I:NEXT
```

I has three values: 1, 9 and 17. Line 5 uses those values in a MID\$() statement to divide R\$ into three pieces, each with 8 characters. The surgery is done by MID\$(R\$,I,8).

When I is 1, the first 8 characters are placed in S0\$(C). Counter C was initialized at line 1. It begins counting at 1. The first byte in the first row of the sprite rectangle goes into S0\$(1). The next byte goes into S0\$(2) and so forth, until 63 bytes have been formed and placed into the array.

Line 7 is the bottom of the I loop. When three bytes have been formed from R\$, line 8 executes. The subroutine returns to the next line in the main program. It gets the next R\$, converts it into three bytes, and plugs them into the array S0\$().

To test this program segment, enter a STOP at line 1000. That prevents it from running down to line 8000 and asking you to make a binary-decimal conversion whether you want to or not.

Run the program. When it stops at line 1000, enter

```
PRINT S0$(1)
```

You should see 11111111. If not, check your typing. Then enter

```
FOR I=1 TO 63:PRINT I, S0$(I):NEXT
```

You should see 63 identical strings. Each of these must be poked into specified locations in memory to create the bit map that the computer will actually use to display the sprite.

Even though the end result is binary numbers in memory, POKE statements use decimal numbers. The next step is to convert each of these 63 strings into its decimal equivalent, so it can be poked.

That's what the routine at line 8000 does. List 8000-9000 and review it. It accepts an 8-bit binary string and converts it to a decimal number, D, by adding up the position values for each bit that is 1.

The routine at line 8000 could be used as a subroutine in the program you are entering. I decided not to do that because you may want to use the routine at 8000 to do other things while you are entering this program. The lines in that routine are so simple that it is no problem to write them into the sprite program. List -1000. Enter

```

90 REM -----
100 REM S0$( ) HOLDS BINARY. CONV TO DEC
110 FOR I=1 TO 63: B$=S0$(I):
    REM CONVERT 63 BYTES
120 D=0:X=7:REM D GETS DECIMAL VAL
    X IS POSITION IN BYTE
130 FOR CV=1 TO 8:REM CHECK 8 BITS
140 IF MID$(B$,CV,1)="1" THEN D=D+2^X:
    REM IF BIT=1 ADD POS VAL TO D
150 X=X-1:NEXT CV
160 S0$(I)=STR$(D):REM PUT IN ARRAY
165 PRINT I, B$, S0$(I):REM TO TEST
170 NEXT I
175 PRINT CHR$(147):REM TEST
190 REM -----

```

Line 110 sets up a loop to run 63 times. Each time, it sets B\$ equal to the content of S0\$(I). By giving each of these 8-bit strings the name B\$ as it is processed, instead of calling it S0\$(I), this routine is simpler to write. B\$ means *binary string*.

This routine takes each value of S0\$(I) and names it B\$. Then it converts it to decimal as D and converts D to a string. Next it puts the result back into S0\$(I), replacing the binary value that was there originally. When it has finished, the array S0\$() holds the same *information* as before but each element has been changed from a binary number to its decimal equivalent.

Line 165 is a temporary line used to test this routine. It displays the byte number, from 1 to 63, the original binary string, and the converted decimal string at each location of the array S0\$. Line 170 is the bottom of the loop. Line 175 erases the display produced by line 165.

Run it. Each 11111111 should be converted to 255. Doing it this way is slow. It is not the way sprite programs should be written. But it shows how to make a sprite. The next step is to poke the 63 decimal numbers into memory.

Two things have happened so far. A uniform bit pattern was placed on program lines by typing it into the program. Each of the bit patterns was divided into three bytes and converted into a decimal number. As you saw in the preceding chapter, poking that decimal number into a memory location will put the *same* bit pattern into memory that was originally typed on the program line.

SELECTING A MEMORY LOCATION FOR A SPRITE MAP

Sprite maps must be accessible to the program. They can be placed in the same section of RAM used by the program. That segment of memory begins at location 2048. There is a small amount of RAM below 2048 that can be used to store a sprite map, but it isn't large enough for several sprites.

This program will put the sprite map at location 12288, which is in the section of memory used by the program itself. If a program is long enough to extend from 2048 past 12288, poking the sprite map into memory at 12288 would destroy part of the program, and it wouldn't run correctly.

You must choose the memory location for sprite maps to avoid that conflict. One way is by experiment. If it works, the sprite map is high enough in memory so there is no conflict with the program.

Another way is to save this program on disk. Then list the disk directory and notice the number of blocks used by this program. Multiply blocks by 256 to get the length of the program in bytes. The program begins at location 2048 in memory. Add the length of the program, in bytes, to 2048.

Sprites And Bit Graphics

The result is the highest memory location used by the program. Put the sprite memory map higher than that.

The upper limit of the memory space used for the program and sprite maps is location 40959. It takes a very large program and a lot of data to fill that space.

Sprite Pointers—Eight memory locations are used to select starting memory locations for eight sprite maps. These locations are called *sprite pointers*. When the program displays sprite number 0, it looks at the pointer for sprite 0. The pointer tells it where to find the beginning of that sprite map in memory.

Sprite-pointer memory locations are just above the screen memory map where characters are stored for display. As you know, the screen memory map uses 1000 locations from 1024-2023.

Sprite pointers use locations 2040-2047. The first line of a BASIC program begins at 2048. Location 2040 is the pointer for sprite 0, location 2041 is for sprite 1 and so forth. The first memory location for the sprite map is placed in the pointer location by poking a *pointer value* from 0 to 255 into the pointer location—such as location 2040 for sprite 0. Then the computer knows where to find that sprite map in memory.

The number poked into the sprite pointer location is not the actual location of the sprite map in memory. Let's call the pointer value *P*. The memory segment designated by *P* begins at $P \times 64$. That formula is valid for the power-up condition of the computer—as it is normally used.

Each value of *P* selects a 64-byte segment of memory. The sprite map uses the first 63 of those bytes. Byte 64 is not used. Sprite-map locations always begin in memory at an even multiple of 64.

One value of *P* to avoid is 32 because $32 \times 64 = 2048$. That would overlay the sprite map at the very beginning of the program using it—obliterating that part of the program.

There is no special reason to use location 12288 in this program as the beginning of the map for sprite 0 except that it is high enough. To find the value of *P* that represents location 12288, divide 12288 by 64. $12288 / 64 = 192$.

To select memory location 12288 as the beginning of the memory map for sprite 0, poke the number 192 into location 2040. Location 2040 is the pointer for sprite 0.

If there is more than one sprite, you would normally use the number 193 as the pointer for sprite 1, and so forth. That map location for sprite 1 is specified by poking 193 into location 2041, which is the pointer for sprite 1, and so forth.

To designate a location for the sprite 0 map in this program, list -1000 and enter

```
200 REM SET SPRITE 0 MAP LOCATION
210 POKE 2040, 192: LOC=12288
220 REM -----
```

Line 210 puts the map address into the pointer at 2040. It also sets a variable, LOC, to that starting address in memory. LOC will be used to poke the sprite map into memory.

Putting the memory location into the sprite pointer tells the computer where to look for the sprite map. That doesn't actually put the sprite map into memory at the specified location. The map must be poked into memory. Enter

```
230 REM POKE SPRITE 0 DATA INTO MEM
240 FOR I=0 TO 62
250 POKE LOC+I,VAL(S0$(I+1))
260 NEXT I
270 REM -----
```

Line 250 converts the strings held in the array S0\$() back into numeric values using VAL(). Now they are decimal numbers again. Each decimal value is poked into memory, beginning at location LOC, which is 12288. Each of those values is the decimal equivalent of one 8-bit binary segment of the sprite map. Each is in the range of 0-255.

Now, the sprite map is in memory at a location higher than the computer will use to store this program.

LOCATING SPRITES ON THE SCREEN

The program must tell the computer where to display each sprite. This is done using X and Y numbers, called *coordinates*. The X number is the number of pixels over from the left. The Y number is the number of pixels down from the top.

The screen location specified by the coordinates X and Y is the location of the *top-left corner* of the sprite rectangle.

When measured in pixels, the screen is wider than it is tall. The range of X values is 0-511. The range of Y values is 0-255. Because X may be larger than 255, it is handled as a two-byte number.

The part of the computer that displays sprites is the video chip. I have been calling it the *display controller*. Sprite instructions are delivered to the video chip by poking numbers into memory locations that are used by the video chip. The first memory location that controls sprite position on the screen is 53248.

Assign the number 53248 to a variable named V, for *video* chip. It is easier to type V than 53248. For the rest of this chapter, V means memory location 53248.

In the video chip section of memory, there are 17 locations for X and Y coordinates. V+0 holds the *low* byte of X for sprite 0. V+1 holds the Y location for sprite 0, which is a single byte.

V+2 and V+3 hold *low* byte of X and the Y byte for sprite 1, and so forth. Eight pairs of locations are used to hold low byte of X and the Y byte for eight sprites. This takes locations V+0 to V+15.

Location V+16 is used to store the *high* byte of X for all eight sprites. Only one bit is stored for the high byte of each sprite. It is the rightmost bit of the high byte for that sprite. It is 1 or 0. All other bits in the high byte are zero, so they are not stored.

Bit 0 at location V=16 stores the high byte of X for sprite 0. Bit 1 stores the high byte of X for sprite 1, and so forth. This arrangement is shown in the accompanying table.

X-Y COORDINATE LOCATIONS FOR TOP-LEFT CORNERS OF SPRITES 0-7 (V=53248)			
SPRITE	LOW BYTE OF X	Y-BYTE	HIGH BYTE OF X
0	V+0	V+1	V+16, bit 0
1	V+2	V+3	V+16, bit 1
2	V+4	V+5	V+16, bit 2
3	V+6	V+7	V+16, bit 3
4	V+8	V+9	V+16, bit 4
5	V+10	V+11	V+16, bit 5
6	V+12	V+13	V+16, bit 6
7	V+14	V+15	V+16, bit 7

Only a single bit is needed for the high byte of X because there are only 511 pixels in the X dimension of the screen. With 8 bits stored as the low byte and one more bit stored at the high byte, 9 bits are used to store the value of X. With 9 bits, you can count to 511.

Considering the value of X as a nine-bit number, the ninth bit is the leftmost bit. It is therefore the *most-significant bit* (MSB) of that nine-bit number because it contributes the greatest position

value to the number. The bit stored as the high byte of X is the MSB of X. To locate sprite 0 on the screen, enter

```
280 REM SET X,Y FOR SPRITE 0
300 V=53248:REM STARTING VIDEO LOCATION
310 POKE V,100:REM X=100
320 POKE V+16,0:REM MSBX=0
330 POKE V+1,100:REM Y=100
340 REM -----
```

In line 310, V is the same as V+0. It stores the low byte of X. Line 320 sets V+16 to 0. That sets all bits to 0. If there were more than one sprite, you should set each bit for each sprite. In this program, there is only sprite 0, so the setting of the other bits doesn't matter.

The low byte of X is 100 and the high byte is 0. The decimal value of a two-byte number is

Value = Low byte + (256xHigh byte)

High byte is first evaluated as if it were a low byte and then multiplied by 256. The value of X is 100 because high byte is 0.

Line 330 puts the Y coordinate into location V+1. The top left corner of the sprite 0 rectangle will be placed 100 pixels down from the top and 100 pixels over from the left.

SELECTING SPRITE COLOR

A sprite may be any of the 16 colors you have been using. Color codes are the same: 0 to 15. There are 8 memory locations to specify sprite colors, one for each sprite. They are locations 53287 to 53294. Location 53287 sets color for sprite 0, and so forth. Summary tables later in this chapter show all of the memory locations used to control sprites.

Color for sprite 0 is poked into location 53287. It must be a number in the range of 0-15. Enter

```
400 REM SET SPRITE 0 COLOR
410 POKE 53287,0:REM SPRITE 0 IS BLACK
420 REM -----
```

If there is more than one sprite, the color for each is set in the same way, using the appropriate memory location.

TURNING ON SPRITES

Sprites are turned on and off by poking into *one* memory location. It is called the *sprite-enable* location. Each of the eight bits at that location controls one sprite. Sprite 0 is controlled by bit 0. Sprite 7 is controlled by bit 7. To turn a sprite on, set the corresponding bit to 1. To turn it off, set that bit to 0. The sprite-enable location is 53269. Enter

```
500 REM ENABLE SPRITE 0
510 POKE 53269,PEEK(53269) OR 1
520 REM -----
```

Line 510 sets bit 0 to 1. That turns on sprite 1. If you have more than one sprite, you will probably want to control individual bits at the sprite-enable location to switch sprites on and off individually or in groups. The AND/OR method discussed in Chapter 18 will do that.

At last, you are ready to display a sprite. Run the program. You can see the sprite map on program lines 11-31 being converted into binary bytes and decimal equivalents. Then the sprite is displayed at X=100, Y=100.

The sprite should be a black rectangle about two inches down from the top and two inches over from the left. Fix typos if necessary.

MOVING SPRITES

To move a sprite, change the X,Y coordinates. That can be done by loops in the program, joysticks or from the keyboard. These lines will change sprite position using the function keys and

also display the X,Y coordinates as the sprite moves. Enter

```

600 REM MOVE SPRITE 0
610 X=100:Y=100
620 GET A$:IF A$="" GOTO 620
630 IF A$=CHR$(133) THEN Y=Y-1:REM F1 UP
640 IF A$=CHR$(134) THEN Y=Y+1:REM F3 DN
650 IF A$=CHR$(135) THEN X=X-1:REM F5 L
660 IF A$=CHR$(136) THEN X=X+1:REM F7 R
670 HX=INT(X/256):REM HI BYTE FOR X
680 LX=X-(256*HX):REM LO BYTE
690 POKE V,LX
700 POKE V+16,(PEEK(V+16) AND 254) OR HX
710 POKE V+1,Y
714 PRINT CHR$(147)
715 PRINT TAB(10)"X ="X" Y ="Y
716 PRINT TAB(10)"LO BYTE ="PEEK(V)
    " HI BYTE ="PEEK(V+16)
720 GOTO 620

```

Line 610 sets X and Y both equal to 100, so this routine will start moving the sprite from its existing position on the screen. It's OK to locate or move sprites from more than one place in a program.

The GET routine at line 620 detects which function key was pressed. Lines 630-660 change the value of X or Y accordingly.

Line 670 takes the high byte of X and names it HX. Line 680 takes the low byte of X and names it LX.

Line 690 pokes the low byte into V. Line 700 pokes the high byte into bit 0 at location V+16. Because the high byte may be 0 or 1, depending on where the sprite is on the screen, this line uses the AND/OR method to switch that bit.

Line 710 pokes the Y coordinate into V+1. Poking changed values for X and Y into those memory locations, causes the sprite to move.

Lines 714-716 are to test the program. They display the values of X and Y and the low and high bytes of X. Line 720 is the bottom of an endless loop. It jumps back to 620 to get another keystroke and move the sprite again.

Run the program. Use the function keys to move the sprite. They move it only one pixel at a time, in each direction. Don't move it all the way off the screen. If the program doesn't move the sprite, check for typos.

MOVING SPRITES ON AND OFF THE SCREEN

Using the function keys, move the sprite so it fits exactly in the top-left corner of the screen. Notice the X,Y coordinates. Those X,Y coordinates are the smallest values that will display the entire sprite when it is near the top or left border of the screen.

I have been saying that X is the number of pixels over from the left and Y is the number of pixels down from the top. X and Y are measured from reference points that are off the screen. This allows you to move a sprite off the screen.

Move the sprite off the screen through all four borders. Notice the X,Y coordinates when the sprite just touches a border and when it moves just beyond the edge of the screen so you can no longer see it. Make notes of those X,Y values.

Don't allow X or Y to become negative. Don't allow X to exceed 511 or Y to exceed 255. If you do, an error occurs and the program stops. No harm is done. Run it again. A working program would have statements to prevent those errors, such as: IF X<0 THEN X=0 and IF Y>255 THEN Y=255

You may notice that the sprite is turned off and reformed when X changes from 255 to 256 or 256 to 255. At those transitions, the MSB is changed in V+16. The computer turns the sprite on and off at that location, not the program.

MOVING MORE THAN ONE SPRITE

If a program has more than one sprite, the general method of moving it is the same. The low byte for X and the value of Y are poked into the correct memory locations for that sprite number. For example, sprite 1 uses V+2 and V+3 for low X and the value of Y.

The high byte of X for each sprite is one of the MSB bits at V+16. The MSB for sprite 0 is bit 0 at location V+16. The MSB for sprite 1 is bit 1 at V+16, and so forth. No matter which sprite you are controlling, its MSB will be either 1 or 0. What you must do is poke 1 or 0 into the correct bit of V+16, as discussed in Chapter 18.

EXPANDING SPRITES

You can make a sprite become twice as wide and twice as tall instantly. Run the program and move the sprite to the top-left corner of the screen. Keep all of the sprite in view. Then break out of the program.

The 8 bits at location V+29 control the width of each sprite. Setting a bit to 1 causes the sprite to double in width. Setting it to 0 causes normal width. The byte at location V+23 controls how tall each sprite is, in a similar way. In the immediate mode, enter

```
POKE V+29,1
POKE V+23,1
CONT
```

Notice that the sprite rectangle grew downward and to the right. The top-left corner did not move. The X,Y coordinates locate the top-left corner of a sprite, whether expanded or not.

Try hiding an expanded sprite off the left edge of the screen. There isn't enough room. Try hiding it off the top and bottom. You can do that. Move the sprite off the right side of the screen. Keep pressing f 7 until it reappears at the left border of the screen and moves fully into view. Notice the value of the high byte of X. It is 2. In binary, decimal 2 is 00000011. Moving sprite 0 off the right side of the screen and back around onto the left side set the MSB bit for sprite 1.

The reason to limit the X value to 511 for sprite 0 is to keep from using the MSB for sprite 1. As the X coordinate becomes larger, the video chip "rotates" it off the right side of the screen and back onto the left side.

If you don't have a sprite 1, you can rotate sprite 0 by that method. If you don't have any other sprites, you can use all eight bits at V+16 as the high byte for sprite 0.

A better way is to move a sprite off the screen and then change the X,Y coordinates to relocate it anywhere you wish, including off the screen at another location.

CHANGING SPRITE COLOR

Poking a color code in the range of 0-15 into location V+39 controls the color of sprite 0. Enter

```
POKE V+39,1
```

Try other colors, if you wish.

CHANGING SPRITE SHAPE

Clear the screen by pressing RUN STOP-RESTORE. List 11-31. Move the cursor into the bit map on those lines. Remove the top-left corner of the sprite by pressing the space bar to replace ones with spaces. Make a hole in the middle of the sprite by replacing ones with spaces. Do something artistic, but leave a few ones in the top row and at the left to mark those boundaries of the sprite rectangle.

You will probably forget to press RETURN before moving the cursor out of each line that is changed. If so, the line is changed on the screen, but not in memory. You may as well make all changes without pressing RETURN. Then place the cursor in line 11 and press RETURN repeatedly until all lines are entered again.

Run the program. You can see the bit patterns being converted into bytes and decimal equivalents. The blanks in the binary numbers are treated as zeros by the decimal conversion routine because it doesn't do anything unless a bit is 1.

When the sprite appears, it will have the same appearance as the changed bit map in program lines 11-31. Move it into the top-left corner again. The X,Y coordinates locate the top-left corner of the sprite, whether anything is there or not.

Move the sprite so the hole in the middle passes over the X and Y coordinates displayed on the screen. Notice that you can see whatever is on the screen, through the hole.

SPRITE OVERLAY PRIORITIES

The sprite seems to pass on top of characters put on the screen by PRINT statements or poking into the screen memory map. If there is more than one sprite, one with a lower sprite number will appear to pass on top of those with a higher sprite number, when their paths intersect.

DETECTING SPRITE COLLISIONS

A sprite may collide with a border of the screen, background characters placed on the screen by PRINT statements or poking into the screen memory map, or with another sprite.

Collisions with a Border—To detect collisions with a border, monitor the X,Y coordinates, remembering that the coordinates locate the top left corner of the sprite rectangle.

Collisions with Another Sprite—Memory location 53278 records collisions between sprites. The 8 bits at that location represent the eight sprites. When sprite 0 is involved in a collision with another sprite, bit 0 is set to 1 by the computer. The bit for the other sprite is also set to 1.

A collision is recorded when any *visible* part of a sprite overlays any visible part of another display on the screen.

Normally, peeking at a memory location does not change the content of that location. That is not true for collision detection. Reading location 53278 by a PEEK statement sets all bits to 0. If you are counting collisions, such as in a game, accumulate them in a variable for each sprite.

Collisions with Screen Background—In a similar way, location 53279 records collisions between a sprite and the screen background—alphanumeric or graphics characters.

Demonstration—Put a background collision detector in the program. List -1000 and enter

```
605 POKE 53279,0:REM CLEAR COLLISIONS
717 IF PEEK(53279)AND 2↑0=2↑0THEN PRINT:
PRINT"Bang!":REM COLLISION WITH BG
```

Line 605 clears the collision register to be sure no bits are set. Line 717 examines bit 0 in the register using AND 2↑0 as a mask. Masking was discussed in Chapter 18.

Run the program. Move the sprite up until it collides with the LO BYTE display. Move the sprite down. Notice that two collisions are recorded: one when the sprite is moved into the other display, another when the sprite is moved back.

ANNOUNCING COLLISIONS

There are a lot of things you can do when a collision occurs, depending on the program you are writing. You can produce sound effects by methods described in the next chapter.

You can create another sprite that looks like an explosion, position it at the collision, and blink it on and off. You can blink the entire screen on and off a few times.

Screen Blinking—Setting bit 4 to 0 at location 53265 changes the entire screen to its border color. The displays are still there, but you can't see them. Setting bit 4 back to 1 turns the screen back on again.

MULTICOLOR SPRITE MODE

The sprite you have been displaying is all one color. Sprites are sometimes more effective if more than one color is used to make them. In the multicolor mode, four colors are available.

The multicolor mode is selected by setting bit 4 at location 53270 to 1. It is turned off by setting that bit to 0.

DISPLAYING SPRITES (V=53248, S=sprite number)		
To Set	POKE Location	What To POKE
Sprite pointers	2040-2047	Encoded starting location, P, of sprite map in memory. Location = Px64.
Sprite color	V+39 to V+46	Color code 0-15
Horizontal expand	V+29	Set bit 2 S
Vertical expand	V+23	Set bit 2 S
Sprite X coord.	V+0 to V+14	X low byte at V+(2 x S)
MSB for X coord.	V+16	Set bit 2 S
Sprite Y coord.	V+1 to V+15	Y value at V+1+(2 x S)
Enable sprite	V+21	Set bit 2 S
Collide with sprite	V+30	Read bit 2 S
Collide with bg	V+31	Read bit 2 S
Background color	V+33	Color code 0-15

When multicolor has been selected, the four available colors are screen color (background), sprite color, color #1 and color #2. These colors are set by poking the same 16 color codes that you have been using.

Screen color is set in the usual way by poking a color code into location 53281. Sprite color is set by a poke into a separate memory location for each sprite, V+39 to V+46.

Sprite color isn't really the sprite color any more. It is just one of the four colors that can be used. It is called that because it is poked into the same memory location that selects sprite color when only one color is used. Colors #1 and #2 are additional colors that can be used.

The table on the next page summarizes poke locations for sprites in multicolor.

If you intend to display a sprite in multicolor, you should make the bit map in a different way than you would to display it in a single color.

When the multicolor mode is selected, the pixels in each row of a sprite on the screen are combined into pairs. The two pixels in a pair are the same color, so they are effectively one large dot.

Instead of 24 small dots in a row in a sprite rectangle, there are 12 larger dots. In the sprite map, each multicolor dot is specified by two adjacent bits.

When colors have been selected and a sprite is set to the multicolor mode, the bit map for that sprite is interpreted in a different way. The first pair of bits in each byte specifies the first large dot, which occupies the space formerly occupied by two pixels. The next pair specifies the next large dot, and so forth.

A pair of bits is called a *bit pair*. Each bit pair specifies one of the four available colors by these rules:

BIT PAIR	DOT COLOR (Two-pixel dots)
00	Screen color (transparent)
01	Multicolor #1
10	Sprite color
11	Multicolor #2

Make the bit map for a multicolor sprite by specifying bit pairs instead of individual bits. If you already have a bit map for a sprite in a single color, selecting the multicolor mode for that sprite will produce unpredictable results.

Collisions in Multicolor— With a multicolor sprite, collisions are recorded when any pixel with color 10 or 11 overlaps a visible part of the background or colors 10 or 11 in another sprite. Sprite multicolors 00 and 01 do not cause collisions to be recorded.

Demonstration—Begin by listing -500. The program segment starting at line 400 now sets the sprite to a single color. Enter these lines to set four colors and set the sprite to the multicolor mode:

```
405 POKE 53281,15:REM SCREEN COLOR GRAY 3
406 POKE 53285,2:REM COLOR #1 RED
407 POKE 53286,5:REM COLOR #2 GRN
410 POKE 53287,7:REM SPRITE YELO
420 POKE V+28,PEEK(V+28) OR 1:REM SET
    SPRITE 0 TO MULTICOLOR
```

Check your typing. Then list 11-31. Change all of the bits in this map to 1. Don't forget to press RETURN at each line. That sets every bit pair to 11. The sprite will be a rectangle in color #2, which is green. The screen will be gray 3, as selected by line 405. Run the program.

Break out of the program by pressing RUN STOP-RESTORE. List 11-31 again. Change the first three rows to this series of binary bit pairs: 1010101010 and so forth to the end of the row. That will make the first three rows of the sprite appear in color 10, which is sprite color yellow.

Change the next three rows to a series of binary 01 bit pairs, which is color #1, red. The rest of the map is a series of binary 11 bit pairs, which is color #2, green.

Move the cursor down into the middle of the area filled with binary 11 bit pairs and make a large hole. To do that, replace bit pairs 11 with bit pairs 00.

Move the cursor to program line 11 and press RETURN repeatedly to enter all of the changes. With the cursor at the bottom of the screen, enter RUN. The result is three yellow rows, three red rows, and a large green area with a hole in the middle.

DISPLAYING SPRITES IN MULTICOLOR (V=53248, S=sprite number)		
TO SET	POKE LOCATION	WHAT TO POKE
Background color	V+33	Color code 0-15
Color #1	V+37	Color code 0-15
Color #2	V+38	Color code 0-15
Sprite color	V+39 to V+46	Color code 0-15
Any sprite to multicolor mode	V+28	Set bit 2 (S to 1)
Any sprite to normal one-color	V+28	Set bit 2 (S to 0)

A BETTER WAY TO PROGRAM SPRITES

One way to develop the 63 bytes that specify the shape of a sprite is to do it with pencil and paper. Make a grid with 24 columns and 21 rows. Fill the squares on the grid to draw the sprite shape. Then convert that shape into binary numbers by taking 8 bits at a time in each row, just as this program does.

Convert the binary numbers to decimal and make a data table showing the decimal values for bytes 1 to 63 of the sprite. You will probably agree that it is time-consuming and tedious to do it that way.

The routine at the beginning of this demonstration program is a convenient and quick way to draw a sprite shape on a map and convert the map into decimal numbers. That job is done when you have the array `S0$()` in memory, filled with 63 decimal numbers. To make a list of those numbers on paper, display the array on the screen and copy the numbers. Or else, write a routine to print the array.

By one method or the other, you end up with a list of 63 decimal numbers on a piece of paper.

A program that uses sprites will get them on the screen much faster if it doesn't have to do the sprite-map conversion used in this demonstration program. Instead, develop the 63-byte specification however you wish.

Then write a routine in your sprite program that puts the 63 decimal numbers into `DATA` statements. Then, use a loop that counts to 63 to read the data and poke it into the area of memory that you are using to store the sprite bit map. When that routine runs, it will put sprites on the screen quickly.

With the methods that have been demonstrated, you can make a display full of moving objects and as dazzling as you want it to be.

If you want to save the program in memory to disk or tape, do it now.

OTHER ADVANCED DISPLAY TECHNIQUES

There is also a multicolor mode for characters, using bit pairs in the same way that bit pairs are used to define colors in sprites.

If you can make a sprite, you can make it look like a large letter A by setting the correct bits in the sprite. By a similar technique, you can design characters that fit into the 8-dot by 8-dot space that is normally used on the screen to display characters from character sets 1 or 2.

You can create your own character sets, which can be used instead of the standard character sets. That requires advanced programming methods to reassign memory locations and direct the screen controller to look for the character sets at different locations in memory than the normal locations.

That technique is not discussed in this book. It is discussed in the *Commodore Programmer's Reference Guide*, available at your Commodore dealer and at bookstores.

The following section will give you a general idea of how it is done—and the amount of labor required to do it.

PRINTING DOT GRAPHICS ON PAPER

By controlling individual dots, you can print graphics bit-by-bit on the Commodore Graphic Printer. This section assumes that you have read Chapter 17.

The print head in the printer has 7 pins arranged in a vertical column. When printing, each of the 7 pins is controlled individually. It strikes the paper, through the ribbon, or it does not.

Standard characters from either of the character sets are formed in a grid, which is called a *dot matrix*. This is done by printing one vertical column of dots, moving the print head to the right, and printing the next vertical column of dots.

When printing standard characters, each character uses 7 dots vertically and 6 dots horizontally. The characters are formed using 7 dots vertically and 5 dots horizontally. The rightmost column of dots is left blank to provide the space between characters. With a magnifying glass, you can see the dots made by the printer.

When printing standard characters, there are 6 lines of type per inch on the paper. That line spacing provides vertical separation between lines of type.

In the dot-graphics mode, the printer makes 9 lines per inch. That spacing allows printing continuous graphics patterns, from one line to the next, without vertical separation caused by advancing the paper. You can provide vertical separation by not printing some of the dots, if you wish.

PRINTER CONTROL CODES

These control codes were shown in Chapter 17, but not all were demonstrated. They are used in the dot-graphics mode.

CODE	EFFECT
CHR\$(8)	Select graphic mode (Remains set until CHR\$(13) executed.)
CHR\$(15)	Select standard characters
CHR\$(16)	Tab print head
CHR\$(26)	Repeat graphics pattern
CHR\$(27)	Specify dot address

SELECTING GRAPHIC MODE

To set the printer for dot graphics, use CHR\$(16) in a statement of this form:

PRINT # *file n*, CHR\$(8)

A similar statement, using CHR\$(15) returns to the standard character mode.

SPECIFYING DOT ADDRESS

In the standard character mode, the printer can print 80 characters on a line. Each character uses 6 dots, so there can be as many as 480 dots in a line or 480 vertical columns of dots on a page.

In the dot-graphics mode, each column of dots is individually addressable. That means, you can position the print head to print at any column of dots on the page. The dot columns are numbered from 0 to 479, left to right, in each line on the page.

To specify the print-head position in the dot-graphics mode requires a two-byte number. It is preceded by a special control code, CHR\$(27) and CHR\$(16). CHR\$(27) tells the printer that the address following CHR\$(16) is a two-byte dot-column location on the paper. The format for a dot address is

PRINT # *file n*, CHR\$(27)CHR\$(16) CHR\$(hi byte) CHR\$(lo byte)

CONTROLLING INDIVIDUAL PINS IN THE PRINT HEAD

One byte controls one vertical row of seven dots on the paper. Bit 0 controls the top pin in the print head. Bit 6 controls the bottom pin. Setting a bit to 1 causes that pin to strike the paper. Setting it to 0 causes the pin not to strike the paper. Bit 7 is always set to 1 even though it doesn't print anything.

DESIGNING A CHARACTER

If you use the standard 6x7 matrix to form a character, 6 bytes are used to print the 6 vertical columns that form one character. Each has 8 bits, numbered 0 to 7. Bit 7 is always 1 but it doesn't print.

One way to design a character is to draw the matrix on paper and fill it in with ones and zeros to draw whatever you want. Here is a matrix full of ones:

BIT NUMBER	BIT VALUE	COLUMNS IN MATRIX					
		1	2	3	4	5	6
0	1	1	1	1	1	1	1
1	2	1	1	1	1	1	1
2	4	1	1	1	1	1	1
3	8	1	1	1	1	1	1
4	16	1	1	1	1	1	1
5	32	1	1	1	1	1	1
6	64	1	1	1	1	1	1
7	128	1	1	1	1	1	1
COLUMN TOTALS		255	255	255	255	255	255

The decimal value of the bit pattern in each column is the sum of the position values for each bit set to 1. In this example, the column totals are all 255. This six-column character will be a solid rectangle of dots.

To change it to another character, replace any of the ones except bit 7 with zeros and make new column totals.

PRINTING THE CHARACTER

In this demonstration, the 6 bytes will be put in a DATA statement and read by a READ statement used in a loop. Enter

```
NEW
10 REM DEMO DOT GRAPHICS
20 PRINT CHR$(147)
30 REM ----- SPECIFY CHAR
40 C$=" ":REM CHAR
50 DATA 255,255,255,255,255,255
60 FOR I=1 to 6
70 READ COL
80 C$=C$+CHR$(COL)
90 NEXT I
95 REM ----- PRINT GRAPHIC
100 OPEN 4,4,0
110 PRINT # 4, CHR$(8);:REM GRAPHICS
120 PRINT # 4, CHR$(27) CHR$(16)
    CHR$(0) CHR$(50);:REM DOT ADDRESS
130 PRINT # 4,C$
140 CLOSE 4
```

Lines 30-90 read the six bit patterns into memory as a single string variable, C\$. Line 100 opens the printer. Opening it with the statement OPEN 4,4,7 will have the same result as far as graphics are concerned. The way it is opened will affect printing standard characters.

Line 110 selects the graphics mode. Line 120 positions the print head at dot column 50. Line 130 prints the character.

Turn on the printer if it is not on. Run the program. The result should be a solid rectangle of dots about one inch from the left margin.

DESIGNING ANOTHER CHARACTER

Let's change the bit pattern to make a letter Z.

BIT NUMBER	BIT VALUE	COLUMNS IN MATRIX					
		1	2	3	4	5	6
0	1	1	1	1	1	1	0
1	2	0	0	0	0	1	0
2	4	0	0	0	1	0	0
3	8	0	0	1	0	0	0
4	16	0	1	0	0	0	0
5	32	1	0	0	0	0	0
6	64	1	1	1	1	1	1
7	128	1	1	1	1	1	1
COLUMN TOTALS		225	209	201	197	195	128

If the row of ones at the bottom makes it difficult to visualize the pattern, you can leave them out of the grid. They are always ones. If you do that, total the columns and then add 128 to account for bit 7. Change line 50 like this:

```
50 DATA 225,209,201,197,195,128
```

Run the program three times. Notice that there is no vertical space between the characters. Each connects to the character directly below it.

By making a larger matrix, you can design a larger character. When coding it, remember that it will be printed one line at a time across the page.

Make as many graphics characters as you wish before reading further in this chapter.

SWITCHING BACK TO STANDARD CHARACTERS

You can intermix dot graphics and standard characters just by switching modes at the printer. List the program and enter

```
140 REM ----- PRINT STANDARD CHAR
150 PRINT # 4,CHR$(15)"DONE"
160 CLOSE 4
```

Run it that way.

REPEATING GRAPHICS CHARACTERS

Control code CHR\$(26) is used to repeat a graphics pattern up to 255 times. The format is PRINT # *file n*, CHR\$(26) CHR\$(repetitions) pattern\$

Repeating a pattern of solid dots is a handy way to make a horizontal bar chart. Enter

```
NEW
10 REM DEMO REPEAT GRAPHICS
20 PRINT CHR$(147)
30 REM ----- SPECIFY CHAR
40 C$=CHR$(255)
90 REM ----- PRINT GRAPHIC
100 OPEN 4,4,0
110 PRINT # 4, CHR$(8);:REM GRAPHICS
120 PRINT # 4, CHR$(26)CHR$(100)C$:
    REM REPEAT 100 TIMES
130 CLOSE 4
```

Run it. It starts at the left margin and prints a vertical column of 7 dots, 100 times. When printed as a dot-graphics character, CHR\$(255) makes a vertical column of 7 dots.

For practice, change the program so it begins printing at dot column 100 and prints 200 columns.

When printed as a standard character, CHR\$(255) is a pi symbol in character set 1 and a checker board in character set 2. Write a routine to demonstrate that.

REVIEW

This chapter is an introduction to sprites and bit graphics. To learn more about programming sprites, you should write some sprite programs while this information is fresh in your mind. It becomes easier after you have done it a few times.

If you do that, I suggest that you use lines 0-170 of the demonstration program as a handy way to produce sprite specifications. Copy or print out the 63 decimal numbers in the array S0\$() on paper as a specification for each sprite in your program. Then write your program, using READ-DATA statements to poke the sprite specifications into memory.

If you make a really nifty sprite, you may want a copy of it on paper. Write another program to duplicate the sprite on paper.

20 Sound And Music

The Commodore 64 can produce sound and music using the speaker in the video monitor or TV set. Sound is commonly used for three purposes: beeping, making music and sound effects.

You can use beeps to guide the user of your programs. If the computer is waiting for a keyboard input, a polite beep is often useful and appreciated. If the user makes an invalid or incorrect response, an impolite beep is sometimes appropriate.

This chapter begins with a simple beep subroutine that you can use in your programs.

Sound is produced by poking and bit manipulation at memory locations. After beeping a few times, this chapter demonstrates methods of controlling the sound generators in the computer. It will play a little Beethoven.

This is another chapter in which you won't understand the beginning very well until you are nearly to the end. To produce sound, you have to control a lot of things. I will just ask you to plug in numbers to control them.

Gradually, you will understand more because I will explain more. At the end of this chapter, you will understand what you did at the beginning. You will be playing Beethoven on the computer, and your neighbors will be impressed.

BEEPS

Here is a subroutine that you can use to signal the user when the computer is waiting for a keyboard input. Or, you may use it to call the butler, or whatever you wish. You can use it in your programs without understanding how it works. Enter

```
NEW
999 REM BEEP SUBR
1000 FOR L=54272 TO 54296:POKE L,0:NEXT
1010 POKE 54296,15:REM VOL
1020 POKE 54273,60:REM PITCH
1030 POKE 54278,15
1040 POKE 54276,17:FOR T=1 TO 1000:NEXT:
      POKE 54276,0:REM T IS TIMER
1050 POKE 54296,0:RETURN
```

Be sure the numbers are typed correctly. Run it from the immediate mode by entering

GOSUB 1000

You should hear a beep. Line 1010 sets volume. The value being poked can be from 0 to 15. This line now calls for maximum volume. If it is too loud, or not loud enough, change the volume control on the display.

Change line 1010 to poke different volume values: 10, 5 and 0. Run it at each volume. Then change line 1010 back again so it pokes the number 15.

Line 1020 controls pitch. Change 60 to 6 and run it. Change it to 255 and run again. Set it at 100 and run again.

The loop at line 1040 controls the time duration of the beep. Change it to run from T=1 to 500. Run the subroutine. Change it to run from 1 to 50 and run it again. Set it to count to 300 and run again. That is probably a good duration for a beep. Line 1050 turns volume down to zero and returns to the calling routine.

You can change volume, pitch and beep duration produced by this subroutine to tailor it for your own programs.

Making Different Beeps—You may decide to use one beep to ask for a keyboard input and a different beep to signal an invalid input. You don't need two subroutines to do that.

One method is to change pitch to distinguish one type of beep from another. Set pitch in the main program. Then, when the subroutine is called, it will use whatever pitch the main program asked for. Let's use the variable name PITCH for pitch. Change line 1020 like this:

1020 POKE 54273,PITCH

Now the subroutine will use whatever pitch number is in memory with the variable name PITCH. The main program can "send" the pitch value to the subroutine by establishing a value for PITCH before calling the subroutine. Enter these lines to simulate the main program:

```
99 PRINT CHR$(147)
100 PRINT"ENTER A NUMBER FROM 0-9"
110 PITCH=50:GOSUB 1000:PRINT
120 GET A$:IF A$=""GOTO 120
130 IF ASC(A$)<48 OR ASC(A$)>57 THEN
    PITCH=10:GOSUB 1000:GOTO 120
140 PRINTA$" WAS ENTERED. THANK YOU":
    PITCH=60:GOSUB 1000:GOSUB 1000
150 END
```

Line 110 sets PITCH to 50 and calls subroutine 1000 to make a beep. That prompts the user to respond to line 100, in case he is looking out the window or taking a little nap. Line 130 is an error trap. If an invalid keystroke is made, PITCH is set to 10 and subroutine 1000 is called again. Then line 130 jumps back to 120 to get another input.

If a valid input is made, line 140 sets still another PITCH value and beeps twice to show appreciation. Programs that actively respond to the user are easier to run. Having a little fun in a program is usually OK.

Obviously, you can use the same method to set other variables before calling the subroutine. That way, you can control volume, pitch and duration of the beeps.

HOW IT WORKS

Sound is produced by a special sound chip in the computer. The memory locations used to control sound are 54272 to 54296. Sound is made by poking, and manipulating individual bits at those locations.

Those locations are unusual in that they cannot be read by a PEEK statement. POKE works, but PEEK gets a value of zero even if these memory locations are not zero.

The sound chip has three sound generators that can produce three different tones simultaneously. These are called *voices*. They are numbered from 1 to 3. You can use any or all of the three voices.

For each voice, you must tell the computer what the sound should sound like and how long it should last. There are several things that determine what we hear when a sound is produced.

Specifying all of the properties of a sound may seem complicated in the following discussion. At the end of the discussion is a summary table that will make the procedure easier to understand.

VOLUME

Volume controls the loudness of a sound. Volume is controlled by poking a number from 0 to 15 into location 54296. That sets the volume for all three voices. Value 0 turns the sound off.

FREQUENCY

Another thing that determines what a sound “sounds like” is the *frequency*, or *pitch*, of the sound. If you make a sound by plucking a guitar string, the string vibrates back and forth to make a sound wave in the air. Each vibration of the string is called a *cycle*. The number of vibrations in one second is called *cycles per second*. Frequency is stated in cycles per second.

Humans can hear frequencies from about 20 cycles per second to about 20,000 cycles per second. One abbreviation for cycles per second is *cps*. Another is the word *Hertz*—in honor of a scientist of that name. One Hertz (Hz) is one cycle per second.

In the beep program, frequency was set in the simplest possible way with a single poke to a single location. For complete control of frequency, two pokes are required into two locations in memory. That’s because the number used to specify frequency can range from 0 to 65535. A lower number produces a lower frequency.

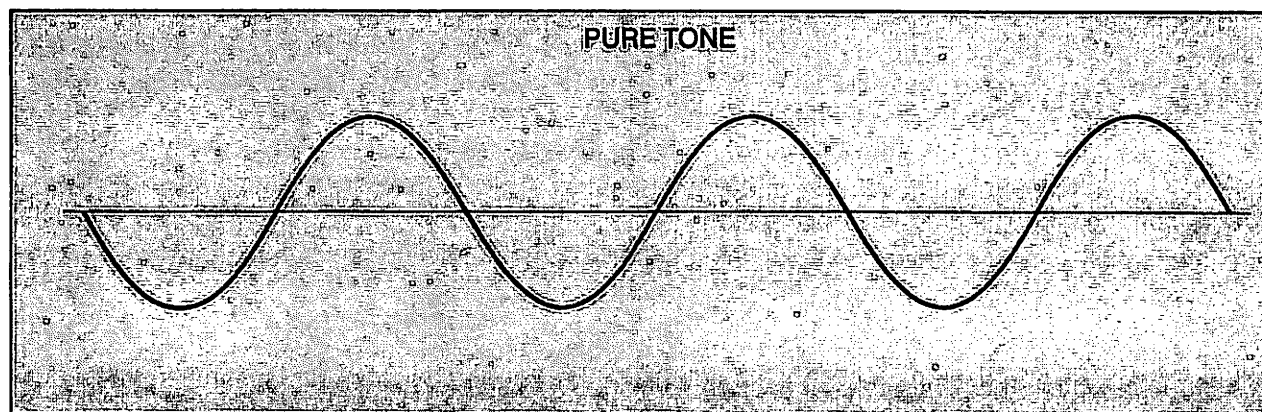
Numbers up to 65535 require two bytes. For voice 1, the low byte is stored at location 54272 and the high byte at 54273.

In the beep routine, only the high byte was poked. That will set a frequency, but does not allow complete control.

WAVEFORM

The tonal quality of a sound made by one kind of musical instrument is different from another kind of instrument. That difference is called *timbre*. Some of the difference in musical sounds is due to the way the musical instrument vibrates to produce sound waves.

Pure Tone—A *pure tone* is produced by vibrations that are steady, regular and unchanging. The pendulum on a clock moves that way, but it doesn’t vibrate fast enough to make an audible sound. The kind of vibration that makes a pure tone produces a sound wave called a *sine wave*. The accompanying sketch shows three cycles of a pure tone.

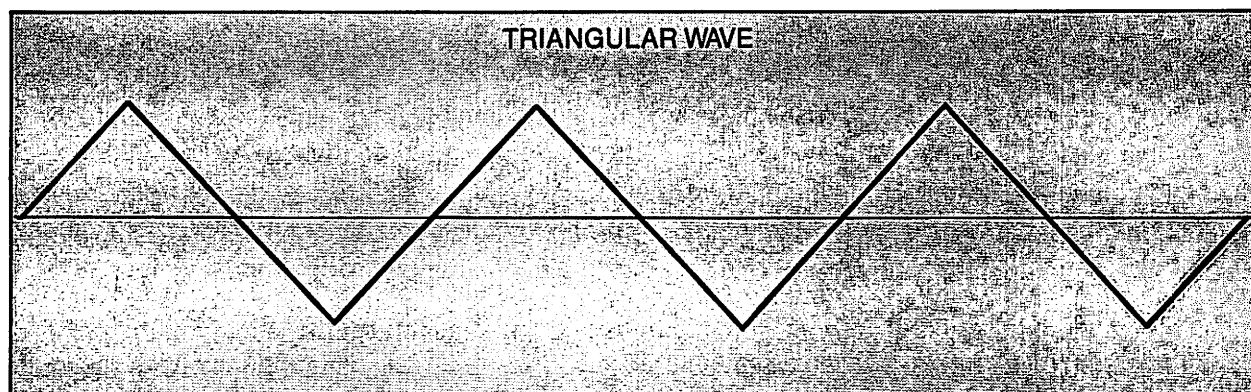


As you can see, a pure tone has a regular, unchanging waveform. This tone is so pure that it doesn't sound like a musical tone at all. It sounds artificial.

Most musical instruments produce sound waves that are not smooth and regular. They have sharp corners and irregularities that make the sound "rich and full-bodied." Irregularities in sound waveforms allow us to distinguish the sound of a guitar from the sound of a clarinet, even when they are playing the same note.

To simulate the irregular waveforms produced by various musical instruments, the Commodore 64 produces three different waveforms, plus one that is not musical.

Triangular Waveform—A *triangular wave* is shown in the accompanying sketch.

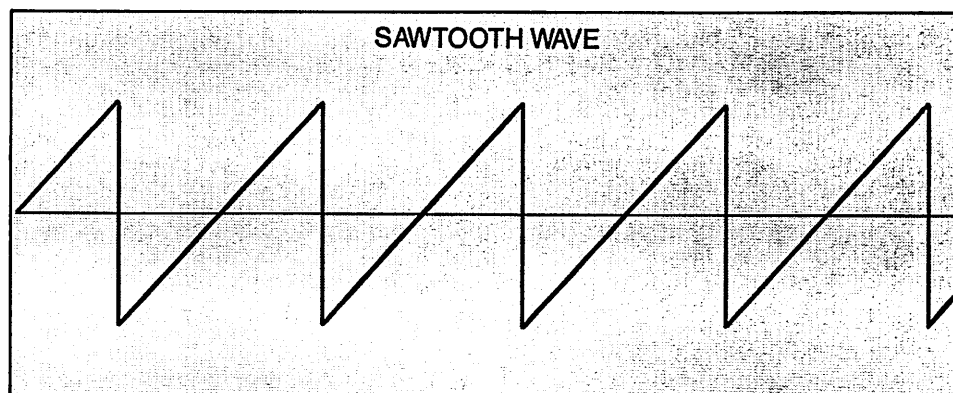


Even if the number of cycles per second is the same, a triangular wave sounds different than a pure tone. The reason is that a triangular wave consists of many different frequencies combined into one wave. A pure tone has only one frequency.

The lowest frequency of the sound made by a triangular waveform is the one you can see by looking at it. It is the number of triangles that occur every second. That frequency is called the *fundamental frequency*. The additional frequencies in that waveform are all higher than the fundamental. They are called *harmonics*. Harmonics are never as loud as the fundamental sound. They add richness and timbre to it in a subtle way.

You can't see the harmonic frequencies by looking at the triangular waveform, but they are there. They *cause* the triangular waveform to be shaped like a triangle. If there were no harmonics present, the triangular waveform would become a pure tone and look like a sine wave.

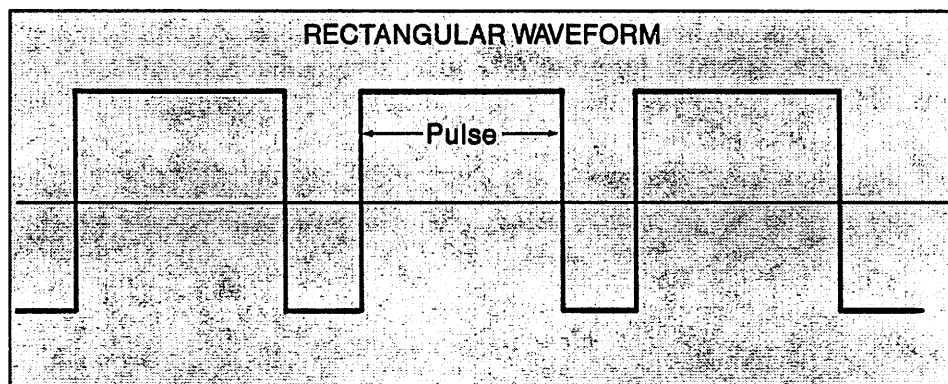
Sawtooth Waveform—The Commodore 64 also produces a *sawtooth waveform* that looks like the accompanying illustration.



Sound And Music

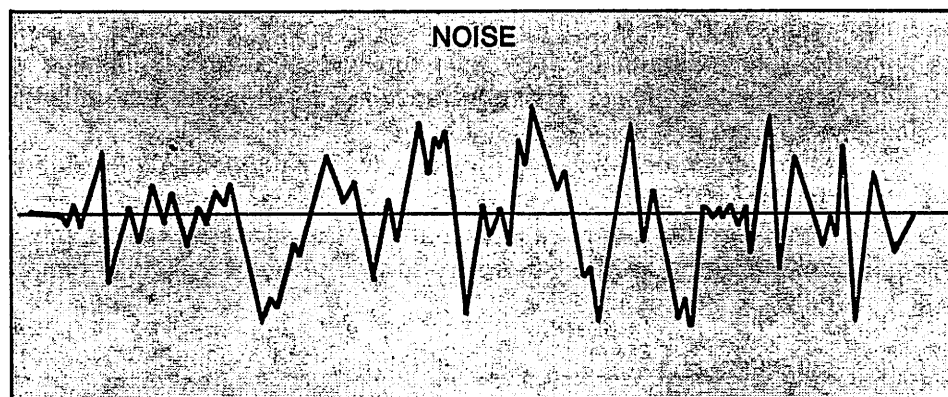
A sawtooth waveform has a different combination of harmonics, which causes it to have a different shape. The sound produced by sawtooth waveforms has a different quality than that produced by other waveforms.

Rectangular Waveform—The Commodore 64 also produces a rectangular waveform that looks like this:



This waveform sounds different than the others because it has still another combination of harmonics. A rectangular wave can be considered as a series of *pulses* rising upward from the lowest value of the wave. This drawing shows three pulses in a rectangular wave. The one in the center is marked.

Noise—One more waveform is available. It produces noise. It is so irregular that no repeating pattern can be seen:



This waveform doesn't have a musical quality. It can be used to sound like waves at the seashore, a rocket passing by, or an explosion. Commodore refers to this sound as *white noise*. The word *white*, when used to describe noise, means that the noise has all frequencies in it that can be heard—each at about the same loudness. It is analogous to the color white, which has all colors in it that can be seen.

Musical Tones—Of the three waveforms available to produce musical tones, the triangular is closest in shape to a sine wave. The triangular waveform makes the purest tone of these three waveforms—but it is not completely pure.

SELECTING THE WAVEFORM

To make a sound, the computer uses one of these waveforms and produces a series of sound waves at the selected frequency and volume. The waveform for each of the three voices is selected separately at separate memory locations, by poking number codes. The codes to select waveform follow:

CODE	WAVEFORM SELECTED
17	Triangle
33	Sawtooth
65	Rectangular
129	Noise

For voice 1, location 54276 is used to specify the waveform. A summary table with all location numbers used to control sound is shown later in this chapter.

Please list the beep program in memory. It uses voice 1. In the subroutine, at line 1040, the first statement pokes decimal 17 into location 54276. That selects the triangular waveform for voice 1.

The loop on that line, using T as the counter, determines how long the sound remains turned on.

The last statement on that line pokes decimal 16 into location 54276. That turns voice 1 off. Line 1050 turns down the volume and returns to the main program.

Run the program once or twice and listen carefully to the tones. Then change the first statement on line 1040 to poke the number 33 instead of 17. That will select a sawtooth waveform instead of a triangular waveform. The line should look like this:

```
1040 POKE 54276,33:FOR T=1 TO 150:NEXT:
      POKE 54276,0:REM T IS TIMER
```

Run the program again and notice the difference in timbre. Select white noise by changing the number 33 to 129 in line 1040. Run the program. Notice that different PITCH values affect the noise sound, but there is no musical tone.

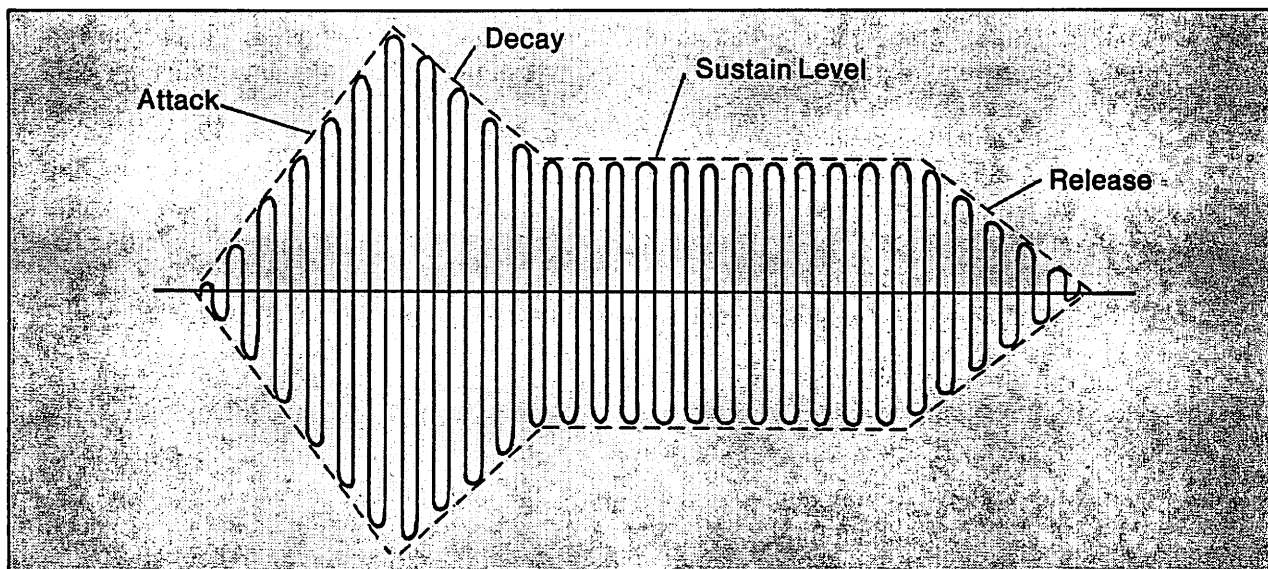
When a rectangular waveform is selected, you must also specify the pulse width. That will be demonstrated later.

ENVELOPE

Only three waves were shown in the preceding sketches of non-noise waveforms. Those drawings were magnified views.

A single note is made by many individual waves. The accompanying sketch represents a sound from its beginning to its end. The waveforms are compressed to fit them on the page.

Where the waveforms are taller, the sound is louder. The height of waves is called *amplitude*. Loudness is proportional to amplitude. This discussion concerns the overall shape or amplitude of the sound, as shown by the dotted lines. The overall outline or shape of a sound is called its *envelope*.



For many musical sounds, the envelope has a distinctive shape. For sounds made by plucking strings, such as a guitar, the sound rises quickly to its maximum amplitude, as soon as the sound begins. Then it decays, or falls off, to a lower amplitude, which it sustains for a time, without much change. Then it drops off to zero and the sound ends.

The computer allows control of four parts of the envelope. They are marked on the sketch.

Attack is how quickly the sound rises or increases in amplitude at the beginning.

Decay is how quickly it drops back down to a sustained amplitude.

Sustain is the amplitude or loudness of the sound after it has dropped down from its initial peak value. Typically, most of a musical note is at the sustain level.

Release is how quickly the note diminishes from the sustained level back to zero.

Attack, Decay, Sustain and Release are abbreviated ADSR.

Not all instruments produce a rapid attack. String instruments played with a bow may have a gradual buildup of amplitude at the beginning of a note. Reed instruments, such as a clarinet, also have a gradual attack.

Instruments with gradual attacks may not have much decay after the note is established. The amplitude may be nearly uniform throughout the note.

The release may be slow or abrupt, depending on the instrument and how it is played.

Instruments with cavities that resonate, such as a violin, have slow releases. A piano may have a slow release or an abrupt ending to each note because there are pedals to control the ending.

Standard electronic music synthesizers control those four properties of the sound envelope. The Commodore 64 controls the same four properties.

ADSR is controlled separately for each voice. It requires two bytes for each voice. For voice 1, Attack/Decay is controlled by location 54277. Sustain/Release is controlled by location 54288. Don't try to memorize these locations. They are all in a table later in this chapter.

The byte at location 54277 is divided into two 4-bit parts. Four binary bits can store numbers from 0 to 15. Attack is controlled by bits 4-7. It may have any value from 0 to 15. A value of 0 produces the fastest attack—the sound rises to maximum loudness quickly. If the value 15 is used, the attack is slow.

Decay is controlled by bits 0-3. It may have any value from 0 to 15. The number 0 produces a rapid decay. Code 15 produces a slow decay.

Even though we consider the codes for attack and decay as two separate numbers from 0-15, they must be combined into a single decimal value to be poked into the memory location that stores them.

Using A to represent the attack code, D to represent the delay code and AD to represent the value that is poked, the number to be poked is calculated by this formula: $AD = (16 * A) + D$

Multiplying A by 16 shifts it to the left four bits and places that code in bits 4 to 7 of the memory location.

The computer then reads bits 0-3 to get the decay value and bits 4-7 to get the attack value.

Sustain/Release is controlled in a similar way. Each may have values from 0 to 15. They are combined into a single decimal number by the method just shown for Attack/Decay. Sustain is the high four bits. Release is the low four bits: $SR = (16 * S) + R$.

For voice 1, Sustain/Release is poked into location 54278.

If a rectangular waveform is selected, one more specification is required. It is the relative width of the pulse, compared to the space between pulses. That is done by controlling 12 bits. For voice 1, the low 8 bits are in location 54274. The remaining 4 bits are bits 0-3 at location 54275. With 12 bits, numbers from 0 to 4095 may be stored.

Specifying the shape of the rectangular waveform requires two pokes. For voice 1, poke 0-255 into location 54274 and 0-15 into location 54275. The effect of this will be demonstrated.

The accompanying tables show all locations to control these properties of the three voices. They are similar, except for poke locations. The volume setting at location 54296 controls the three voices. It is shown in each table.

CONTROLLING VOICE 1

To Set	Location	What To POKE
Volume	54296	0-15 (0 is off)
Frequency	54272 & 54273 (0-65535)	Low byte in 54272 High byte in 54273
Attack/Decay	54277	0-15 Attack & 0-15 Decay $AD = (16 * A) + D$
Sustain/Release	54278	0-15 Sustain & 0-15 Release $SR = (16 * S) + R$
Waveform	54276	17 for triangle on/16 off (also turns 33 for sawtooth on/32 off sustain off) 65 for rectangular on/64 off 129 for noise on/128 off
Pulse shape	54274 & 54275	0-255 in 54274, 0-15 in 54275 (Rect. only)

CONTROLLING VOICE 2

To Set	Location	What To POKE
Volume	54296	0-15 (0 is off)
Frequency	54279 & 54280 (0-65535)	Low byte in 54279 High byte in 54280
Attack/Decay	54284	0-15 Attack & 0-15 Decay $AD = (16 * A) + D$
Sustain/Release	54285	0-15 Sustain & 0-15 Release $SR = (16 * S) + R$
Waveform	54283	17 for triangle on/16 off (also turns 33 for sawtooth on/32 off sustain off) 65 for rectangular on/64 off 129 for noise on/128 off
Pulse shape	54281 & 54282	0-255 in 54281, 0-15 in 54282 (Rect. only)

CONTROLLING VOICE 3

To Set	Location	What To POKE
Volume	54296	0-15 (0 is off)
Frequency	54286 & 54287 (0-65535)	Low byte in 54286 High byte in 54287
Attack/Decay	54291	0-15 Attack & 0-15 Decay $AD = (16 * A) + D$
Sustain/Release	54292	0-15 Sustain & 0-15 Release $SR = (16 * S) + R$
Waveform	54290	17 for triangle on/16 off (also turns 33 for sawtooth on/32 off sustain off) 65 for rectangular on/64 off 129 for noise on/128 off
Pulse shape	54288 & 54289	0-255 in 54288, 0-15 in 54289 (Rect. only)

PLAYING BY EAR

Setting the values of numbers in advance, so they can be used in calculations or to control something, is often called *setting parameters*. The best way to become familiar with the sounds that can be produced is to set the parameters and listen.

Demonstration—Here is a demonstration program that sets parameters for sound from the keyboard. It is fairly long, but it's a good way to become acquainted with the sound-making capabilities of your computer. After you use it as a demonstration program, I will add a routine that allows you to use the same program to write and save music or sound effects.

PLAN FOR SOUND DEMO PROGRAM

- 1) Set variables for memory locations to control voice 1.
- 2) Set parameters for voice 1 using subroutine.
- 3) POKE voice 1 data into memory.
- 4) Produce specified sound or note.
- 5) Display menu.
 - Repeat note.
 - Change note.
- 6) End

If you keep the plan in mind while entering the program, it will help you understand it. Please enter:

```
NEW
10 PRINT CHR$(147):REM SOUND DEMO
20 DIM TUNE(200,9):C=0:REM COUNTER
30 REM SET MEM LOCATIONS FOR VOICE 1
40 VOL=54296:FRQ=54272:WAVE=54276:
  AD=54277:SR=54278:PULSE=54274
50 C=C+1:NN=C:REM NEXT NOTE NUMBER
60 FOR I=54272 TO 54296:POKE I,0:NEXT:
  REM CLR SOUND
70 REM -----
```

Line 20 dimensions an array and zeros a counter that will be used later to store music or sound effects. Line 40 establishes variable names for the memory locations used to control voice 1. The other two voices are not used in this program.

Long variable names are used so line 40 is easier to understand. VOL means volume. FRQ is frequency. AD is Attack/Decay. SR is Sustain/Release. Later, the program uses only the first two characters in the long names, such as VO for volume and WA for waveform.

Line 50 operates a counter to assign note numbers as they are written. The value of the counter is taken as NN, which means *next note*. I will discuss this later.

Line 60 clears all of the locations that control sound by poking zero into each location. That takes care of the preliminaries. Enter

```
90 REM INPUT PARAMETERS FOR VOICE 1
100 PRINT"WRITE UP TO 200 NOTES":PRINT
110 PRINT"SET NEW VALUE OR PRESS RETURN"
115 PRINT
120 PRINT"NOTE NUMBER (1-"C") ="C:;
  GOSUB 900:IF NV<>-1 THEN C=C-1:NN=NV
121 REM:IF C NOT USED, RESET
125 PRINT"VOL (0-15) ="V1:;
  GOSUB 900:IF NV<>-1 THEN V1=NV
```

This program asks the user to set values for voice 1 and plays the sound that was set by those values—more than once if he wishes. If he likes the note, he can keep it in the program. If not, the user can change the parameters to make a different sound and then test the new sound. He can do that as much as he wishes until he finds just the right sound.

To change the parameters, the program displays the values that have already been set. The user can change each one by entering a new value, or not change it by just pressing RETURN.

Line 120 displays note numbers, starting at 1 and counting up to 200. The note counter doesn't have much use until later, when a routine will be added to allow writing music and storing it in an array. If you want to write a longer tune, change the number of notes that can be stored. I will discuss line 120 later.

In line 125, V1 is the value that controls volume for all three voices. The line begins by printing VOL (0-15). This shows the name of the variable being set and, in parentheses, the allowable range of values for that variable. Displaying the allowable range of the variable is helpful to the user. Then it displays the value of V1, so the user can see what has already been set and is already in memory for that variable.

After displaying the existing value, line 125 jumps to a subroutine at 900. The subroutine gets a keyboard input and returns to line 125 with a value for NV. NV means *new value*.

If the user doesn't change the existing volume setting by entering a new value while the subroutine runs, the subroutine returns the value -1 to the calling routine. This is a signal or flag to tell the program not to change the volume setting.

In line 125, NV is tested to see if it is -1. If it is -1, the IF statement in line 125 will not execute, so the value of V1 will not be changed.

If NV is not -1, a new value for V1 was input in the subroutine. Line 100 sets V1 equal to the new value, NV.

To see how the subroutine works, enter

```
890 REM GET NEW VALUE
900 NV$=" ":INPUT NV$
910 IF LEN(NV$)=0 THEN NV=-1:RETURN:
    REM PRESSED RETURN WITH NO INPUT
920 NV=VAL(NV$):REM ENTERED NEW VALUE
930 RETURN
```

List the program. The purpose of this subroutine is to enter a new value for NV or detect that the RETURN key was pressed without entering a new value.

To understand this routine, you should know what the RETURN key does in an INPUT statement. If a variable exists in memory, and a program line inputs a new value for it, two things can happen. If the user actually enters a new value, it replaces the old value in memory. If the user just presses RETURN, the old value in memory *remains unchanged*.

This routine takes NV\$ as the input. At line 900, NV\$ is set to a null. Then a new keyboard input is received by the variable NV\$.

If RETURN is pressed without pressing any other key first, NV\$ remains a null string of zero length. If something is typed and then the RETURN key is pressed, NV\$ does not have zero length.

At line 910, the length of NV\$ is tested. If it is zero, the user pressed RETURN without entering anything. The numeric variable NV is set to -1 and the subroutine returns to the main program. When NV is -1, that is a signal to the main program that a new value for NV *was not entered*. The program will continue to use the existing value.

If line 910 does not execute, NV\$ does not have zero length. Something was entered. Line 910 sets NV equal to the value of NV\$ and returns to the main program.

At line 125, the value of NV is tested. If it is not -1, then the volume-control variable, V1, is set to the value of NV, whatever it is. This changes the volume-control setting to the new value.

This program has a series of lines to input new values for the other variables that specify a sound. They all work the same way. List -125 and enter

```
130 PRINT"FREQ (1-65535) ="F1;;
    GOSUB 900:IF NV<>-1 THEN F1=NV
135 PRINT"ATTACK (0-15) ="A1;;
    GOSUB 900:IF NV<>-1 THEN A1=NV
140 PRINT"DECAY (0-15) ="D1;;
    GOSUB 900:IF NV<>-1 THEN D1=NV
145 PRINT"SUSTAIN (0-15) ="S1;;
    GOSUB 900:IF NV<>-1 THEN S1=NV
150 PRINT"RELEASE (0-15) ="R1;;
    GOSUB 900:IF NV<>-1 THEN R1=NV
155 PRINT"WAVEFORM (17/33/65/129) ="W1;;
    GOSUB 900:IF NV<>-1 THEN W1=NV
160 IF W1<>65 GOTO 180:REM NOT RECTANG
165 PRINT"PULSE (0-4095) ="P1;;
    GOSUB 900:IF NV<>-1 THEN P1=NV
170 HP=INT(P1/256):REM HI BYTE OF PULSE
175 LP=P1-(HP*256):REM LO BYTE
180 PRINT"TIME DURATION (1-65535) ="T1;;
    GOSUB 900:IF NV<>-1 THEN T1=NV
185 REM -----
```

This routine either gets a new value for each of these variables or retains the value that is already in memory.

If a rectangular waveform is selected at line 155, line 165 asks for the pulse specification for the rectangular wave. Lines 170 and 175 separate the pulse-specification number into low byte and high byte.

When the program reaches line 185, a value is in memory for each variable required to specify sound. Because voice 1 is being used, the variable names use the number 1. V1 is volume. F1 is frequency. A1, D1, S1 and R1 are the ADSR specifications. W1 is waveform. P1 is the pulse specification if W1 selected a rectangular waveform. T1 is time duration.

Test—Enter this temporary line to test the program up to here:

```
190 PRINT V1,F1,A1,D1,S1,R1,W1,P1,HP,LP,
    T1
```

Run the program. For note number, press RETURN. Enter the highest allowable value for each variable except waveform. For WAVE, enter 65. That selects the rectangular waveform, which requires a pulse specification. Use the maximum value for PULSE.

The display should show the values you entered in this form:

15	65535	15	15
15	15	65	4095
15	255	65535	

On the bottom row, the numbers 15 and 255 are the high- and low-byte values for pulse, which was entered as 4095. If you see all of these values, the program is probably OK. If not, look for typos. When the program works correctly, delete temporary line 190.

Poking Parameters—Now, the program is ready to poke those values into memory to create that sound. List the program. Remember that long variable names, such as VOL for volume and FRQ for frequency were set at line 40. These are the memory locations that control those properties. This routine uses the first two letters of those long variable names. VO is the memory location for

volume, and so forth. The numbers poked into those locations end in 1 because voice 1 is being programmed. V1 is the volume setting for voice 1. It goes into memory location VO. Enter

```

190 REM POKE VALUES FOR VOICE 1
200 POKE VO,V1:REM VOLUME
210 FHI=INT(F1/256):FLO=F1-(FH*256):
    REM GET FREQ HI & LO BYTES
220 POKE FR,FLO:POKE FR+1,FHI
230 POKE AD,(16*A1)+D1:REM HI/LO NYBBLE
240 POKE SR,(16*S1)+R1:REM HI/LO
250 IF W1=65 THEN POKE PU,LP:
    POKE PU+1,HP:REM LO & HI PULSE SPEC
260 REM -----

```

FR is location 54272, the memory location for the *low byte* of the two-byte frequency number. The high byte goes into the next higher memory location, FR + 1.

At line 210, voice 1 frequency, F1, is divided into low byte and high byte. FLO is low byte. FHI is high byte. At line 220, these two bytes are poked into the correct memory locations by the statement POKE FR,FLO:POKE FR+1,FHI

At line 230, Attack/Decay values are poked into a single memory location. Attack is the high nybble at that location, Decay is the low four bits. Line 230 combines the Attack/Decay values into a single byte and pokes the result into memory location AD.

Line 240 does a similar operation to poke the four-bit values for Sustain and Release into location SR in memory.

That routine pokes everything except waveform into memory. Poking waveform starts the sound. Enter

```

290 REM TEST SOUND
300 POKE WA,W1:REM BEGIN
310 FOR S=1 TO T1:NEXT:REM SUSTAIN
320 POKE WA,W1-1:REM END SUSTAIN
330 FOR R=1 TO T1/2:NEXT:REM RELEASE
340 REM -----
350 STOP

```

At line 300, W1 is one of the four numbers used to select waveform. For example, 17 selects a triangular waveform. When all of the sound parameters have been poked into memory, poking the waveform number starts the sound. It will rise during the attack interval, drop during the delay interval, and then remain at the *sustain* value until sustain is turned off.

To turn off the sustain level, poke the waveform number minus one back into the same location. If 17 was poked to start the sound, poking 16 into the same location will stop the sustain level. Then the release action begins.

There are two time durations. One is the time for attack, decay and sustain. I will refer to this as *sustain* time, even though it also includes the initial attack and decay times. After the sustain time ends, the release time begins.

These time intervals are controlled by two loops. The duration of the sustain level is controlled by running a timer loop using S as the counter while the sound is at the sustain level. The computer will produce the sustain level indefinitely, until it is turned off by poking the waveform number minus one into memory.

The sustain-level timing loop is at line 310. The value of T1 determines how long the sound will remain at the sustain level. When the timer loop finishes running, line 320 turns the sustain level off by poking W1 - 1 into location WA.

The sequence is: Line 300 begins the sound. Line 310 holds it at sustain level until that loop

finishes running. Then line 320 turns off the sustain level.

When sustain is turned off, the computer automatically begins the release action, which brings the loudness down from sustain level to zero. Rate of release is determined by the release value that is part of the Sustain/Release specification.

Line 330 provides a time interval for the release by running a second loop using R as the counter. It prevents the computer from making another note or turning the volume down until the release event has finished.

This program counts to T1 in the main timing loop that holds the sustain level. It allows half as much time for the release action. The release timer counts to T1/2 as the time interval for release. That may be too much or too little. It's an arbitrary value for this demonstration.

Test—Run the program. It will show you that you can enter any note number in the range of 1-1. That means, you must enter the first note. The note number is set to 1. After you have entered some notes, such as 15 notes, this display will show that you can enter any note number in the range of 1-15. That allows you to back up and change a note entered earlier.

Leave the note counter set to 1 by pressing RETURN. Then enter these values:

VOL: 15
FREQ: 4291
ATTACK: 5
DECAY: 5
SUSTAIN: 5
RELEASE: 5
WAVEFORM: 17
TIME DURATION: 500

When you enter time duration, the program should immediately make a musical note of medium pitch for about 1/2 second. The note is middle C on a piano, if that helps identify it.

To make it sound like a piano, run the program again and enter this data for note 1:

VOL: 15
FREQ: 4291
ATTACK: 0
DECAY: 9
SUSTAIN: 0
RELEASE: 0
WAVEFORM: 65
PULSE: 255
TIME DURATION: 500

That note should sound more like middle C on a piano because of the different ADSR settings and the rectangular waveform. Adding another routine to this program makes it easy to change parameters and explore what can be done with sound. Delete the temporary stop at line 350. Then enter

```
490 REM MENU
500 PRINT CHR$(19):FOR I=1 TO 15:PRINT:
    NEXT:REM LOCATE MENU
505 PRINT" T - TEST NOTE AGAIN"
510 PRINT" C - CHANGE NOTE"
515 PRINT" P - PLAY TUNE USING NOTE"
520 PRINT" Q - QUIT WITHOUT SAVING TUNE"
525 REM MAKE CHOICE - - - - -
530 GET A$:IF A$="" GOTO 530
535 IF A$="T" GOTO 300:REM TEST AGAIN
```

(Program continued on next page.)

```

540 IF A$="C" THEN PRINT CHR$(147):
      GOTO 50:REM CHANGE SAME NOTE
545 IF A$="P" GOTO 600:REM USE NOTE
550 IF A$="Q" THEN POKE VO,0:END
555 GOTO 530:REM TRAP
560 REM -----

```

At line 540, if C is pressed, the program jumps back to line 50 to change the note that was just played. List 50-125. At line 50, the counter, C, is incremented. The next note number, NN, is set equal to C. That prepares the program to input the next new note by assigning an unused new value of C to NN.

Suppose the new value of C and NN is 6. At line 120, the user does not accept the number, 6, offered by the display because he wants to change a note that was entered earlier. Instead, he enters the number 3, because he wants to change note 3. Actually, any earlier note number can be entered, if some other note is to be changed.

At line 120, NV will not be -1 because a value was entered in subroutine 900. Therefore, the IF statement will execute. The counter C will be reset to C-1. That sets it back to 5. That is done because the new value of C is not being used. The next time the program loops back to line 50, C will be incremented again. It will be set to 6 again, in case the user wants to enter a new note rather than change an old one.

At line 120, after C is reset, NN is set to the number entered in the subroutine, NV. In this example, it will be set to 3. Then new data can be entered for any specification for note 3. That can be repeated as often as desired.

Test—To test this routine, run the program. Enter these settings for note 1:

```

VOL: 15
FREQ: 4291
ATTACK: 12
DECAY: 13
SUSTAIN: 0
RELEASE: 0
WAVEFORM: 17
TIME DURATION: 5000

```

You should hear a slow rise during the attack and a slow decay to zero because the sustain level is zero. Press T on the menu. The note should repeat. Don't press P. That part of the program isn't written yet.

Press C to change this note. The display shows that the program is ready to receive specifications for note 2. Type the number 1 and press RETURN. That causes the program to accept new specifications for note 1. You can change only the values that you want to change and leave the others set by pressing RETURN without entering new values.

Set sustain to 5 and release to 9. Don't change any other values. You should hear the sound rise, then decay to the sustain level. The sustain level is held for a short time, then the release action takes place and lowers the volume slowly down to zero. To hear it again, press T.

ATTACK/DECAY/SUSTAIN/RELEASE

For attack, decay and release, setting higher values means more gradual changes. For sustain, a higher number means a higher sustain level. Play the note again by pressing T.

Press C to change the values. Set the note number to 1 again. Change attack to 0 and leave everything else as it was. Notice that the quick attack sounds like a note on a string instrument.

Press C. Set the note number to 1 again. Set attack to 12 and decay to zero. Notice the gradual rise and the instant decay to the sustain level.

Press C. Set the note number to 1 again. Change sustain to 0. You can hear the gradual attack, and the instant decay to zero because the sustain level is zero. When sustain is zero, the release setting doesn't matter because there is nothing to release.

Press C. Set the note number to 1 again. Change the sustain level to 15. At the maximum setting of 15, the sustain level is the same as the peak level. Even though a decay is specified, it decays from 100% to 100%, which is no change. Now, you can hear the slow release again.

Press C. Set the note number to 1 again. Enter these values:

VOL: 15

FREQ: 4291

ATTACK: 10

DECAY: 13

SUSTAIN: 7

RELEASE: 12

WAVEFORM: 33

TIME DURATION: 4000

You should hear all four parts of the envelope. You haven't tested Q on that menu yet. Press Q. The program should end and the READY prompt should appear.

TIMING CONSIDERATIONS

Attack, delay and release each take a specific amount of time. If the timing loops don't allow that much time, the sound will end abruptly before completion of the programmed Attack/Decay or the release. The accompanying table shows the amount of time required for each setting.

TIME INTERVALS REQUIRED FOR ATTACK, DELAY AND RELEASE (In milliseconds, 1/1000 second)			
POKE Value	Attack	Decay	Release
0	2	6	6
1	8	24	24
2	16	48	48
3	24	72	72
4	38	114	114
5	56	168	168
6	68	204	204
7	80	240	240
8	100	300	300
9	250	750	750
10	500	1500	1500
11	800	2400	2400
12	1000	3000	3000
13	3000	9000	9000
14	5000	15000	15000
15	8000	24000	24000

Notice that higher values require very long times to run to completion. 24000 milliseconds is 24 seconds!

The total time allowed for a note or sound is the sum of the count reached by the sustain timer loop plus the release-timer loop. For example, if the sustain loop counts to 250, and the release timer counts to 50, the total time will be a count of 300. For both of these loops, a count of 1 is approximately equal to 1 millisecond. A total count of 300 will occur in 300/1000 second, which is 0.3 second.

If you are writing music, the rhythm or meter will normally govern the timing. If you assign a count of 300 for a full note, the complete ADSR should happen in that length of time. If not, you

may hear an abrupt cutoff of the decay or release events. If that happens, you should probably decrease the time required by one or more of the ADSR events by changing the control number for that event. Or, you can increase the time allowed by one or both of the loops.

If you are making sound effects, the rise and fall times are probably more important than the duration of the sound effect. Make the loops run long enough to allow the effect that you want.

EXPERIMENT

Spend as long as you wish with this program. You don't have to reset the note counter each time. Let it run. I suggest that you experiment with all four waveforms. For each waveform, use several combinations of ADSR settings. Try a range of frequencies, from very low to very high. You will discover that the computer can make an amazing variety of sounds.

MAKING MUSIC

With a little more programming, we can convert the demonstration program into a music maker or a program that can make a complicated sound effect. The program will store up to 200 notes entered from the keyboard in an array in memory. It allows you to test each note as you enter it and change it if you wish, before it goes into the array.

It will play the tune or sound effect that you have stored in the array. If you hear a bad note, it lets you change that note in the array. If you wish, it will store the tune on tape or disk, so you can play it again later.

There are four routines to add. One stores the notes in an array in memory. Another plays the tune or sound effect that is stored in the array. Another puts the array on tape or disk, to save the tune. A menu lets you choose among those options.

Let's begin with the array. Please list -500. The routine that ends at line 340 pokes the sound values into memory after they are entered. This allows you to hear the note. The menu at line 500 allows you to jump back to the beginning of the program and change the note if you wish.

This new routine fits between those two. After the note specifications are entered and you hear the note played, this routine puts the note specifications into an array in memory. Then, the menu is displayed.

If you decide to back up and change the note, that's OK. The new note specifications will replace whatever is in the array for that note. Enter

```
390 REM PUT EACH NOTE IN TU(200,9)
400 REM NN IS NOTE ROW IN ARRAY. THIS
    ALLOWS EDIT NOTE AND PUT IT BACK
405 CO=1:REM CO COUNTS COLUMNS 1-9
```

The array to be filled was dimensioned at line 20 as TUNE(200,9). It stores a tune with 200 notes. It takes 9 items to specify each note, so the array has 200 rows, with 9 items on each row. The items will be obvious from the program lines. Enter

```
410 TU(NN,CO)=
```

Then duplicate that by overtyping line numbers 415, 420, 425, 430, 435 and so forth, through line 450. Then finish each line so the result looks like this:

```
410 TU(NN,CO)=V1:CO=CO+1
415 TU(NN,CO)=FL:CO=CO+1
420 TU(NN,CO)=FH:CO=CO+1
425 TU(NN,CO)=(16*A1)+D1:CO=CO+1
430 TU(NN,CO)=(16*S1)+R1:CO=CO+1
435 TU(NN,CO)=W1:CO=CO+1
440 TU(NN,CO)=LP:CO=CO+1
445 TU(NN,CO)=HP:CO=CO+1
450 TU(NN,CO)=T1
460 REM -----
```

List -500 and check your typing. This routine operates each time a sound is specified and then produced by the routine beginning at line 300.

Each sound has a note number, NN, that is assigned when the specifications for that note are entered. Each statement in this routine puts a value into the two-dimensional array TU(NN,CO), which was dimensioned as TUNE(200,9). There can be 200 values of NN and 9 values of CO. This ignores the zero elements of the array, even though they could be used.

The note number, NN, is the same in each of these statements. That means the row number is the same. If note 23 was just played, NN is 23. This routine puts the sound specifications into row 23 of the array.

The counter CO is used to generate column numbers. It is set to 1 at line 410. Each time it is used to place a value in the array, it is increased by 1 by a statement CO=CO+1. CO counts from 1 to 9. The array locations are TU(NN,1) to TU(NN,9). NN is the note number for a single note.

Starting at line 410, these values are plugged into the array for each note:

LINE	ARRAY COLUMN	VALUE	WHAT IT IS
415	1	V1	Volume setting
415	2	FL	Low byte of frequency spec
420	3	FH	High byte of frequency spec
425	4	(16*A1)+D1)	Combined Attack/Decay value
430	5	(16*S1)+R1)	Combined Sustain/Release value
435	6	W1	Waveform value
440	7	LP	Low byte of pulse spec if rect.
445	8	HP	High byte of pulse spec
450	9	T1	Counter for sustain loop

No value is placed in the array for the release-loop counter because this program derives it from T1.

Test—Enter this temporary line to test the program up to here:

```
491 FOR I=1 TO 9:PRINT TU(NN,I),:NEXT:  
STOP
```

Run the program. Enter these values for note 1:

```
VOL: 15  
FREQ: 6000  
ATTACK: 7  
DECAY: 7  
SUSTAIN: 5  
RELEASE: 5  
WAVEFORM: 65  
PULSE: 2000  
TIME DURATION: 500
```

The display should show the values you entered in this form:

```
15      112      23      119  
85      65       208      7  
500
```

If you see these values, the program is probably OK. If not, look for typos. When the program works correctly, delete temporary line 491.

PLAYING THE ENTIRE TUNE

Please list -600. As notes or sounds are entered, tested and accepted by the user, they are placed in the array TU() in numerical order. Then a menu is displayed by the lines beginning at 500. The only choice on this menu that has not been used is P, PLAY THE TUNE. It hasn't been used yet because there was no array to store the tune. Now there is an array but the routine to play it hasn't been written yet.

At line 545, when P is selected, the program jumps ahead to line 600. What is to be done is read the array, one note at a time, and poke the values into memory for each note. That will play the tune or sound effect that is stored in the array. The number of notes in the array is equal to C, the value of the note counter.

After the tune has been played, another menu offers several options. You can play it again, add another note, change one of the existing notes, save the tune to tape or disk, or quit without saving it.

Here is a listing of the entire program. Lines 590-885 have not been entered. When you get it working, it is fun to run.

I invited a musician who plays a synthesizer to run this program. He had never operated a computer before, but he recognized ADSR and the other parts of the program instantly. In about two minutes, he was playing beautiful music. Anyway, I thought it was beautiful because I wrote the program. As a musician, I can just barely play the radio.

Please enter lines 590-885 and then I will discuss the added routines.

```
10 PRINT CHR$(147):REM SOUND DEMO
20 DIM TUNE(200,9):C=0:REM COUNTER
30 REM SET MEM LOCATIONS FOR VOICE 1
40 VOL=54296:FRQ=54272:WAVE=54276:
   AD=54277:SR=54278:PULSE=54274
50 C=C+1:NN=C:REM NEXT NOTE NUMBER
60 FOR I=54272 TO 54296:POKE I,0:NEXT:
   REM CLR SOUND
70 REM -----
90 REM INPUT PARAMETERS FOR VOICE 1
100 PRINT"WRITE UP TO 200 NOTES":PRINT
110 PRINT"SET NEW VALUE OR PRESS RETURN"
115 PRINT
120 PRINT"NOTE NUMBER (1 -"C") ="C:;
   GOSUB 900:IF NV<>-1 THEN C=C-1:NN=Nv
121 IF C NOT USED, RESET
125 PRINT"VOL (0-15) ="V1:;
   GOSUB 900:IF NV<>-1 THEN V1=NV
130 PRINT"FREQ (1-65535) ="F1:;
   GOSUB 900:IF NV<>-1 THEN F1=NV
135 PRINT"ATTACK (0-15) ="A1:;
   GOSUB 900:IF NV<>-1 THEN A1=NV
140 PRINT"DECAY (0-15) ="D1:;
   GOSUB 900:IF NV<>-1 THEN D1=NV
145 PRINT"SUSTAIN (0-15) ="S1:;
   GOSUB 900:IF NV<>-1 THEN S1=NV
150 PRINT"RELEASE (0-15) ="R1:;
   GOSUB 900:IF NV<>-1 THEN R1=NV
155 PRINT"WAVEFORM (0-15) ="W1:;
   GOSUB 900:IF NV<>-1 THEN W1=NV
160 IF W1<>65 GOTO 180:REM NOT RECTANG
```

(Program continued on next page.)

```
165 PRINT"PULSE (0-4095) ="P1;;
    GOSUB 900:IF NV<>-1 THEN P1=Nv
170 HP=INT(P1/256):REM HI BYTE OF PULSE
175 LP=P1-(HP*256):REM LO BYTE
180 PRINT"TIME DURATION (1-65535) ="T1;;
    GOSUB 900:IF NV<>-1 THEN T1=Nv
185 REM -----
190 REM POKE VALUES FOR VOICE 1
200 POKE VO,V1:REM VOLUME
210 FHI=INT(F1/256):FLO=F1-(FH*256):
    REM GET FREQ HI & LO BYTES
220 POKE FR,FLO:POKE FR+1,FHI
230 POKE AD,(16*A1)+D1:REM HI/LO NYBBLE
240 POKE SR,(16*S1)+R1:REM HI/LO
250 IF W1=65 THEN POKE PU,LP:
    POKE PU+1,HP:REM LO & HI PULSE SPEC
260 REM -----
290 REM TEST SOUND
300 POKE WA,W1:REM BEGIN
310 FOR S=1 TO T1:NEXT:REM SUSTAIN
320 POKE WA,W1-1:REM END SUSTAIN
330 FOR R=1 TO T1/2:NEXT:REM RELEASE
340 REM -----
390 REM PUT EACH NOTE IN TU(200,9)
400 REM NN IS NOTE ROW IN ARRAY. THIS
    ALLOWS EDIT NOTE AND PUT IT BACK
405 CO=1:REM CO COUNTS COLUMNS 1-9
410 TU(NN,CO)=V1:CO=CO+1
415 TU(NN,CO)=FL:CO=CO+1
420 TU(NN,CO)=FH:CO=CO+1
425 TU(NN,CO)=(16*A1)+D1:CO=CO+1
430 TU(NN,CO)=(16*S1)+R1:CO=CO+1
435 TU(NN,CO)=W1:CO=CO+1
440 TU(NN,CO)=LP:CO=CO+1
445 TU(NN,CO)=HP:CO=CO+1
450 TU(NN,CO)=T1
460 REM -----
490 REM MENU
500 PRINT CHR$(19):FOR I=1 TO 15:PRINT:
    NEXT:REM LOCATE MENU
505 PRINT:PRINT"T - TEST NOTE AGAIN"
510 PRINT"C - CHANGE NOTE"
515 PRINT"P - PLAY TUNE USING NOTE"
520 PRINT"Q - QUIT WITHOUT SAVING TUNE"
525 REM MAKE CHOICE -----
530 GET A$:IF A$=""GOTO 530
```

(Program continued on next page.)

```
535 IF A$="T" GOTO 300:REM TEST AGAIN
540 IF A$="C" THEN PRINT CHR$(147):
      GOTO 50:REM CHANGE SAME NOTE
545 IF A$="P" GOTO 600:REM USE NOTE
550 IF A$="Q" THEN POKE VO,0:END
555 GOTO 530:REM TRAP
560 REM -----
590 REM PLAY IT
600 PRINT CHR$(147)
605 FOR PL=1 TO C:PRINT"PLAYING NOTE"PL
610 POKE VO,TU(PL,1)
615 POKE FR,TU(PL,2)
620 POKE FR+1,TU(PL,3)
625 POKE AD,TU(PL,4)
630 POKE SR,TU(PL,5)
635 IF TU(PL,6)=65 THEN
      POKE PU,TU(PL,7):POKE PU+1,TU(PL,8)
640 POKE WA,TU(PL,6)
645 FOR S=1 TO TU(PL,9):NEXT
      REM RUN SUSTAIN LOOP
650 POKE WA,TU(PL,6)-1
655 FOR R=1 TO TU(PL,9)/2:NEXT
      REM RUN RELEASE LOOP
660 NEXT PL
665 REM -----
690 REM MENU
700 PRINT CHR$(147)
710 PRINT"P - PLAY AGAIN"
715 PRINT"A - ADD OR CHANGE NOTE"
720 PRINT"S - SAVE TUNE"
725 PRINT"Q - QUIT WITHOUT SAVING TUNE"
730 GET A$:IF A$="" GOTO 730
735 IF A$="P" GOTO 600
740 IF A$="A" THEN PRINT CHR$(147):
      GOTO 40
745 IF A$="S" GOTO 800
750 IF A$="Q" THEN POKE VO,0:END
755 GOTO 730
760 REM -----
790 REM SAVE TUNE ROUTINE GOES HERE
      AND ENDS PROGRAM
885 END
890 REM GET NEW VALUE
900 NV$=" ":INPUT NV$
```

(Program continued on next page.)

```
910 IF LEN(NV$)=0 THEN NV=-1:RETURN:
    REM PRESSED RETURN WITH NO INPUT
920 NV=VAL(NV$):REM ENTERED NEW VALUE
930 RETURN
```

Lines 600-660 read the array TU() one note at a time and poke the values into memory to play that note. Line 605 sets up a loop to do that. It uses the counter PL, which means *play*. It counts from 1 to C. C is the value of the note counter used earlier in the program and therefore the number of notes or in the tune. The notes can be sound effects.

Lines 700-755 are a menu. You can play it again or go back and change or add notes. You can save the tune or sound effect, if you write the segment that line 745 jumps to. Or else, you can press Q and abandon the whole project.

There is one routine that is not written—lines 800-880. It is intended to save the tune or sound effect to disk or tape. This program will work without that routine, but you can't save the tune or sound effect until you write the routine to save it.

How it is written depends on which way you will save it. I will leave that up to you. Chapters 15 and 16 should have given you enough information to write that routine. It is relatively simple—just use a loop to put the array on disk or tape.

You may decide not to save it. That's OK. You can still run the program and write music or sound effects. If you do save it, then you will have to write another program to read it back and play the tune or sound effect again. This demonstration program can be modified to do that.

BEETHOVEN

I promised you some Beethoven. Here it is. The accompanying table shows 15 well-known notes with frequency and time duration for each. Run the program and enter the notes and time durations. For the other values, such as waveform, use whatever values you wish. Then play the tune.

NOTE NUMBER	FREQUENCY	TIME DURATION
1	4050	300
2	4050	300
3	4291	300
4	4817	300
5	4817	300
6	4291	300
7	4050	300
8	3608	300
9	3215	300
10	3215	300
11	3608	300
12	4050	300
13	3608	450
14	3215	150
15	3215	600

The name of that tune is *Ode To Joy*. If you are a musician, you can write music using this program if you have a list of frequency numbers for each note on the scale. They are in the table on the next page.

FREQUENCIES FOR EACH NOTE

Octave	Note	Frequency (Hertz)	Octave	Note	Frequency (Hertz)
0	C	268	4	C	4291
0	C#	284	4	C#	4547
0	D	301	4	D	4817
0	D#	318	4	D#	5103
0	E	337	4	E	5407
0	F	358	4	F	5728
0	F#	379	4	F#	6069
0	G	401	4	G	6430
0	G#	425	4	G#	6812
0	A	451	4	A	7217
0	A#	477	4	A#	7647
0	B	506	4	B	8101
1	C	536	5	C	8583
1	C#	568	5	C#	9094
1	D	602	5	D	9634
1	D#	637	5	D#	10207
1	E	675	5	E	10814
1	F	716	5	F	11457
1	F#	758	5	F#	12139
1	G	803	5	G	12860
1	G#	851	5	G#	13625
1	A	902	5	A	14435
1	A#	955	5	A#	15294
1	B	1012	5	B	16203
2	C	1072	6	C	17167
2	C#	1136	6	C#	18188
2	D	1204	6	D	19269
2	D#	1275	6	D#	20415
2	E	1351	6	E	21629
2	F	1432	6	F	22915
2	F#	1517	6	F#	24278
2	G	1607	6	G	25721
2	G#	1703	6	G#	27251
2	A	1804	6	A	28871
2	A#	1911	6	A#	30588
2	B	2025	6	B	32407
3	C	2145	7	C	34334
3	C#	2273	7	C#	36376
3	D	2408	7	D	38539
3	D#	2551	7	D#	40830
3	E	2703	7	E	43258
3	F	2864	7	F	45830
3	F#	3034	7	F#	48556
3	G	3215	7	G	51443
3	G#	3406	7	G#	54502
3	A	3608	7	A	57748
3	A#	3823	7	A#	61176
3	B	4050	7	B	64814

ADVANCED MUSIC PROGRAMMING

Instead of saving a tune or sound effect on disk or tape as a data file, you can put the numbers into a program as DATA statements. Then, the program can read the data and play that tune. With perhaps 9 numbers for each note or sound, the DATA statements will become very long unless the tune or effect uses only a few notes.

There are ways to reduce the amount of numbers required to specify a note by combining some of them using a coding scheme. Then, the program can decode the number to get several values to poke into memory.

The *Commodore Programmer's Reference Guide*, published by Commodore, shows one way to do that. In addition, it shows sample programs that control all three voices simultaneously.

It also shows how to use filters built into the computer to alter the tone quality of sounds. Filters work like tone controls on a radio or stereo.

For some wild sound effects, you can change the frequency and other specifications of a sound while the sound is playing. This is called *changing the specs dynamically*. Use a loop to change the value in memory while the note is playing. There is a demonstration of that technique in the *Commodore Programmer's Reference Guide*.

SUMMARY

With the information in this chapter, you can write tunes and make sound effects. Playing three voices simultaneously uses exactly the same techniques, except that the program is three times as complicated. The demonstration program in this chapter anticipates using more voices by coding the number values for voice 1 as F1, and so forth. For voice 2, use F2, and so forth. If you want to try more voices, expand the demo program.

21 Programming And Debugging Techniques

This book has demonstrated a great variety of programming techniques, including dot graphics, music and sound effects. You should consider the techniques demonstrated here as tools to be used when you need them.

Few programs will use all of the capabilities of the Commodore 64. Most programs will use several of them. For example, most programs should take advantage of color—at least to make menus and displays attractive and easy to understand. Many programs benefit from sound, even if it is a simple beep or two as audible signals to the user.

If you made it this far, and are not just peeking back here to see if the story ends, congratulations. It was a long trip.

Making it to here shows that you are a goal-oriented person—one who decides to do something and then does it. Programming is done by goal-oriented people. Writing a program gets you into frustrations and tedious labor, usually with nobody to help you solve problems or get out of trouble. Nevertheless, for those who stick to their goals and write programs, the rewards are a feeling of accomplishment and satisfaction. And, worthwhile programs.

MAKING A PLAN

The first step in writing a program is to decide what you want it to do. Professional programmers work very hard at this because a program that ends up doing the wrong things is worthless. They interview people who will use the program to find out what is expected and how the user wants the program to work. You should do no less, even if the program is being written just for one user—yourself.

The next step is to make a plan and write it down. How you write it is not very important if you are just learning to program. A simple list of things to be done in the program, in the order that they will be done, is enough.

Flowcharts—Some experienced programmers use diagrams called *flowcharts* to show how a program works. Flowcharts use boxes with special shapes and notes about the program inside the boxes. The shape of a box shows what that part of the program does—such as a decision point or branch point in a program.

Textbooks and some programming instructors emphasize using flowcharts as a way of planning a program. Some say that a flowchart is necessary before you start writing the program.

Experienced programmers think of programs in chunks or segments. When you think that way, the program segments are easy to visualize and put in boxes on a flowchart.

Beginners often don't think that way. The "flowchart mentality" is the result of writing a lot of programs. Flowcharting is not the way to begin writing programs—unless it is easy for you to do.

Don't worry about flowcharts. Make any kind of a plan and start writing your program. Write segments and test them, following your plan, just as you did in this book. When you are ready to use flowcharts, you will know it and start using them.

When you see a really good flowchart, perhaps in a book or magazine, you can be certain that the programmer made the chart after he got the program written and working.

PROGRAMMING STYLE

There are lots of experts on programming style. When you are a beginner, you can't understand what they are saying.

A good programming style is developed by doing most things wrong at first. You *should* do things wrong at first. That's how experts learn what to avoid and how to do it better.

You will discover the need to make programs easy to read and understand all by yourself. It will happen when you look at a program you wrote a few months ago. If you can't understand it, you will automatically start writing programs so you can understand them later, when you have forgotten the details.

If a program is easy to understand, it is probably well written and well organized. If it is well organized, it will probably run efficiently and not fail unexpectedly.

I think the programming style used in the last few chapters of this book is reasonable. You may want to begin with that style until you develop or learn one that you like better.

COMPACTING PROGRAMS

You probably noticed that the program examples in Commodore literature are very dense, with no spaces, and use control symbols often instead of control-code numbers.

I think this is a habit among Commodore "old-timers" because the early Commodore computers didn't have much memory. Commodore still has some models without much memory.

The Commodore 64 has lots of memory. It has more than enough memory to store and run most BASIC programs. There is little to be gained by compacting a program so it uses only 10000 bytes and leaves 20000 bytes unused in the computer. If a program fits in memory and runs, that's enough. Making it smaller doesn't make it fit better.

I suggest that you begin by writing programs with lots of remarks and lots of open space. When you start running out of memory, then you can start compacting your programs.

If you do something tricky in a program, put enough remarks in it to explain what you did. Use several lines of remarks if necessary. Later, if you have to work on that program, the remarks may save you hours of puzzling.

HOW TO CHECK THE AMOUNT OF AVAILABLE MEMORY

As a beginner, you are not likely to write a program that is long enough to fill up the available memory. You may dimension a huge array that fills it up, or you may make a programming error that fills it up.

Occasionally, you may want to know how much memory is free and still available to be used. If you get a memory-full error, you will become very interested in finding out what happened to the memory.

A useful function is `FRE(0)`. The statement `PRINT FRE(0)` will display the amount of memory that has not been filled by a program or data. Any number can be used in the parentheses. It is a dummy variable.

Because `FRE(0)` is treated as an integer in the computer, it has the same limits as integer numbers: -32768 to $+32767$. With a short program in memory, there are more than 32767 bytes of memory free.

To allow reporting free memory greater than 32767 bytes, a trick is used. The number 65536 is subtracted from the amount of available memory. The result is a negative number that is within the range of integer values. The negative number is displayed on the screen. The accompanying table

shows some free-memory values and the value that will be displayed by the statement `PRINT FRE(0)`.

Several values of free memory are shown above and below 32768, the transition point that causes the reported value to become negative. That is just to give you the idea.

FREE MEMORY	MINUS	WHAT IS DISPLAYED
0		0
25000		25000
32766		32766
32767		32767
32768	65536	-32768
32769	65536	-32767
32770	65536	-32766
38000	65536	-27536
65535	65536	-1

If you display free memory, and the result is a negative number, you can calculate the actual amount by this formula: $\text{FREE MEMORY} = (\text{NEGATIVE VALUE}) + 65536$

Here is a fairly sophisticated statement that will display the amount of free memory, whether it exceeds 32767 or not: `PRINT FRE(0) - ((FRE(0) < 0) * 65536)`

In the second part of that statement, the term $(\text{FRE}(0) < 0)$ is a proposition to be tested and assigned a true/false value of 0 or -1 as discussed in Chapter 18. If it is true, its value will be -1. If it is false, its value will be 0.

Suppose $\text{FRE}(0)$ is a positive number. $(\text{FRE}(0) < 0)$ will be false. Therefore, it will be assigned a value of 0. The second part of the statement will evaluate to zero because $0 * 65536 = 0$. Nothing will be subtracted from $\text{FRE}(0)$ because nothing should be subtracted.

Suppose $\text{FRE}(0)$ is a negative number. A negative number is less than 0, so the proposition $(\text{FRE}(0) < 0)$ will be true. Its value will be -1. The second part of the statement evaluates to $(-1 * 65536)$, which is (-65536) .

The statement becomes `PRINT FRE(0) - (-65536)`, which is the same as `PRINT FRE(0) + 65536`. Remembering that $\text{FRE}(0)$ was negative, that makes the adjustment to display the correct amount of free memory.

If you are working on a program that may fill up memory, you can use that statement in the program to monitor available memory and do whatever is appropriate if memory becomes nearly full.

DEBUGGING

Problems and mistakes in a program are often called *bugs*. Nearly every program is written with bugs concealed in it. Nearly every bug is simple and obvious, once it is found. When one is found, programmers say, "How could I be so stupid? Of course that won't work."

Fixing a program with a bug is usually very easy—after the bug is found. The problem is finding it.

As you have seen, I think it's a good idea to write programs in short segments and test each segment as you go along. Eventually, you will notice that you are planning the test method while you are planning or writing the segment. That's good. Writing a program that is easy to test is almost a certain guarantee of writing a good program.

Your success at debugging a program depends almost entirely on how good you are at finding the bugs. Once found, they are usually trivial problems. The most important debugging technique is testing.

WHAT YOU NEED TO KNOW

When a program doesn't work correctly, you need to know two things: Where did it go? What did it do?

Where Did It Go?—That means what lines executed and in what order? Problems due to the order of execution are often difficult to find. They may be caused by a branching routine written incorrectly. It may say to jump when $X=11$, when it should say to jump when $X>11$.

Usually, you think a program line does what you intended it to do when you wrote it. If you actually wrote it to do something else, the mistake is difficult for you to see.

There are ways to find out which lines in a program executed and in what order. One is to put in temporary program lines that stop execution. When the program stops, the computer tells you the last line number that was executed.

If a program branches and you want to verify that it branched correctly, put a temporary STOP in each branch. Run it and you will see where it went.

Another way is to put in temporary lines that print their own line numbers on the screen. For example, line 91 says PRINT 91. Scatter a few of those through a program and run it. The screen will display the path that the program took when it ran.

What Did It Do?—Sometimes, a program runs correctly part of the time but not all of the time. When that happens, the problem is usually in the value of a variable. It may take values that you didn't expect.

One way to test that is to put variables into the program and run it to see what happens. You can do that by temporary lines, such as $X=90$. Run it, and you know exactly what happens when $X=90$.

You can also establish values for variables by assignment statements in the immediate mode. Then run the program with a GOTO statement starting at whatever line number is appropriate for the test you are making.

Another way to track variables is to print them while the program runs. Put in temporary lines such as PRINT X. As the program runs, you can see the values taken by X. If it receives a value that you didn't anticipate, you have a clue to find the problem.

If you have a printer, you have two displays. If the screen display is critical, printing program variables will change it so you may not be able to see clearly what is happening on the screen. To solve that difficulty, print on the printer instead. You will have a record on paper of what the program did.

Printing the program on paper is useful when debugging. Write on it and mark it to show where jumps went and what values the variables have at various points in the program.

SUMMARY

You can't debug a program if you can't understand what it is supposed to do. Write your programs so you can understand them later, when you have forgotten the clever tricks that you used.

When a program fails for no obvious reason, you can become so frustrated while looking for the problem that you stop thinking clearly. When that happens to me, I get a cup of coffee, move my chair about six feet from the keyboard and sit in the chair.

I think about the clues that I have about the problem. Then I ask myself: Where did it go? What did it do? When I can answer those two questions, I can fix the program.

Programming is fun. Among owners of small computers, more programming is done for fun than for profit. I hope this book gives you a lot of fun—and profit if that's what you seek.

It is too optimistic to hope that your programs won't have bugs. I hope they are easy to find.
CHR\$(66)CHR\$(89)CHR\$(69)

Appendix A

TABLE 6-1
EFFECT OF ASCII CODES IN UPPER-CASE AND GRAPHICS MODE

Code	Display	Effect If Not Displayed	Code	Display	Effect If Not Displayed
0		NO EFFECT	67		
1		NO EFFECT	68		
2		NO EFFECT	69		
3		NO EFFECT	70		
4		NO EFFECT	71		
5		WHITE CHARACTERS	72		
6		NO EFFECT	73		
7		NO EFFECT	74		
8		DISABLE COMMODORE-SHIFT	75		
9		ENABLE COMMODORE-SHIFT	76		
10		NO EFFECT	77		
11		NO EFFECT	78		
12		NO EFFECT	79		
13		CARRIAGE RETURN	80		
14		SET UPPER & LOWER CASE	81		
15		NO EFFECT	82		
16		NO EFFECT	83		
17		CURSOR DOWN	84		
18		REVERSE ON	85		
19		HOME CURSOR	86		
20		BACKSPACE & DELETE	87		
21		NO EFFECT	88		
22		NO EFFECT	89		
23		NO EFFECT	90		
24		NO EFFECT	91		
25		NO EFFECT	92		
26		NO EFFECT	93		
27		NO EFFECT	94		
28		RED CHARACTERS	95		
29		CURSOR RIGHT	96		
30		GREEN CHARACTERS	97		
31		BLUE CHARACTERS	98		
32		SPACE	99		
33			100		
34	#		101		
35	\$		102		
36	%		103		
37	&		104		
38	'		105		
39	(106		
40)		107		
41	*		108		
42	+		109		
43	,		110		
44	-		111		
45	.		112		
46	/		113		
47	0		114		
48	1		115		
49	2		116		
50	3		117		
51	4		118		
52	5		119		
53	6		120		
54	7		121		
55	8		122		
56	9		123		
57	:		124		
58	;		125		
59	<		126		
60	=		127		
61	>		128		NO EFFECT
62	?		129		ORANGE CHARACTERS
63	@		130		NO EFFECT
64	A		131		NO EFFECT
65	B		132		NO EFFECT
66			133		CODE FOR FUNCTION KEY 1

TABLE 6-1 (continued)
EFFECT OF ASCII CODES IN UPPER-CASE AND GRAPHICS MODE

Code	Display	Effect If Not Displayed	Code	Display	Effect If Not Displayed
134		CODE FOR FUNCTION KEY 3	165		
135		CODE FOR FUNCTION KEY 5	166	*	
136		CODE FOR FUNCTION KEY 7	167		
137		CODE FOR FUNCTION KEY 2	168	*	
138		CODE FOR FUNCTION KEY 4	169	▼	
139		CODE FOR FUNCTION KEY 6	170		
140		CODE FOR FUNCTION KEY 8	171		
141		CARRIAGE RETURN ON SCREEN	172	*	
142		SET UPPER CASE & GRAPHICS	173	*	
143		NO EFFECT	174		
144		BLACK CHARACTERS	175		
145		CURSOR UP	176		
146		REVERSE OFF	177		
147		CLEAR SCREEN, HOME CURSOR	178		
148		INSERT	179		
149		BROWN CHARACTERS	180		
150		LIGHT RED CHARACTERS	181		
151		GRAY 1 CHARACTERS	182		
152		GRAY 2 CHARACTERS	183		
153		LIGHT GREEN CHARACTERS	184		
154		LIGHT BLUE CHARACTERS	185		
155		GRAY 3 CHARACTERS	186		
156		PURPLE CHARACTERS	187		
157		CURSOR LEFT	188		
158		YELLOW CHARACTERS	189		
159		CYAN CHARACTERS	190		
160		SPACE	191		
161					
162	■		192 TO 223	SAME AS 96 TO 127	
163	■		224 TO 254	SAME AS 160 TO 190	
164	■		255	SAME AS 126	

TABLE 6-2
EFFECT OF ASCII CODES IN UPPER-AND-LOWER-CASE MODE

Code	Display	Effect If Not Displayed	Code	Display	Effect If Not Displayed
0		no effect	32		space
1		no effect	33		
2		no effect	34		
3		no effect	35	*	
4		no effect	36	*	
5		white characters	37	%	
6		no effect	38	&	
7		no effect	39	'	
8		disable commodore-shift	40	<	
9		enable commodore-shift	41	>	
10		no effect	42	*	
11		no effect	43	+	
12		no effect	44		
13		carriage return	45	-	
14		set upper & lower case	46	.	
15		no effect	47	/	
16		no effect	48	0	
17		cursor down	49	1	
18		reverse on	50	2	
19		home cursor	51	3	
20		backspace & delete	52	4	
21		no effect	53	5	
22		no effect	54	6	
23		no effect	55	7	
24		no effect	56	8	
25		no effect	57	9	
26		no effect	58		
27		no effect	59	:	
28		red characters	60	@	
29		cursor right	61	#	
30		green characters	62	\$	
31		blue characters	63	%	

TABLE 6-2 (continued)
EFFECT OF ASCII CODES IN UPPER-AND-LOWER-CASE MODE

Code	Display	Effect If Not Displayed	Code	Display	Effect If Not Displayed
64	@		131		no effect
65	a		132		no effect
66	b		133		code for function key 1
67	c		134		code for function key 3
68	d		135		code for function key 5
69	e		136		code for function key 7
70	f		137		code for function key 2
71	g		138		code for function key 4
72	h		139		code for function key 6
73	i		140		code for function key 8
74	j		141		carriage return on screen
75	k		142		set upper case & graphics
76	l		143		no effect
77	m		144		black characters
78	n		145		cursor up
79	o		146		reverse off
80	p		147		clear screen, home cursor
81	q		148		insert
82	r		149		brown characters
83	s		150		light red characters
84	t		151		gray 1 characters
85	u		152		gray 2 characters
86	v		153		light green characters
87	w		154		light blue characters
88	x		155		gray 3 characters
89	y		156		Purple characters
90	z		157		cursor left
91	[158		yellow characters
92	\		159		cyan characters
93]		160		space
94	↑		161	■	
95	←		162	■	
96	-		163	■	
97	A		164	■	
98	B		165	■	
99	C		166	■	
100	D		167	■	
101	E		168	■	
102	F		169	■	
103	G		170	■	
104	H		171	■	
105	I		172	■	
106	J		173	■	
107	K		174	■	
108	L		175	■	
109	M		176	■	
110	N		177	■	
111	O		178	■	
112	P		179	■	
113	Q		180	■	
114	R		181	■	
115	S		182	■	
116	T		183	■	
117	U		184	■	
118	V		185	■	
119	W		186	■	
120	X		187	■	
121	Y		188	■	
122	Z		189	■	
123	+		190	■	
124	×		191	■	
125					
126	×				
127	×		192 to 223		same as 96 to 127
128		no effect	224 to 254		same as 160 to 190
129		orange characters	255		same as 126
130		no effect			

TABLE 7-2
SCREEN-DISPLAY CODES USING UPPER-CASE-AND-GRAPHICS MODE
 (A reversed space is printed in front of reversed characters.)

Code	Display	Code	Display	Code	Display
0	@	57	9	114	+
1	A	58		115	+
2	B	59	.	116	+
3	C	60	<	117	+
4	D	61	=	118	+
5	E	62	>	119	+
6	F	63	o	120	+
7	G	64	1	121	+
8	H	65	→	122	+
9	I	66	—	123	+
10	J	67		124	+
11	K	68		125	+
12	L	69		126	+
13	M	70		127	+
14	N	71		128	+
15	O	72		129	+
16	P	73	~	130	+
17	Q	74	o	131	+
18	R	75	o	132	+
19	S	76	∟	133	+
20	T	77	/	134	+
21	U	78	∟	135	+
22	V	79	∟	136	+
23	W	80	∟	137	+
24	X	81	•	138	+
25	Y	82	•	139	+
26	Z	83	•	140	+
27	[84	•	141	+
28	£	85	∠	142	+
29]	86	X	143	+
30	↑	87	O	144	+
31	←	88	+	145	+
32		89	—	146	+
33	!	90	•	147	+
34	"	91	+	148	+
35	#	92	—	149	+
36	\$	93	—	150	+
37	%	94	▲	151	+
38	&	95	▲	152	+
39	'	96	■	153	+
40	<	97	■	154	+
41	>	98	■	155	+
42	*	99	■	156	+
43	+	100	■	157	+
44	,	101	■	158	+
45	.	102	■	159	+
46	/	103	■	160	+
47	0	104	■	161	+
48	1	105	■	162	+
49	2	106	■	163	+
50	3	107	■	164	+
51	4	108	■	165	+
52	5	109	■	166	+
53	6	110	■	167	+
54	7	111	■	168	+
55	8	112	■	169	+
56		113	■	170	+

TABLE 7-2 (continued)

Code	Display	Code	Display	Code	Display
171	+	200	■	229	■
172	■	201	■	230	■
173	■	202	■	231	■
174	■	203	■	232	■
175	■	204	■	233	■
176	■	205	■	234	■
177	■	206	■	235	■
178	■	207	■	236	■
179	■	208	■	237	■
180	■	209	■	238	■
181	■	210	■	239	■
182	■	211	■	240	■
183	■	212	■	241	■
184	■	213	■	242	■
185	■	214	■	243	■
186	■	215	■	244	■
187	■	216	■	245	■
188	■	217	■	246	■
189	■	218	■	247	■
190	■	219	■	248	■
191	■	220	■	249	■
192	■	221	■	250	■
193	■	222	■	251	■
194	■	223	■	252	■
195	■	224	■	253	■
196	■	225	■	254	■
197	■	226	■	255	■
198	■	227	■		
199	■	228	■		

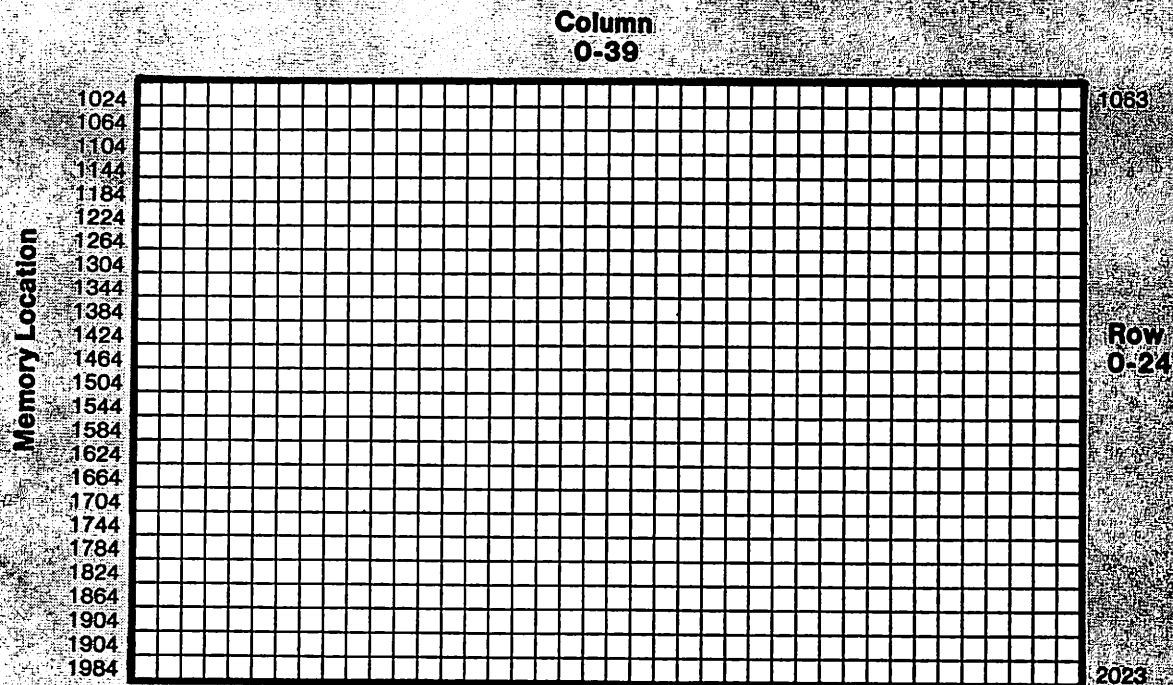
TABLE 7-3
SCREEN-DISPLAY CODES USING UPPER-CASE-AND-LOWER-CASE MODE
(A reversed space is printed in front of reversed characters.)

Code	Display	Code	Display	Code	Display
0	@	27	[54	6
1	a	28	f	55	7
2	b	29]	56	8
3	c	30	↑	57	9
4	d	31	↑	58	
5	e	32		59	
6	f	33		60	<
7	g	34	=	61	=
8	h	35	#	62	>
9	i	36	\$	63	>
10	j	37	%	64	
11	k	38	&	65	B
12	l	39	'	66	B
13	m	40	(67	C
14	n	41)	68	D
15	o	42	*	69	E
16	p	43	+	70	F
17	q	44	,	71	G
18	r	45	-	72	H
19	s	46	.	73	I
20	t	47	/	74	J
21	u	48	0	75	K
22	v	49	1	76	L
23	w	50	2	77	M
24	x	51	3	78	N
25	y	52	4	79	O
26	z	53	5	80	P

TABLE 7-3 (continued)

Code	Display	Code	Display	Code	Display
81	0	141	0	201	0
82	1	142	1	202	1
83	2	143	2	203	2
84	3	144	3	204	3
85	4	145	4	205	4
86	5	146	5	206	5
87	6	147	6	207	6
88	7	148	7	208	7
89	8	149	8	209	8
90	9	150	9	210	9
91	+	151	+	211	+
92	-	152	-	212	-
93	=	153	=	213	=
94	*	154	*	214	*
95	/	155	/	215	/
96	%	156	%	216	%
97	^	157	^	217	^
98	~	158	~	218	~
99	!	159	!	219	!
100	@	160	@	220	@
101	#	161	#	221	#
102	\$	162	\$	222	\$
103	%	163	%	223	%
104	&	164	&	224	&
105	'	165	'	225	'
106	(166	(226	(
107)	167)	227)
108	*	168	*	228	*
109	+	169	+	229	+
110	,	170	,	230	,
111	.	171	.	231	.
112	/	172	/	232	/
113	:	173	:	233	:
114	;	174	;	234	;
115	'	175	'	235	'
116	[176	[236	[
117]	177]	237]
118	^	178	^	238	^
119	_	179	_	239	_
120	`	180	`	240	`
121	~	181	~	241	~
122	!	182	!	242	!
123	@	183	@	243	@
124	#	184	#	244	#
125	\$	185	\$	245	\$
126	%	186	%	246	%
127	&	187	&	247	&
128	'	188	'	248	'
129	(189	(249	(
130)	190)	250)
131	*	191	*	251	*
132	+	192	+	252	+
133	,	193	,	253	,
134	.	194	.	254	.
135	/	195	/	255	/
136	:	196	:		
137	;	197	;		
138	'	198	'		
139	[199	[
140]	200]		

FIGURE 7-4
SCREEN-LOCATION NUMBERS & SCREEN MEMORY MAP

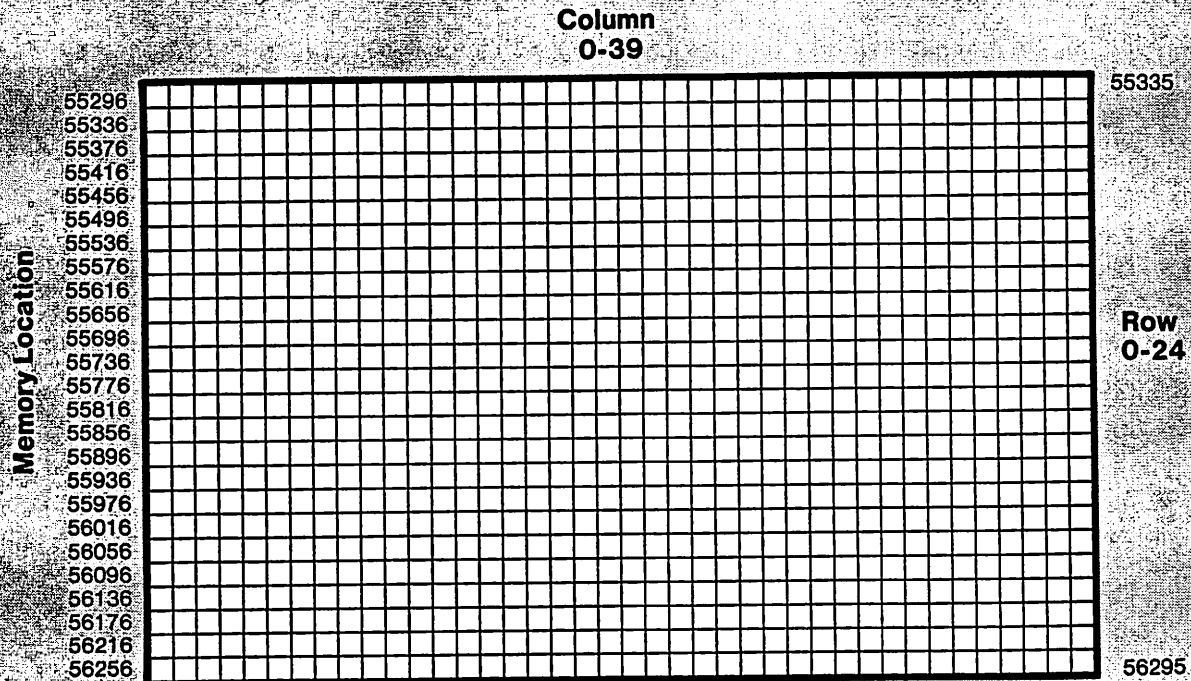


To place a character at any location on the screen, the computer puts a screen-display code in the corresponding location of this screen-memory map in the computer memory. Location 1024 on this map in memory corresponds to location 0 on the screen—the top left corner.

There are 1000 print locations on the screen and 1000 locations in the screen memory map.

Characters can be displayed by poking screen-display codes into the appropriate locations in this screen memory map. You must also poke character color into the color memory map, Figure 7-5.

**FIGURE 7-5
COLOR MEMORY MAP**



After a screen-display code is placed in the screen memory map, the character color for that location on the screen is placed in this color memory map. Location 55296 on this map in memory corresponds to location 0 on the screen—the top left corner.

There are 1000 print locations on the screen and 1000 locations in the screen memory map.

If there is no character on the screen at a certain location, and you display a character by poking it into the screen memory map, you must also poke the character color into the corresponding location of this color memory map.

If a character is already being displayed, you can change the character by poking a different screen-display code into the screen memory map. If you do not poke a new color into the color memory map, the new character will be displayed using the same color as the character that it replaced.

Appendix B:

BASIC Words

And

Definitions

This appendix gives definitions of BASIC words. Some are simplified to agree with the scope of this book. For more precise definitions, refer to the *Commodore User's Guide*, packaged with the Commodore 64.

Where a BASIC word requires a value in parentheses, an empty set of parentheses is shown with the word.

ABS()

Takes the positive value of the expression in parentheses, even if it is negative. Mathematically, absolute value means a number without consideration of its sign. In the computer, numbers must be positive or negative.

$ABS(5.5) = 5.5$

$ABS(-5.5) = 5.5$

AND

Used in IF-THEN statements to connect relational operations for a more complex test. The statement `IF A=B AND X<>Y GOTO 100` will execute only if $A=B$ and $X \neq Y$.

Used to control individual bits in a byte by ANDing two values. When two bits are ANDed, the result is 1 only if one bit *and* the other bit are both 1. In binary, $111 \text{ AND } 101 = 101$.

ASC()

Takes the ASCII code of the first character of a string expression in the parentheses. `ASC(A$)` will be 65 if $A\$ = A$ because 65 is the ASCII code for A. See Appendix A.

`ASC(X$)` produces an error if $X\$$ is a null. If it is possible for $X\$$ to be a null string, prevent the error by using `ASC(X$+CHR$(0))`. This will always have the same value as `ASC(X$)` and will become 0 when $X\$$ is a null.

ATN()

Is the angle whose tangent is in the parentheses. The angle is in radians. `ATN(1)` is the angle in radians whose tangent is 1.

CHR\$()

The way to include ASCII code numbers in statements. The ASCII code number is enclosed in the parentheses. `CHR$(32)` is recognized by the BASIC interpreter as the character or action indicated by ASCII code 32.

Appendix B: BASIC Words And Definitions

CLOSE

Closes communication with files and devices. Causes characters held in buffer to be placed in file. Causes disk directory and block-allocation records to be updated on disk. Must use file number that was used in earlier OPEN statement. CLOSE 7 closes file or device that was opened as file number 7

CLR

Clears everything from memory except a BASIC program. Clears all variables, FOR-NEXT loop specifications, GOSUB addresses, arrays and user-defined functions.

CMD

Substitutes another device or open file for the display. Device being substituted must have been opened using a file number. CMD statement uses same file number. CMD 4 substitutes device or file opened as file number 4 for the display. PRINT and LIST statements will then send data to the substituted device or file.

CONT

Used to restart program halted by STOP, END or pressing RUN STOP key. While program is halted, statements can be executed in the immediate mode. When CONT is entered in the immediate mode, program will restart, if possible, at next statement to be executed when halted. Program cannot restart and CAN'T CONTINUE error message will be displayed if a program line was changed, or there was a program error.

COS()

The value of COS(a) is the cosine of the angle, a , enclosed in parentheses. The angle is in radians. COS(1) is the cosine of an angle of 1 radian.

DATA

Used on a program line to signal the BASIC interpreter that the line holds data items to be read by a READ statement. The READ statement must supply variable names for each data item as it is read. DATA statements are read in sequence, starting with the lowest DATA line number, from left to right in each DATA line. Data items are separated by commas.

Type designation of variable names supplied by READ statement must match data item being read. Some data items must be enclosed in quotation marks to be read. These are commas, colons, spaces and shifted keystrokes.

DEF FN

Used to define a user-defined function. Format is DEF FN *function name* (*variable name*)=*formula*

The *function name* is used thereafter to call the function. The *variable name* used in the DEF FN statement is given a value in parentheses when the function is called. The *formula* is stored by the computer and used to calculate the value of the function.

For example, DEF FN Z (X)=3*X defines the function FN Z as a function of X. PRINT FN Z (4) uses the value 4 for X in the formula provided by the DEF FN Z statement. PRINT FN Z (4) will display the value 12 because 3*X will be 12 when X is 4.

DIM()

Used to dimension an array that will have a maximum subscript greater than 10. The format is DIM *array name* (r,c,d)

The *array name* is a variable name whose type-designation controls the variable type that can be placed in the array. Up to three dimensions are indicated by (r,c,d). The dimension r is the number of rows, c is the number of columns, d is the depth or number of similar two-dimensional data structures having the same r and c dimensions—used to form a three-dimensional array.

The numbers in a DIM statement are integers. The smallest value is 0 and the largest value is 32767. If c and d are omitted, their values are 10.

DIM A\$(3) dimensions a one-dimensional string array with rows 0-3.

DIM B(5,3) dimensions a two-dimensional numeric array with rows 0-5, and columns 0-3.

DIM C%(7,15,3) dimensions a three-dimensional integer array with rows 0-7, columns 0-15, and a depth of 0-3. This provides up to four similar two-dimensional tables, one "behind" the other. The variable **C%(5,6,1)** is the value in the fifth row of the sixth column of table 1 in this array. The variable **C%(5,6,2)** is the value in the fifth row of the sixth column of the two-dimensional table 2 in this array.

END

Stops program execution and returns to the immediate mode.

EXP()

Has the value of the mathematical constant *e* raised to the value in the parentheses. **PRINT EXP(2)** will display *e* raised to the second power.

FOR

Part of FOR-NEXT statement such as **FOR I=1 TO 50**. Names loop counter and sets number of loop operations. See **NEXT** and **STEP**.

FRE()

Has value of unused memory available for BASIC program and data. Used with dummy number in parentheses. **PRINT FRE(0)** will display free memory. If number displayed is negative, adding 65536 to negative number yields free memory. Refer to pages 306 and 307.

GET

Accepts single character from keyboard buffer, even if character is a null. Supplies variable name for character. **GET A\$** accepts one character and names it **A\$**. Allows typing and entering single characters without pressing **RETURN**. Character received is not displayed. **GET** is often used in a loop to stop program execution until a key is pressed.

GET#

Accepts single character from open file or device using file number. Supplies variable name for character. Format is **GET# n,A\$** in which *n* is file number used to open file or device. Accepts any ASCII code number including punctuation and **CHR\$(13)**. To get more than one character, use **GET#** in a loop. The character received is not displayed.

GET# can be used to read characters in sequence from the screen display by opening screen as device number 3.

GOSUB

Jumps to specified line number and executes program from that line number until a **RETURN** statement is executed. **RETURN** statement jumps back to statement immediately following **GOSUB**. Used to call subroutines. **GOSUB 1000** sends program to subroutine at line 1000. Target line number must exist, but it can be a remark.

GOTO

Jumps to specified line number. Program then executes from the specified line number. Target line number must exist, but it can be a remark.

IF

Part of IF-THEN statement. States condition that will be tested for true/false, such as **IF X=Y**. If true, the following **THEN** statement will be executed. If condition is false, program jumps immediately to next numbered line number. Nothing behind the **IF** condition will be executed. Format is **IF (condition) THEN (action)**

Appendix B: BASIC Words And Definitions

Example: IF X=Y THEN Z=4 If action is GOTO a line number, the IF-THEN statement can be written three ways:

```
IF X=Y THEN GOTO 100
IF X=Y THEN 100
IF X=Y GOTO 100
```

INPUT

Prepares computer to receive data typed at keyboard. Accepts up to two screen rows of characters (80 characters).

Supplies variable name for data. Receives data until a comma, semicolon, colon or carriage return is entered. Will accept those characters as CHR\$() or if enclosed in quotation marks. May set up a list of variable names to receive data.

INPUT A\$ receives string data and places it in memory as A\$.

INPUT A\$,B receives two data items: A\$ and B. A prompt may be included in an input statement as instructions to the keyboard operator.

INPUT "ENTER YOUR NAME";N\$ displays the prompt shown inside the quotation marks, then accepts N\$ from the keyboard. A semicolon must be used after the closing quotation mark.

INPUT#

Prepares computer to receive data from open file on disk or tape. Accepts up to 80 characters. Supplies variable name or list of variable names for data. Receives data into variable name until a comma, semicolon, colon or carriage return is encountered. Will accept those characters without ending input if written as CHR\$() or if enclosed in quotation marks.

Uses file number of opened file. INPUT# 4, A\$ receives string data and places it in memory as A\$. INPUT# 5, A\$,B receives two data items—A\$ and B.

If screen is opened as a file, may be used to read and display one logical line at a time. INPUT# 3, L\$ gets one logical line from screen.

INT()

Is the integer value of number in parentheses. Value of INT(5.5) is 5. Value of INT(-5.5) is -6.

LEFT\$

Takes left part of a string. Format is LEFT\$(a\$,n) in which a\$ is the name of the string and n is the number of characters to be taken. If n is greater than length of string, entire string is taken. If n is zero, a null string is taken.

LEN()

Is length of string name enclosed in parentheses. LEN(A\$) is length of A\$.

LET

Optional for assignment statements. LET X=3 means same as X=3.

LIST

Used in immediate mode to list a program in memory. Can be used in program line, but program will return to immediate mode after executing LIST command. If CMD is used to substitute another open file or device for screen, program lines are sent to file or device. Options for LIST command are:

LIST lists entire program.

LIST -500 lists lines from beginning through 500.

LIST 500- list lines from 500 to end of program.

LIST 500-1000 lists lines from 500 through 1000.

LIST 500 lists line 500.

LOAD

Loads program from disk or tape file. Format is `LOAD "filename", device number`. If device number omitted, loads from tape. If filename omitted, loads first program on tape.

Can use "wild cards" when loading from disk. An asterisk substitutes for any filename. `LOAD "*", 8` loads first program on disk. A question mark substitutes for any character in filename. `LOAD "A????", 8` loads the first program on disk beginning with A, followed by any three characters.

LOG()

Is the natural logarithm of a number enclosed in the parentheses. $\text{LOG}(n)$ is the log to base e of the number n . To find a logarithm of a number to the base 10, divide $\text{LOG}(n)$ by $\text{LOG}(10)$.

MID\$()

Takes a segment a string. Format is `MID$(a$, f, n)` in which $a\$$ is name of string, f is number of first character to be taken, n is number of characters to be taken.

Both f and n may have values from 0 to 255. If f is larger than the length of the string, no characters are taken. If n is zero, no characters are taken. If n is omitted or extends beyond the length of the string, all characters from f to the end of the string are taken.

NEW

Clears memory. If used on a program line, clears memory and erases preceding program lines.

NEXT

Used at bottom of loop in FOR-NEXT loops. Using name of loop counter, such as `NEXT I`, is optional. Increments loop counter by STEP value and tests value of counter against maximum value specified in FOR statement. If loop counter does not exceed maximum value, the loop runs again.

NOT

Used as a logical operator to reverse the sense of a bit. In binary, $X = \text{NOT } 1$ means $X = 0$.

ON

Used in `ON n GOTO` or `ON n GOSUB` statements. Example: `ON N GOTO 100,200,300`. Range of N must be 1,2,3. If $N = 1$ program jumps to first listed target line, and so forth.

OPEN

Used to open devices or files. Format varies according to use. See relevant chapters.

OR

Used in IF-THEN statements to connect relational operations for a more complex test. The statement `IF A=B OR X<>Y GOTO 100` will operate if either $A=B$ or $X<>Y$.

Used to control individual bits in a byte by ORing two values. When two bits are ORed, the result is 1 if one bit or the other bit is 1. In binary, $111 \text{ OR } 101 = 111$.

PEEK()

Has value stored in memory location specified in the parentheses. `PEEK(3030)` has the value stored at location 3030. The value stored in a memory location is in the range of 0-255. The number used to specify the memory location must be in the range of 0-65535.

POKE

Places a number in the range of 0-255 into a memory location. Format is `POKE l,n` in which n is the value to be poked and l the memory location in the range of 0-65535. Not all memory locations in this range are RAM.

Appendix B: BASIC Words And Definitions

POS()

Value is the position of the cursor in an 80-character logical line. Leftmost character is position zero. A logical line occupies two screen rows. Format is POS(0) in which number in parentheses, can have any value. It is a dummy variable.

PRINT

Displays data on screen. A single data item may be displayed or a list of data items. PRINT X displays the value of X.

PRINT A\$ B\$ X displays three items, one following the other. Normally, a carriage return is executed automatically at the end of a PRINT statement. Commas between the items in a PRINT list cause automatic tabs to the next print zone on the screen. A semicolon at the end of a PRINT statement suppresses the carriage return and holds the cursor on that screen row.

Control symbols may be enclosed in quotation marks in a PRINT statement. If so, they will be executed at part of the statement.

PRINT#

Writes data to a file on disk or tape using file number that was used to open the file. Format is PRINT# *file n, variable name*

In a sequential file, the simplest way to separate data items is to write each item separately, using a PRINT# statement for each item.

READ

Reads data items from a data list in a DATA statement. Must supply a variable name for each data item read. READ A\$ reads a string from the data list and puts it in memory with the name A\$. If there are several items in a data list, using a loop to read them into an array is good practice.

REM

Used to precede remarks on program lines. The BASIC interpreter stops executing a program line when it encounters a REM statement.

RESTORE

Used at the end of a DATA statement to reset the pointer to the beginning of the first DATA statement in the program. Allows READING the DATA statement more than once.

RETURN

Used as last statement in a subroutine. Returns to statement following the statement that called the subroutine.

RIGHT\$

Takes characters from right end of a string. Format is RIGHT\$(*a\$,n*) in which *a\$* is the name of the string, *n* is the number of characters to be taken, counting from the rightmost character.

RND()

Has the value of the next number in a series of random numbers ranging from 0 to 1. Number in parentheses is the *seed*. If seed is positive, a repeatable pseudo-random series of numbers is produced. It will repeat when started again with the same seed.

If seed is negative, random number sequence is reseeded to new starting value. If seed is zero, random numbers are derived from system clock.

RUN

Used in the immediate mode to run a program. Clears memory before starting to run the program. RUN starts running from the beginning of the program. RUN followed by a line number, such as RUN 500, begins running at the specified line number. If the specified line number does not exist, an error results.

SAVE

Used in the immediate mode to save a program to disk or tape. Format is

SAVE *filename, device number*

The filename may be up to 16 characters in quotation marks and must be a string variable name. If device number is omitted, program is saved to tape. If device number and filename are omitted, program is saved to tape with no filename. If program is first program on tape, may be loaded without specifying filename. If device number is 8, program is saved to disk.

SGN()

Value is determined by sign of numeric expression, n , in the parentheses. If numeric is positive, the value of **SGN**(n) is 1. If numeric is zero, value of **SGN**(n) is zero. If numeric is negative, value of **SGN**(n) is -1 .

SIN

The value of **SIN**(a) is the sine of the angle a enclosed in parentheses. The angle is in radians. **SIN**(1) is the sine of an angle of 1 radian.

SPC()

On screen, **SPC**(n) moves the cursor n columns to the right of cursor position when **SPC**(n) is executed. On printer, moves print head n columns to right.

SQR()

The value of **SQR**(n) is the square root of n . If n is negative, an error results.

STATUS

The computer maintains a numeric variable **STATUS**, or **ST**, in memory to monitor input-output operations to files. For sequential disk files, **ST**=64 indicates end of file. For disk files or printer, **ST**=7 indicates device not connected or not turned on. For tape file, **ST**=0 is normal. **ST**=64 indicates end of file. **ST**= -128 indicates end-of-tape code read from tape.

STEP

Used with **FOR-NEXT** loop to specify amount that loop counter is incremented by **NEXT** statement. Example: **FOR** $I=1$ **TO** 10 **STEP** 2 causes counter to be incremented by 2 at **NEXT** statement. **FOR** $I=100$ **TO** 30 **STEP** -5 causes counter to count backward from 100 to 30 in steps of 5.

STOP

Used on program line to halt execution of a program. Computer returns to immediate mode. Memory is not erased. Statements can be executed in the immediate mode. Program can be restarted by **RUN**, **CONT** or **GOTO** statement.

STR\$()

Converts a numeric expression in the parentheses to a string expression. Will read expression in parentheses to first non-numeric character. The statement **X\$=STR\$(123ABC)** sets **X\$** equal to the string "123"

SYS

Used to call machine-language routines from a BASIC program.

TAB()

The statement **TAB**(n) positions the cursor at column n on the screen, counting the leftmost column as zero. On a printer, **TAB**() has the same effect as **SPC**().

TAN()

The value of **TAN**(a) is the tangent of the angle, a , enclosed in parentheses. The angle is in radians. **TAN**(1) is the tangent of an angle of 1 radian.

Appendix B: BASIC Words And Definitions

THEN

Used in IF-THEN statements to provide the THEN expression. Example:

IF I=40 THEN GOTO 500

TIME

TI is a decimal-number variable maintained in memory by the computer. Its value is the time since the computer was turned on, in 1/60ths of a second. The timer is turned off during tape input-output operations. Resumes counting as last value before tape input-output.

TI\$

TI\$ is a six-digit string variable derived from TI. The first two digits are hours on a 24-hour clock. The next two digits are minutes. The rightmost two digits are seconds. The timer is disabled during tape input-output. Resumes counting at last value of TI\$ before tape input-output operation. TI\$ can be set to actual time by an assignment statement.

USR

Used to call machine-language routines.

VAL()

The value of VAL (s\$) is the numeric value of the string enclosed in the parentheses. Evaluates string from first character to first non-numeric character. Accepts decimal points and exponent E in floating-point notation.

VERIFY

Compares program on tape or disk to program in memory. Format is

VERIFY *"filename", device number*

If device number is omitted, next program on tape is verified.

WAIT

Used to halt program execution until a specified location in memory has a specified bit pattern.

Appendix C:

Abbreviations For BASIC Words

The Commodore 64 allows you to type either complete BASIC words or abbreviations. The abbreviations are a shortcut method of typing and entering statements. When a BASIC word is entered as an abbreviation, it appears in a program listing as the complete word.

Using these abbreviations does not save space in memory. BASIC words are stored in memory by symbols that are the same, no matter how the word was entered.

Each of these abbreviations is made by one or two keystrokes, followed by one keystroke made with a SHIFT key held down. The shifted keystroke is shown in this table as SHIFT-*L*, in which *L* is the letter to be typed.

Exceptions are PRINT, STATUS, TIME and TIME\$. Abbreviations for those words are shown in the table.

Because shifted keystrokes produce graphic symbols on the screen, what you see will be the unshifted letter or letters that you type, followed by the graphics symbol produced by the shifted keystroke. In a program listing, the basic word appears—not the combination of letters and a graphic symbol.

Where no abbreviation is shown on the table, there is no abbreviation. Type the complete BASIC word

BASIC WORD ABBREVIATION

ABS	A SHIFT-B
AND	A SHIFT-N
ASC	A SHIFT-S
ATN	A SHIFT-T
CHR\$	C SHIFT-H
CLOSE	CL SHIFT-O
CLR	C SHIFT-L
CMD	C SHIFT-M
CONT	C SHIFT-O
COS	
DATA	D SHIFT-A
DEF	D SHIFT-E
DIM	D SHIFT-I
END	E SHIFT-N
EXP	E SHIFT-X
FN	
FOR	F SHIFT-O
FRE	F SHIFT-R
GET	G SHIFT-E
GET#	
GOSUB	GO SHIFT-S

BASIC WORD ABBREVIATION

GOTO	G SHIFT-O
IF	
INPUT	
INPUT#	I SHIFT-N
INT	
LEFT\$	LE SHIFT-F
LEN	
LET	L SHIFT-E
LIST	L SHIFT-I
LOAD	L SHIFT-O
LOG	
MID\$	M SHIFT-I
NEW	
NEXT	N SHIFT-E
NOT	N SHIFT-O
ON	
OPEN	O SHIFT-P
OR	
PEEK	P SHIFT-E
POKE	P SHIFT-O
POS	
PRINT	?
PRINT#	P SHIFT-R
READ	R SHIFT-E
REM	
RESTORE	RE SHIFT-S
RETURN	RE SHIFT-T
RIGHT\$	R SHIFT-I
RND	R SHIFT-N
RUN	R SHIFT-U
SAVE	S SHIFT-A
SGN	S SHIFT-G
SIN	S SHIFT-I
SPC(S SHIFT-P
SQR	S SHIFT-Q
STATUS	ST
STEP	ST SHIFT-E
STOP	S SHIFT-T
STR\$	ST SHIFT-R
SYS	S SHIFT-Y
TAB(T SHIFT-A
TAN	
THEN	T SHIFT-H
TIME	TI
TIMES	TI\$
USR	U SHIFT-S
VAL	V SHIFT-A
VERIFY	V SHIFT-E
WAIT	W SHIFT-A

Appendix D:

DOS Error Messages

These error messages are reported through command channel 15 of the disk drive. In some cases, the probable cause has been simplified to conform to the scope of this book. Where the same error message has more than one message number, there is more than one possible reason for the error. These reasons are listed in the instruction booklet packaged with the disk drive. Usually, understanding the reason requires advanced technical knowledge of the Disk Operating System.

ERROR MESSAGE

20-24 READ ERROR

25 WRITE ERROR

26 WRITE PROTECT ON

27 READ ERROR

28 WRITE ERROR

29 DISK ID MISMATCH

30-35 SYNTAX ERROR

39 SYNTAX ERROR

50 RECORD NOT PRESENT

51 OVERFLOW IN RECORD

52 FILE TOO LARGE

60 WRITE FILE OPEN

61 FILE NOT OPEN

62 FILE NOT FOUND

63 FILE EXISTS

64 FILE TYPE MISMATCH

65 NO BLOCK

PROBABLE CAUSE

Problem in reading record on disk.

Problem in writing to disk.

Program attempted to write to disk with write-protect notch covered.

Problem in reading record on disk.

Problem in writing to disk.

Incorrect disk may be in drive.

Command received through the command channel that cannot be understood by the Disk Operating System. Possible error in spelling or punctuation.

Program attempted to read a relative record that is more than 1 record number past end of file.

Data item sent to relative file exceeds allowable record length in file. Change length of data item or change OPEN statement to increase record length.

File exceeds available space on disk.

Program opened same file twice without closing it in between.

Program attempted to read or write to a file that is closed.

File not found on disk. Possibly filename misspelled or incorrect punctuation, or wrong disk in drive.

Program attempted to open file in write mode that already exists.

Statement uses file-type designation that does not match actual file type.

Technical error in addressing disk.

Appendix D: DOS Error Messages

ERROR MESSAGE	PROBABLE CAUSE
66 ILLEGAL TRACK AND SECTOR	Technical error in addressing disk.
67 ILLEGAL SYSTEM T OR S	Technical error in addressing disk.
70 NO CHANNEL AVAILABLE	Maximum number of files already open.
71 DIRECTORY ERROR	Technical error in disk directory.
72 DISK FULL	Disk full. Not all data being sent to disk may actually be recorded on disk. Data will still be in memory.
73 DOS MISMATCH	DOS system is different than DOS used to make disk in disk drive.
74 DRIVE NOT READY	No disk in drive or latch not closed.

Appendix E:

BASIC Error Messages

This table summarizes error messages produced by the BASIC interpreter. BASIC error messages are preceded by a question mark, as an identifier, and followed by the word ERROR. Appendix D shows DOS error messages produced by the disk operating system.

ERROR MESSAGE	PROBABLE CAUSE
BAD DATA	String data was input to a numeric variable name.
BAD SUBSCRIPT	An array-element name was used with a subscript that is out of the range established by the DIM statement or out of the default range of 0-10 if the array is not dimensioned.
CAN'T CONTINUE	Response to a CONT command when the BASIC interpreter cannot continue running the program. Caused by program changes or an error.
DEVICE NOT PRESENT	Program attempts to open, close, send or receive data from a device, such as a disk drive, that is not available. Possibly device is not connected, or not turned on.
DIVISION BY ZERO	Program attempted to divide by zero. If divisor is a variable, it evaluates to zero.
EXTRA IGNORED	Too many items were input to an INPUT statement. Often caused by comma in data.
FILE NOT FOUND	Disk or tape file not found. Filename may be typed incorrectly or file may not exist.
FILE NOT OPEN	A file was addressed that is not open. CLOSE, CMD, INPUT#, OUTPUT# and GET# statements must address an open file.
FILE OPEN	An OPEN statement attempted to open a file that was already open.
FORMULA TOO COMPLEX	Too many nested parentheses or a statement too complex. Simplify by breaking it into two statements.
ILLEGAL DIRECT	The BASIC interpreter received a statement in the immediate or direct mode that can be used only in a program. INPUT is an example.

Appendix E: BASIC Error Messages

ERROR MESSAGE	PROBABLE CAUSE
ILLEGAL QUANTITY	A number or the value of a numeric variable in a statement is outside of the allowable range for that variable type. For integers, range is -32768 to +32767. For decimal numbers, maximum value is 1.70141183E+38. Exceeding these ranges will produce an error message. Decimal numbers smaller than 2.93873588E-39 are handled as zero without producing an error message.
LOAD	Problem in loading a program from tape.
NEXT WITHOUT FOR	The BASIC interpreter found a NEXT statement without a matching FOR statement. May be due to incorrect variable name in NEXT statement.
NOT INPUT FILE	Program attempted to input data from a file that was opened for output to the file.
NOT OUTPUT FILE	Program attempted to send data to a file that was opened for input to the computer.
OUT OF DATA	Program executed a READ statement with no DATA item left to be read.
OUT OF MEMORY	Memory may be full. Also caused by too many nested FOR-NEXT loops, too many nested GOSUBS or too many FOR-NEXT loops left without running to normal termination.
OVERFLOW	The numeric result of a calculation is out of range for that variable type. See ILLEGAL QUANTITY error message.
REDIM'D ARRAY	The program executed a dimension statement twice for the same array. May be caused by looping back. May be caused by using an array element name before executing a dimension statement for that array. Using the array name causes automatic dimension to maximum subscript of 10. A following dimension statement produces this error message.
REDO FROM START	Letters or non-numeric symbols were input to a numeric variable name in an INPUT statement. Program will accept another complete input without stopping.
RETURN WITHOUT GOSUB	Program executed a RETURN statement without prior GOSUB.
STRING TOO LONG	A string longer than 255 characters was created by concatenating.
SYNTAX ERROR	A statement is not understood by the BASIC interpreter. May be misspelled or have incorrect punctuation.
TYPE MISMATCH	Variables with different type designations were set equal to each other, such as A\$=B.
UNDEF'D FUNCTION	BASIC encounters a statement that appears to call a user-defined function that has not been defined by an earlier statement.
UNDEF'D STATEMENT	A statement jumped to a line number that does not exist in the program.
VERIFY	A program on disk or tape does not match the program in memory.

Index

A

ABS, 149
Algebra, Boolean, 147
AND, 39, 258
 logical, 258
Area of a circle, 159
Arithmetic operator, 147
Arithmetic symbols, 13
Array, 115-126
 lookup table, 123
 sorting method, 186
 changing elements, 118
 clear, 119
 dimensioning, 120
 elements, 118
 number of, 119
 filling, 117
 how to display, 122
 interchanging elements, 174
 more than two dimensions, 125
 notation, 115
 putting values in, 116
 two dimensions, 121
 variable name, 118
 with loops, 116
ASC, 78, 143
ASCII codes, 69-83
 Commodore, 69-83
 character color, 79
Assignment statement, 30
ATN, 158

B

Background color, 87
Backups, 219, 224
BASIC, 2
 abbreviations, 67
 interpreter, 4
 prompt, 4
 V2, 3
Beep, 282
Binary-decimal conversion, 254
Binary numbering system, 251
Binary numbers, adding, 252
 complementary, 264
Binary operations, 250-265
 left shift, 256
 masking, 264
 right shift, 256
Bit, 252
 changing in byte, 260
Bit graphics, 266-281
 printer, 278

Bit operations, 260
 masking, 264
 set to 0 with AND, 262
 set to 1 with OR, 261
 switching, 263

Blocks, 190
Boolean algebra, 147
Border color, 87
Bubble sort, 174
Buffer, disk, 217
 keyboard, 77
 printer, 246
Byte, 250

C

Calculator mode, 13
Carriage return, 21
 suppressing, 21
Cassette recorder, 14-15,
 188-189, 221-237
 end of file, 224
 opening file, 226
Cassette index program, 230
CHR\$, 73
Channel, 192
 command, 192
 data, 192
Characters, 5
 color, 9, 10
Character set, 98, 91
 choosing, 99
 reversed, 9
Cheapsort, 186
Clear array, 119
Clear screen, 8, 20
Close file, 196, 219, 226
 rules, 196
Close printer, 242
Color codes, 95
 background, 87
 border, 87
 table, 88
Color memory map, 95
CLR, 119
CMD, 242
Comma eater, 142
Commands, 22
Command mode, 13
Commas, 59, 86, 140
Commodore ASCII, 69-83
Commodore key, 5
Comparisons, 258
 symbols, 27

Complementary numbers, 264
Compound interest, 156
Concatenate, 127
Constants, 111
CONT, 25
Control codes, 75
 printer, 246, 279
 screen, 77
Control symbols, 60-65
 special entry, 64
Converting units of measure, 149
COPY, 198
COS, 158
Counters, loop, 42
CTRL, 9
Cursor, 4, 58
 control, 7

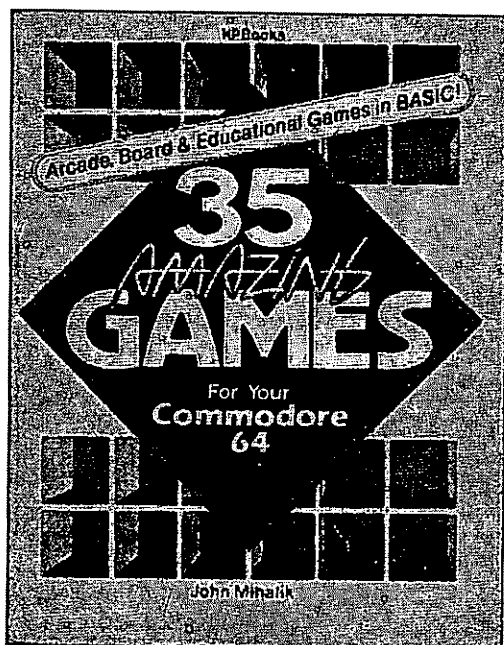
D

DATA, 111
Data file, cassette, 235
 tape, 235
Data in program lines, 111
Data pointer, 111
Data separators, 227
Decimal-binary conversion, 254
Decimal numbering system, 250
Decimal-number variable, 38
Decision process, 26
DEF FN, 158
Defining functions, 158
DELETE, 5
Device numbers, 189
DIM, 120
Dimension arrays, 120
 changing, 125
Direct mode, 13
Directory, disk, 194
Disk buffer, 217
Disk, preparing new, 15-16, 192
Disk record, 191
Disk commands, format table,
 199
Disk directory, 194
 printing, 243
Disk drive, 14-15, 188-189
 190-220
 blocks, 190
 opening, 192
 sectors, 190
 track, 190
Disk Operating System (DOS), 190
 commands, table, 191
 errors, 193

- Display, 85
- Divide by zero, 153
- Does-it-exist test, 260
- DOS wildcards, 220
- Dot graphics, printer, 278
- E**
- Editing 56-68
- Elements, array, 118
- END, 22
- Enter, 13
- Entering program lines, 56
- Error check, 204, 214
- Error traps, 99
- EXP, 155
- Exponent, 150
- Exponentiation, 150
- Expression, 13
 - evaluating, 13
 - numeric, 16
 - string, 16
- F**
- File
 - close, 196, 219
 - cassette, 221-237
 - disk, 14-15, 188-189
 - data, 188
 - open, 195
 - program, 188
 - tape, 14-15, 188-189
- Filename, 194
 - changing, 197
- File type, 192
 - program, 192
 - random, 192
 - relative, 192
 - sequential, 192
- File type designations, 202
- Flag, 137
- Floating-point
 - notation, 151
 - variable, 38
- Fonts, printer, 239
- FOR-NEXT, 44, 47
- Format utility, 200
- FRE(0), 306
- Function keys, 80
- Functions, 158
 - defining, 158
- G**
- GET, 77, 107
 - in loop, 110
 - routine, 109
 - to receive list, 110
- GET#, 197, 229
- GOSUB, 50, 143
- GOTO, 28, 41, 143
- Graphics
 - bit, 266-281
 - printer, 238-249
 - sprite, 266-278
 - symbols, 59-63
- H**
- Hexadecimal numbers, 257
- I**
- IF-THEN, 27, 143
- Immediate mode, 13
- Increment, 42
- INITIALIZE, 198
- INPUT, 87, 1-4, 107
 - from keyboard, 104-114
 - list of items, 108
 - prompt with, 107
 - to decimal-number variable, 104
 - to integer variable name, 105
 - to string variable name, 106
- INPUT#, 197
- Input-output, 188
 - devices, 84
- Insert mode, 8, 66
- Insert sort, 179
- INST DEL, 7
- INT, 149
- Integer variable, 38
- Interpreter, 4
- K**
- Keyboard buffer, 77
 - clear, 166
- Keyboard, mode, 6
 - changing, 6
- Keyword search, 134
- L**
- LEFT\$, 130
- LEN, 128
- Line feed, 21
- Line
 - logical, 20
 - program, 19, 20
 - screen, 19
 - number, 19
- LIST, 19, 241
- List program, 241
- Lists, 173
- Literal string, 16
- LOAD, 15-16, 194, 222
- Lockup, 247
- LOG, 155
- Logical AND, 258
- Logical line, 20
- Logical NOT, 265
- Logical operations, 250-265
- Logical operator, 39, 257
- Logical OR, 258
- Lookup table, 123
- Loop
 - boundaries, 42
 - counter, 42
 - counter name, 47
 - counting backward, 44
 - delay, 48
 - endless, 29
 - exits, 44
 - FOR-NEXT, 44, 45
 - in immediate mode, 48
 - nested, 50, 56
 - normal termination, 49
 - number of operations, 43
 - order of events, 43
 - terminating, 48
 - variables as counters, 46
- M**
- Mailing list, 184
- Masking, 264
- Mathematical operations, 147-160
 - order of precedence, 148
 - parentheses, 148
- Memory, 85
 - free, 304
 - locations, 85
- Menus, 161-172
 - select by function keys, 167
 - select by letter, 167
 - select by moving symbol, 168
 - select by number, 165
 - using color, 171
- MID\$, 132
- Miles per gallon, 160
- Multicolor sprite mode, 275
- Music, 282-304
- N**
- NEW, 22, 198
- NOT, 39, 265
- Numbers
 - complementary, 264
 - ordinary, 151
 - random, 153
 - range of, 153
 - storing in two bytes, 256
 - two-byte, 212
- Numeric expression, 16
- O**
- ON N GOSUB, 166
- ON N GOTO, 166
- OPEN, 192, 239

- cassette file, 226
- device, 189
- disk drive, 192
- disk sequential file, 201
- file, 195
- rules, 196
- relative file, 211
- the printer, 239
- Operating System, 4
- Operations
 - bit, 258
 - logical, 259
- Operators
 - arithmetic, 147
 - logical, 39, 257
 - relational, 127
- OR, 39, 258
 - logical, 258
- Order of precedence, 148
- Ordinary numbers, 151
- P**
- Parentheses, 148
 - nested, 148
- PEEK, 87
- POKE, 86
 - characters, 98
- Power, raising to, 150
- Powers of ten, 151
- PRINT, 16
- PRINT#, 196, 227, 242
- Printer; 238-249
 - as typewriter, 249
 - buffer, 246
 - close, 242
 - commands, 245
 - connecting, 238
 - control codes, 246, 279
 - dot graphics, 278
 - fonts, 239
 - graphic mode, 279
 - list a program, 243
 - number pages, 246
 - open, 239
 - print disk directory, 243
 - skip perfs, 245
 - substituting for screen, 242
 - tabbing, 244
 - top of form, 245
- Printing
 - from program, 244
 - list, 54, 57
 - numerics, 57
 - spaces, 57
 - strings, 57
- Program
 - calculations in, 155
 - changing, 195
 - debugging, 307-308
 - listing, 243
 - loading, 194
 - planning, 81, 305-306
 - saving, 14-15
 - saving existing, 195
 - saving to tape, 222
- Program file, 192, 194
- Program line, 19
 - adding, 19
 - changing, 19
 - deleting, 19
 - duplicating, 22
 - editing, 65
 - entering, 56
 - line numbers, 19, 23
 - spaces, 67
- Program mode, 17, 19
- Programs in this book
 - area of circle, 159
 - beep subroutine, 282
 - bubble sort, 187
 - binary-decimal, 257
 - cassette index, 230
 - comma eater,
 - compound interest, 156
 - decimal-binary, 255
 - development, 171
 - DOS errors, 193
 - format utility, 200
 - guess the secret number,
 - low and high bytes, 212
 - miles per gallon, 160
 - music maker, 290
 - printer as typewriter, 249
 - printer fonts, 247
 - roundoff, 152
 - Social Security, 157
 - sprite maker, 267
- Q**
- Quotation marks, 8
- Quote mode, 61
- R**
- RAM, 4
- Random file, 192
- Random numbers, 153
- READ, 111
- READ DATA, 111
 - multiple statements, 114
 - variable type, 114
- READY, 4
- Record, 191
- Record numbers, 212
- Relational operators, 27, 127
- Relative file, 192, 211-218
- field pointer, 212
- open, 211
- record number, 212
- record pointer, 212
- past last record, 215
- REM, 36
- Remark, 36
- RENAME, 198
- RESTORE, 10, 113
- RETURN, 13
- Reversed characters, 9
- Reversed video, 9
- RIGHT\$, 131
- RND, 153
- Rounding off, 151
 - 5/4, 151
 - algorithm, 152
 - errors, 152
 - rule, 151
- RUN STOP, 24, 223
- RUN STOP-RESTORE, 11
- RUN, 18, 41
- S**
- SAVE, 15-16, 194, 222, 225
- Scientific notation, 151
- SCRATCH, 198
- Screen
 - clear, 8
 - clear, partial, 162
 - color, 88
 - columns, 101
 - rows, 101
- Screen-display codes, 84-103
 - tables, 92-94
- Screen displays, 161-172
- Screen line, 19
- Screen location, controlling, 163
- Screen memory map, 95
- Scroll, 17, 24
 - control, 52
- Sectors, 190
- Sequential file, 192
 - number of records, 210
 - pointer, 204, 226
 - record length, 210
 - detecting end, 205
 - disk, 201-211
 - disk, erasing, 204
 - disk, open, 201, 203
 - first open, 209
 - tape, 225
- SGN, 150
- SHIFT, 5
- SHIFT-INST DEL, 7
- SIN, 158
- Social Security, 157

- Sort
 - mailing list, 184
 - number of comparisons, 176
 - strings, 179
 - upper and lower case, 182
- Sorting routines, 173-187
 - bubble, 174
 - cheapsort, 186
 - insert, 179
- Sorted list, add to, 179
- Sound, 282-304
 - ADSR, 288
 - ADSR table, 296
 - attack, 288
 - decay, 288
 - envelope, 287
 - frequency, 284
 - release, 288
 - setting ADSR, 295
 - sustain, 288
 - volume, 284
 - waveform, 284
 - waveform codes, 287
- Spaces, 8
 - in lines, 67
 - managing, 164
- SPC, 58
- Sprite, 266-278
 - color 274
 - detecting collisions, 275
 - locating on screen, 271
 - map, 269
 - moving, 272
- multicolor mode, 275
- overlay, 275
- pointers, 270
- programming, 278
- shape, 274
- turning on, 272
- SQR, 150
- Statement, 13, 22
 - assignment, 30
 - multiple, 21
- STATUS (ST), 205, 215, 229
- STOP, 24
- STR\$, 128
- String, 16, 127-146
 - changing, 135
 - length, 129
 - comparing, 127
 - shorten, 139
 - sorting, 179
- String variable, 37
- Subroutine, 141
- Subscripted variable, 116
- SYNTAX ERROR, 11
- T
- TAB, 58, 163
- Tabbing, 55
 - automatic, 59
- TAN, 158
- Tape command format, 230
- Test
 - does-it-exist, 260
 - explicit, 36
 - implicit, 36
- TI, 133
- TI\$, 133
- Time, 133
- Trap, error, 99
- Trigonometry, 158
- True/false operations, 258
- Two-byte numbers, 256
- Type-designation symbols, 37
- U
- Upper and Lower Case, 6
- Upper Case and Graphics, 6
- User-defined functions, 158
- V
- VAL, 108
- VALIDATE, 198
- Value, 118
- Variable, initializing, 42
- Variable name, 31
 - array, 118
 - decimal-number, 38
 - integer, 38
 - string, 37
 - rules, 33
- Variable, subscripted, 116
- VERIFY, 223
- Video, reversed, 9
- W
- Wildcards, DOS, 220



If you enjoyed this book, you'll also have fun with **35 AMAZING GAMES For Your COMMODORE 64** by John Mihalik, also published by HPBooks®. It contains entertaining and challenging arcade, board and educational games in BASIC.

*Other Fine Computer Books
From The Knight-Ridder Press*

35 Amazing Games for Your Commodore 64
35 Amazing Games for Your Commodore 128
The Essential Commodore 128 User's Guide
The Essential Atari 520ST & 1040ST User's Guide
BASIC Program Conversions
Quick & Easy WordStar 2000
How to Program Your IBM PC: BASIC for Beginners
How to Program Your IBM PC: Advanced BASIC Programming
How to Program Your IBM PC: Color & Graphics

Learn to Use Your COMMODORE 64—the Easy Way!

No experience necessary—this book starts at the beginning. Clear, straightforward example programs help you understand how programs work and how your computer works. How to use and enjoy essential hardware—including cassette-tape storage, disk drive and printer.

You'll learn the basics of BASIC: How to write and use loops. Build and use arrays. Edit and improve programs. Finding and fixing problems. Math operations, strings, storing data on disk or tape. How to use the printer. All about exciting features like sound, music, graphics and color.

Now you can really master your amazing COMMODORE 64!

Reviewers Praise *How to Program Your Commodore 64, BASIC FOR BEGINNERS*

“A very high rating . . . I recommend it to anyone interested in learning to program BASIC on the Commodore 64.”

— *RUN Magazine*

“A treasury of well-organized, entry-level information.”

— *American Library Association's Booklist*

“Good value . . . A well-written guide.”

— *The Province*



\$12.95
\$18.50 Canada

ISBN 0-89586-310-3