SYBEX

THE EASY GUIDE TO YOUR COMMODORE 64™

JOSEPH KASCMER

THE EASY GUIDE TO YOUR COMMODORE 64

THE EASY GUIDE TO YOUR COMMODORE 64®

JOSEPH KASCMER

SYBEX® Berkeley • Paris • Düsseldorf

Cover art by Sato Yamamoto
Book design by Ingrid Owen

Library of Congress Card Number: 83-40232
ISBN 0-89588-126-8
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

To J. A. Kubokawa, a researcher and photographer with a passion for quality.

CONTENTS

ACKNOWLEDGEMENTS

Thanks to James Compton for his editing and to those in the editorial and production departments at Sybex who were active in turning my manuscript into this book.

Special thanks to Heidi Miller for preparing the index and to Bill Mlotok for bringing his technical knowledge, classroom experience, and good sense to a valuable review of the book's content.

Also thanks to Paul Losness of PC Computers in El Cerrito, CA, for providing information.

The poetry on page 11 is by Philip James Bailey, from his *Festus*. The verse on pages 12-13 is from Robert W. Service's "The Law of the Yukon."

PREFACE

This book is about controlling your personal computer. In it you will discover how you can achieve control over a computer in a few days. Jargon and theories of computer science don't help in this aim, and in this book you won't be burdened with them. To direct the computer as an extension of your own mind, you need no special background in mathematics or bent for programming. In fact, you can control a computer just as easily as you do an automobile or typewriter. As you learn the essentials, the operation becomes simpler.

If you have a particular use for the Commodore 64, or if you're interested in finding out about control of a personal computer, this book is for you.

Because you can use a computer in different ways, you can approach this book from more than one direction:

Chapter 1 is a brief introduction telling where the controls are, how to plug your computer system together, and what buttons to push and not to push. No matter what you plan to use your computer for, a run through this chapter will put you at ease with the box in front of you.

If you plan to use only prerecorded programs and wish to avoid writing computer directions yourself, you can skip ahead to **Chapter 7** if the programs are on cassette tape, or to **Chapter 8** if they are on disks. And if you'd like to make full use of the programs on disks, read the second part of **Chapter 9**. Since the Commodore 64 isn't yet as well supplied with commercial software as some older personal computers, you may not want to limit yourself to these chapters.

If you'd like to write your own directions to the computer, you should read **Chapters 2, 3, 4, 5** and **6**, and also the first part of **Chapter 9**.

The most optional chapter of the book is **Chapter 10**. If you'd like more control over the Commodore 64 than is provided through the BASIC commands, or have an interest in how the computer works and what else can be done with a more customized computer, read Chapter 10. It will help you to understand the machine and to expand your abilities with it. For using the Commodore 64, however, it's not essential.

The chapters concentrate on the commands and operations most useful to the general reader. You'll find other available commands built into the Commodore 64 listed in **Appendix A** according to their type and use.

In the interest of clarity and ease of comprehension, jargon has been avoided throughout the chapters of this book. However, you may encounter other computer users, salespersons or books lapsing into the dialect. **Appendix B** can help make them intelligible to you.

A decade before the first Commodore 64 was built, the idea of a "personal computer" with such capabilities was unheard of. Its current availability is a measure of the changes brought about by the technology developed in the intervening years. More changes lie ahead, and in learning to control a computer, you are taking a first step toward preparing for those changes.

Editor's Note

To get the most out of this book, you should read it with your computer in front of you, set up and connected to a video monitor or TV set as described in Chapter 1. As each new command, group of commands, or sample program is presented, type it in *exactly as printed* to see the results on your screen. Be sure to type in each character that you see: letters, numbers, punctuation marks, and spaces. (In typeset program lines, the space equivalent to one stroke of the space bar on your keyboard will vary slightly, but it will always be approximately equal to the space between words in this sentence.) The spaces are not absolutely essential; any program will run without them, but you will find the screen display much easier to read if you include them.

After you type in a sample program, and before you run it, check what you have typed against the book. If you find you have made a mistake, go back and correct it. (You'll see how to do that in Chapter 2.) The programs have all been carefully tested, and will run as described if entered correctly.

Finally, you will see that the Commodore 64's command words (or "reserved" words) have been set in **boldface** type. Boldfacing is simply a typographical convention, meant to call your attention to these words and help you to remember them. You will not see this effect on your screen, and you need not try to reproduce it.

1 COMING INTO POWER

MASTER AND SERVANT MEET

Facing a computer for the first time, people often feel powerless over the machine—and for a good reason. After all, when it comes to computers, knowledge is power, and we all start off ignorant.

A worker of incredible speed and unimaginable precision, the computer is nonetheless a completely obedient servant. It only needs directions, stated in terms it was built to recognize. You can command the Commodore 64 with a vocabulary no larger than that understood by a highly-trained canine. A computer following your commands can choose among alternatives, organize information, search for facts and patterns, and control devices. It can be ordered to begin any duty in response to a single word or by the triggering of a single event.

No computer is all-powerful or all-knowing, and there are limits to the operation of any machine. But those limits arise from the design and speed of the computer and the size of its memory.

Although you can gain control of the Commodore 64 in a short time, to do so, you must first learn to become its master. Through this book you'll find a clear way to do just that.

SETTING UP THE COMPUTER

The Commodore 64 computer can be set up by anyone who can position a typewriter and hook up a television. It can work anywhere a typewriter might, in any place undisturbed by strong vibrations, or extremes of temperature, dust, or humidity. In fact, the Commodore 64 will work in most places where its operator is comfortable. A typical set-up is shown in Figure 1.1. The television (or video monitor) you'll use for a display screen can sit anywhere reached by the cable from the computer.

You can prepare the Commodore 64 for use by connecting only two electrical cords, one for power and one for the television or monitor. One end of each cord plugs into the computer. On the right side of the Commodore 64, shown in Figure 1.2, you'll see a switch marked ON, and a circular jack next to the label POWER. Into that jack you can plug the sleeve-like end of the power cord. *With the switch turned to OFF*, you can plug the three-pronged end of the power cord into a wall outlet to complete the power connection.

The connection to a television is just as simple. Near the center of the computer's back panel, and next to a small switch, you'll find a small, projecting metal jack, which takes the televison cable. The cable has the same post-in-a-sleeve metal phono plug at each end. One end plugs into the metal television jack at the back of the computer; the other end fits into a jack on a small box that allows you to switch the monitor between television reception and computer display. From that box (known as a "switch box") come wire leads that you can attach to the television in place of the vhf antenna leads. If you next connect the vhf leads to the other side of the switch box, you can switch between television reception

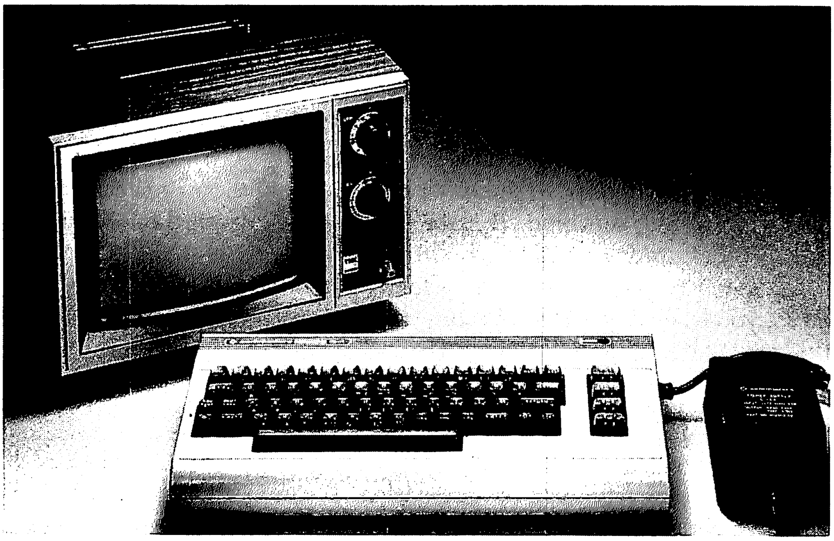


Figure 1.1: A typical Commodore 64 computer set-up.

and computer display. Figure 1.3 shows the system connected for computer use only.

The final adjustments include setting your television tuner to either channel 3 or channel 4, whichever one is not used for broadcast in your

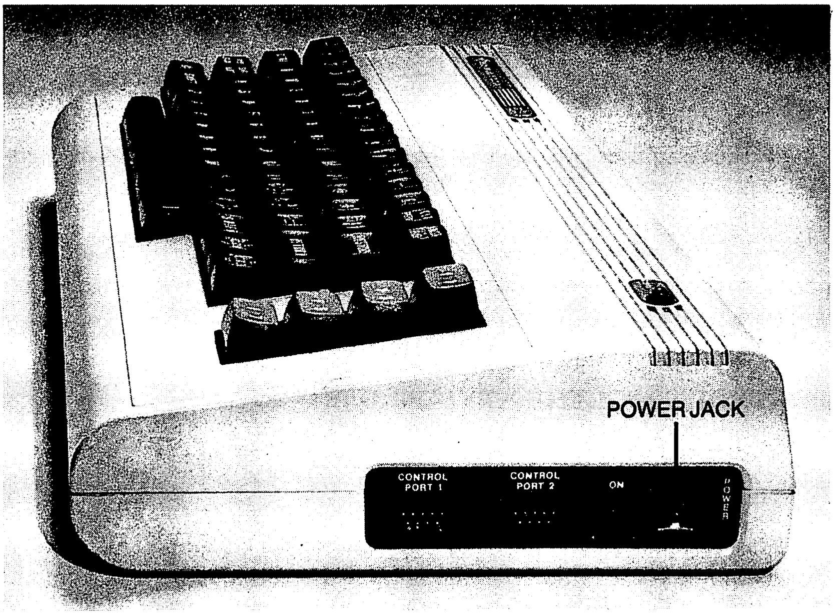


Figure 1.2: Power connection to the computer.

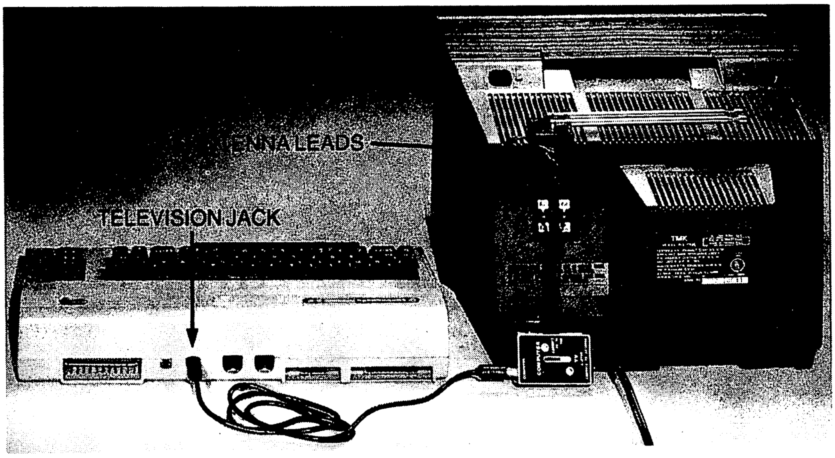


Figure 1.3: Connections between computer and television.

area. Also set the channel on the computer. If channel 4 is unused, slide the small, recessed switch at the back of the computer (next to the metal television jack) toward the jack. If it's channel 3, slide it away from the jack.

Now that the connection between computer and television is complete, you can use the computer by setting the switch on the box to COMPUTER. To use the television for tv reception, slide the switch to TELEVISION.

If you use a video monitor, a different cable—one ending in a five-pin plug at the computer end and a phono plug at the monitor end—connects to a flat, circular video jack at the back of the computer, next to the television jack, as shown in Figure 1.4.

Once connected to a television or monitor, the computer before you is part of a fully operational system that can display on the screen whatever you type.

CONTROL FROM THE KEYBOARD

All you need to command the computer sits before you. The keyboard is laid out like a typewriter's, but has a few extra keys that give you more control than is possible with a typewriter.

Before turning on power to the computer, consider a simple, comforting fact: Nothing you can do from the keyboard will damage the computer. With some of the commands you'll find in this book, you can change the screen display, the contents of the computer's active memory, or even the

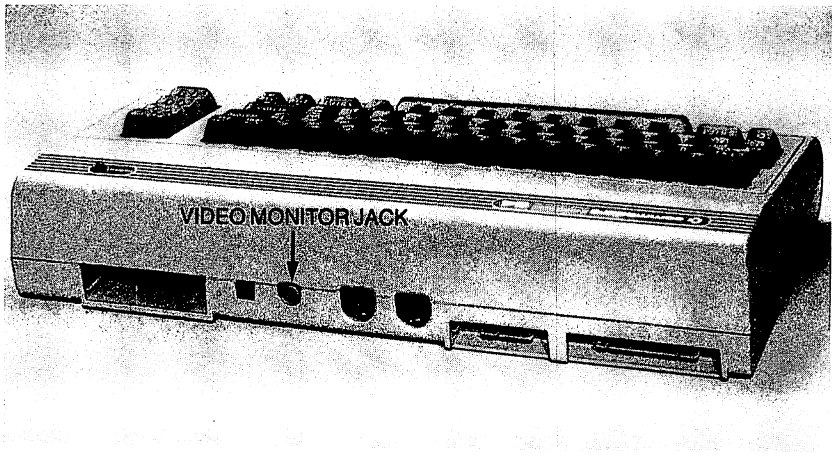


Figure 1.4: The video monitor jack.

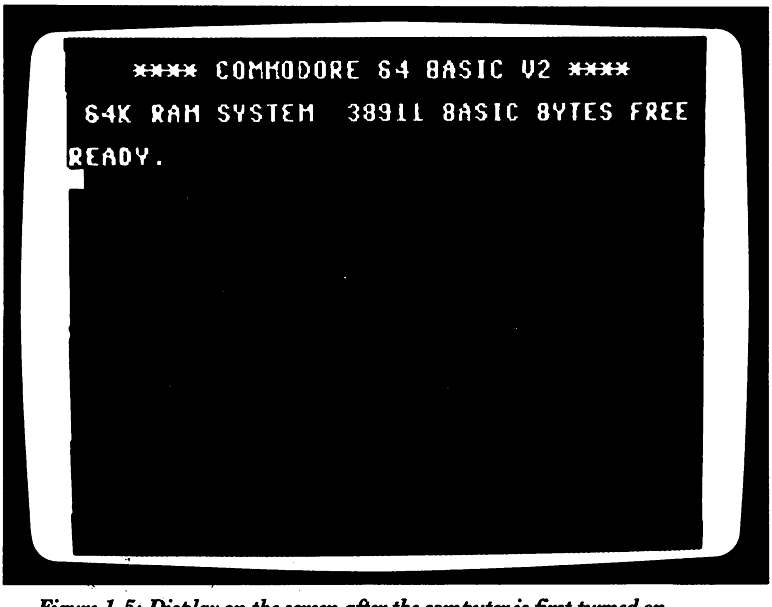
way the computer works, but when you turn the power to the Commodore 64 off and then on again, you wipe clean the changes you've made. The computer then greets you in readiness, all previous keyboard manipulations forgotten.

The single most useful bit of advice while you're learning to operate the computer from the keyboard is this: If something interests you, try it. The worst that can happen will be the loss of some command, information or calculation. In the earliest stages, while you're experimenting, such a loss is part of learning to control the beast. No matter what happens, the computer will emerge from human keyboard error or curiosity completely undamaged.

When you switch on first the monitor or television, then the computer, printing will appear on the screen. Across the top, if all is connected and working properly, three lines will be printed as light letters on a dark screen, as in Figure 1.5.

The Cursor

At the upper left of the screen you will see the word READY, and below that a character-sized flashing square; these are signals from the computer



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

Figure 1.5: Display on the screen after the computer is first turned on.

that it's ready to read whatever you type from the keyboard. The flashing square marks where the next character from the keyboard will appear on the screen, and is known as the *cursor.*

The Commodore 64 is designed to display immediately what's typed from the keyboard, and will print light blue letters on a dark blue screen as you type them. The word READY is a signal from the computer that it's prepared to accept a command. But whether you type one of the scores of commands the Commodore 64 can act on or a line of poetry, your typing will appear immediately on the screen. And as you type, the computer waits for a signal from you that will tell it to act on the command you've just typed.

The RETURN Key

You signal the computer to act on a typed command by pressing the RETURN key. If the command is no more than two lines long and the cursor is on one of those lines, the computer will compare the typed command with a set of built-in directions, and take action. But if you merrily type along without pressing the RETURN key, the computer will follow directions that tell it to simply display characters on the video screen.

The SHIFT and C= Keys

Like a typewriter, the Commodore 64 can display both uppercase and lowercase letters. But unlike a typewriter, it can also display certain graphics symbols. These are the small boxed designs on the front of most of the keys. When you first switch on power to the Commodore 64, the computer is automatically readied to display the uppercase of any letter key that you press. Press a number key and the number will be displayed. Press a key with punctuation and that mark will be displayed. By pressing certain combinations of keys, you can also type the graphics designs shown on the front of each key. If you hold down the SHIFT key while pressing a key with graphics symbols, the symbol on the right of the key will be displayed. Hold down the C= key (or "Commodore" key), next to the SHIFT key, while pressing a key, and the symbol on the left of the key will be displayed.

Each time you turn on power to the computer, it is automatically set to recognize keystrokes in the uppercase-graphics mode. You can switch it into another keystroke mode by pressing down the SHIFT and C= keys together once, and then releasing them. Once this is done, the computer treats a solitary keystroke as a signal to display a lowercase character. Pressing the SHIFT key and a letter key will now produce an uppercase character, and pressing the C= key and a graphics symbol key will produce the character shown on the left of that key, as it did before.

You switch out of this lowercase-and-uppercase mode and back to the uppercase-and-graphics mode by pressing SHIFT and C= again. In fact, each time you press the SHIFT and C= keys together you switch the computer from one display mode to the other. Any typing already on the screen will also change when the computer is switched from one display to the other. Together, the SHIFT and C= keys act as your switch to control the display from the computer.

The SHIFT LOCK key can be set to either of two positions. When pressed down and allowed to click into a lowered position, its effect is the same as holding down the SHIFT key. Pressed again, the SHIFT LOCK key will click into a raised position, and there is no such effect.

Moving the Cursor: the UP/DOWN and RIGHT/LEFT keys

Treating the screen of the Commodore 64 like the video equivalent of a sheet of paper in a typewriter, you can type on it anywhere within its preset margins. With the two arrow keys in the lower-right corner of the keyboard, you can move the flashing cursor to any location on the screen, and whatever you type will appear at that point. When you press it alone, the key marked with up-and-down arrows (we'll call this the CURSOR UP/DOWN key) moves the cursor down on the screen. Likewise, the key marked with right-and-left arrows (we'll call this one CURSOR RIGHT/LEFT) moves the cursor to the right. If held down, both of these keys continue this cursor movement until released. When either the SHIFT, SHIFT LOCK or C= key is held down while the CURSOR UP/DOWN key is pressed, the computer moves the cursor up the screen. Likewise, the CURSOR RIGHT/LEFT key moves the cursor to the left when it's used in combination with any of those three keys. Once the cursor is at the bottom of the screen, the down arrow cranks up all the typed lines like printed lines on an endless roll of paper being fed through a typewriter.

Inserting, Deleting, and Clearing: INST/DEL and CLR/HOME

Two other keys allow you to control the display further. The key in the upper-right corner of the keyboard marked INST DEL serves two functions. It can insert or delete. When pressed by itself, it directs the cursor to erase and close up one space to the left of the cursor. When pressed while the SHIFT key is held down, it directs the computer to open a space to the left of the cursor and move the cursor there, so another character can be typed. If you hold down this key in either mode of operation, it will repeat the effect until you release it.

The key marked CLR/HOME, next to it, works in two ways also. You can use it to direct the computer to send the cursor "home," to the upper-left corner. Or, pressing SHIFT and that key together (SHIFT-CLR/HOME), you can direct the computer to clear the screen and send the cursor up to that corner.

More Keys for Control

The three keys marked RUN STOP, CONTROL, and RESTORE can be used to alter the computer's actions when it's under the control of a program of commands.

The four rightmost keys, marked f1 f2, f3 f4, f5 f6, and f7 f8, can each be called on by programs to control the computer in specially programmed ways.

RETURN Revisited

By sending a signal to the computer to act on what you've typed at the keyboard, the RETURN key works as your messenger in routing keyboard statements into the *translator* of the Commodore 64. When you press the RETURN key after typing something in response to a READY prompt, the computer's translator compares what you've sent with the vocabulary of words built into it. If the form of your typed statement matches a command, the translator passes the orders on to the processor part of the computer, which takes action according to built-in directions.

You can move the cursor anywhere around the screen and type anything you like. If you press the RETURN key and the translator doesn't match your instructions with its vocabulary, the computer will signal you that there is a mismatch. In the Commodore 64 this signal takes the form, "?SYNTAX ERROR". This type of "error" reply from the computer simply means there has been a momentary lapse in commmunication. ("Syntax" refers to the exact form or sequence in which the computer expects commands to be given.) In any case, the RETURN key is one you'll find useful when you're ready to give commands to the computer.

Summary of Keyboard Operations

Each key has an effect (prints a character or sends a command), and some have several different effects when used in combination. You can acquire programs (sets of commands) that will change the effect of any or all of the keys. Pressing any key simply provides a signal to the computer, which responds to that signal by printing a character or taking some preplanned action. All the keys are shown in Figure 1.6.

Whether you are using prerecorded commands (from tapes or disks, as explained in Chapters 7 and 8) or direct commands, you ultimately control the Commodore 64 through the keyboard. The keyboard serves as reins and bridle to the computer and any devices attached to it.

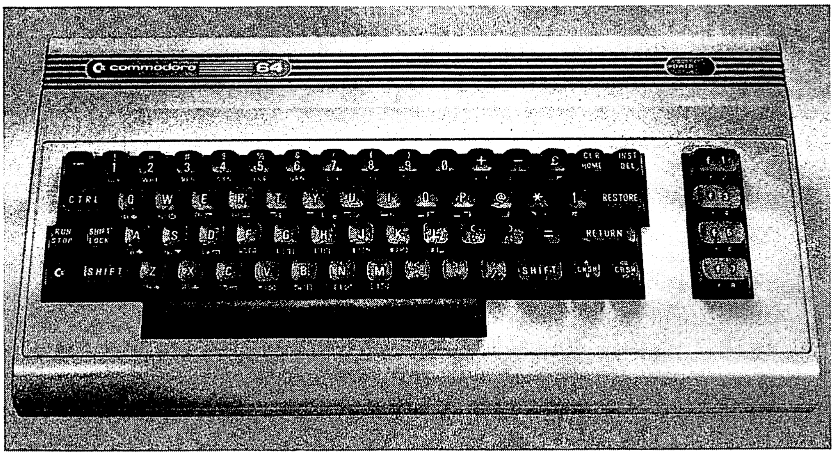


Figure 1.6: The Commodore 64 keyboard.

2 VIDEO PRINTING

Johann Gutenburg set the standard for printing five hundred years ago, and it really hasn't changed much until recently. In fact, only the introduction of the computer in the last few years has had a major impact. Typesetting by computer puts the printed page on a screen before it goes on paper. With your Commodore 64, you can create screen displays controlled by command.

SCREEN CONTROL

In simply relaying your keystrokes to the video screen, your Commodore 64 draws upon none of its real capabilities. With commands matching those to which it responds, however, you can direct the computer in preplanned displays. To some commands, the computer reacts instantly when you send them with a press of the RETURN key. Among these "immediate" commands are those which enable you to fashion the screen display.

You can fill the screen with any characters you like by typing on and on from the keyboard without sending the computer a recognizable command. If you type more than a screenful, the computer will move the entire display up one line to make room for the most recent line, and lose the topmost line from the screen.

You can direct the Commodore 64 with single keystroke commands as well as typed word commands. Pressing the SHIFT and CLR/HOME keys at the same time, for instance, directs the computer to clear the screen of all printing and to move the flashing cursor to the upper left corner. When given this command, the computer presents a blank screen, regardless of what was typed there before. If you wax poetic, for instance, and type onto the screen something like the lines in Figure 2.1, you can remove what you've typed at once by doing the following:

(press SHIFT and CLR/HOME)

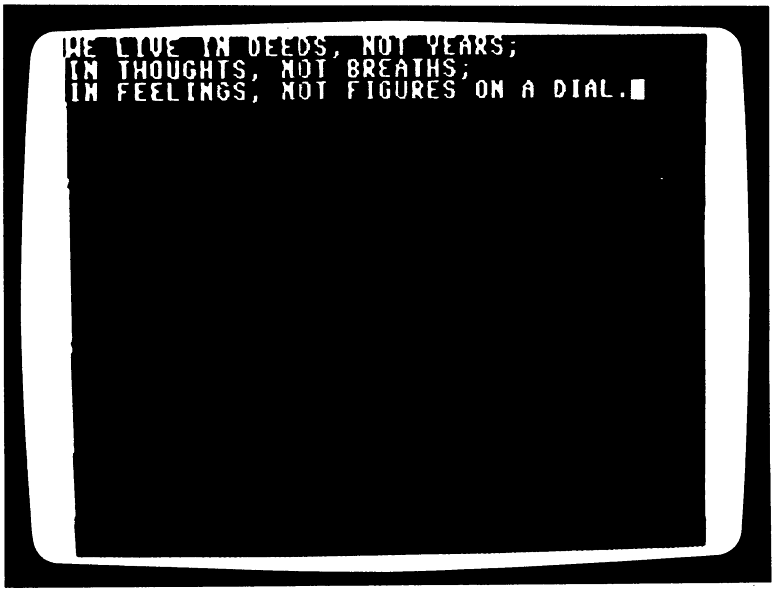
Directed by this keystroke command, the computer erases all printing, leaving a blank screen with the cursor flashing in the upper-left corner—the video equivalent of a fresh sheet of paper and poised pen.

When you type keyboard characters, the computer sends them to the screen as light-colored dot formations on a dark background. It answers you in the same way, with a light-on-dark display it chooses automatically. There are, however, alternative ways of presenting text on the screen, useful in varying the display or calling attention to particular parts of text.

If you want dark characters to appear, each on its own light-colored background, you can direct the Commodore 64 to print them that way with the keystroke command:

(press CTRL and 9)

and everything that follows on the screen will be printed in what is called



```
WE LIVE IN DEEDS, NOT YEARS;
IN THOUGHTS, NOT BREATHS;
IN FEELINGS, NOT FIGURES ON A DIAL.█
```

Figure 2.1: One possible screen display.

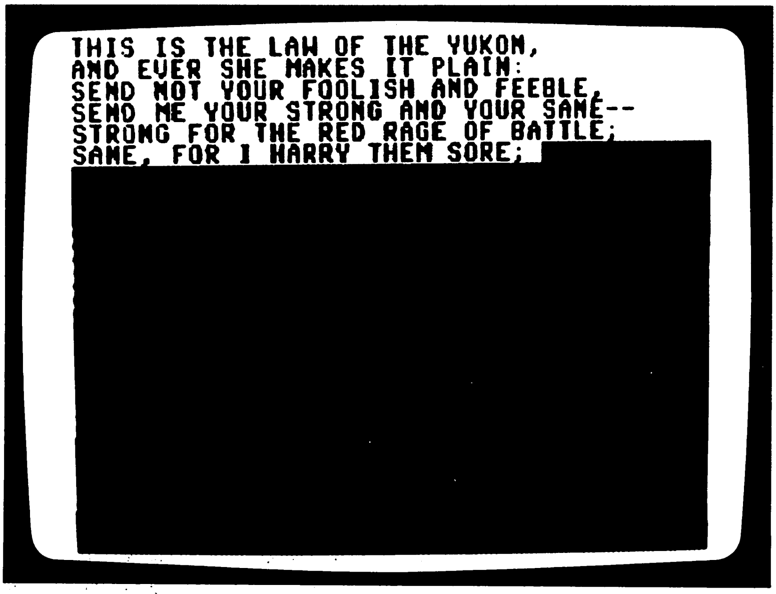
the "reverse-video" mode. The CTRL-9 command controls the computer like a switch, so that until some other command is sent to change the screen display (or the RETURN key is pressed), all new printing on the screen will appear as dark-on-light characters. Characters typed before the CTRL-9 keypress command was sent remain as light-on-dark, the "normal" display mode the computer always presents automatically, until otherwise directed. The poet Robert W. Service, giving the CTRL-9 command at the beginning of a verse, would have seen a screen like that in Figure 2.2.

You can switch the computer back into the normal light-on-dark display mode with the keystroke command:

(press CTRL and 0)

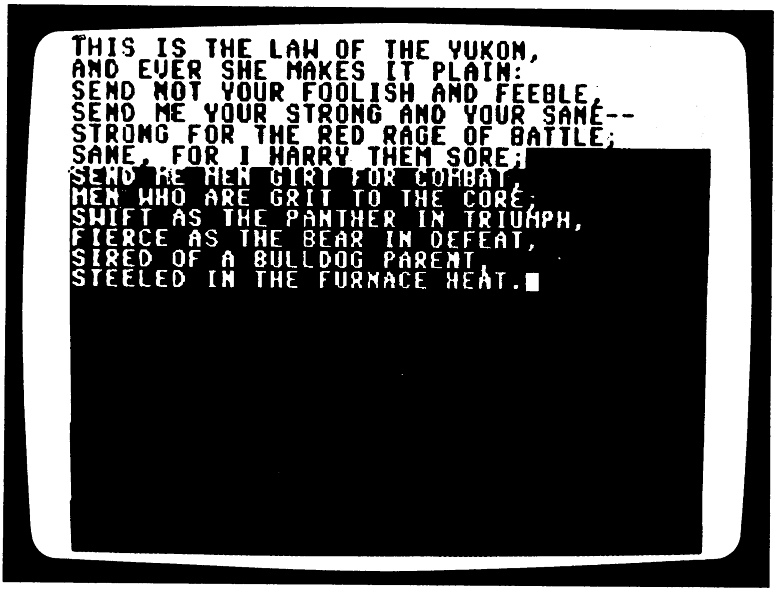
which directs the computer to print light-on-dark every character typed after that command is sent. The same poet, advancing through his poem after pressing CTRL and 0, would see the screen display shown in Figure 2.3.

You can also create multicolored displays of text. Using the keys numbered 1 through 8 with the CTRL key, you can choose different colors for characters printed on the screen. To get yellow letters, for instance, you



```
THIS IS THE LAW OF THE YUKON,
AND EVER SHE MAKES IT PLAIN:
SEND NOT YOUR FOOLISH AND FEEBLE,
SEND ME YOUR STRONG AND YOUR SANE--
STRONG FOR THE RED RAGE OF BATTLE;
SANE, FOR I HARRY THEM SORE;
```

Figure 2.2: The effect of the CTRL-9 command.



```
THIS IS THE LAW OF THE YUKON,
AND EVER SHE MAKES IT PLAIN:
SEND NOT YOUR FOOLISH AND FEEBLE,
SEND ME YOUR STRONG AND YOUR SANE--
STRONG FOR THE RED RAGE OF BATTLE;
SANE, FOR I HARRY THEM SORE;
SEND ME MEN GIRT FOR COMBAT,
MEN WHO ARE GRIT TO THE CORE;
SWIFT AS THE PANTHER IN TRIUMPH,
FIERCE AS THE BEAR IN DEFEAT,
SIRED OF A BULLDOG PARENT,
STEELED IN THE FURNACE HEAT.
```

Figure 2.3: The effect of the CTRL-0 command.

can press together the CTRL and 8 keys:

(press CTRL and 8)

If the computer is in a light-on-dark display mode, all characters typed after the CTRL-8 command will appear as light yellow letters on a dark blue screen. If the command for reverse character display is given, CTRL-9, the display will be dark blue letters, each on its own yellow block. You can switch the character display into other colors by pressing the CTRL key together with any of the other number keys. When you press the C= key and one of the numbered keys, the computer will display even more colors.

VIDEO PRINTING

The Commodore 64, like any computer worth its circuits, can do more than display your keystrokes on a video screen. In fact, much of the Commodore 64's computing power lies in the built-in operations it works on furiously at your command, without giving so much as a clue to what it's doing. The simplest example can be seen with elementary arithmetic, which the Commodore 64 works on automatically and unseen when so directed. The familiar calculator functions of addition, subtraction, multiplication, and division are commanded from the keyboard with the symbols +, −, *, and /. If you direct the computer to multiply −2 and 4, by typing:

```
-2*4
```

you won't see the result, just a ?SYNTAX ERROR reply and another READY prompt. To see the results of that multiplication, you must add a command.

The PRINT Command

The command that directs the computer to display the results of arithmetic operations on the screen is PRINT. It can be used with an "operator" command like the multiplication sign, *, to display one of the computer's internal operations. You type this useful command first, before the operation you'd like displayed, like this:

```
PRINT -2*4
```

As used here, the PRINT command directs the computer to display the result of the multiplication that follows it. After sending this command by pressing the RETURN key, you'd see the value −8 on the screen and a new READY prompt below that, signalling the computer's readiness for another command, as shown in Figure 2.4.

You can put the PRINT command into service for any combination of arithmetic commands according to the rules of algebra, to operate your sophisticated Commodore 64 as a simple calculator, with entries like this one:

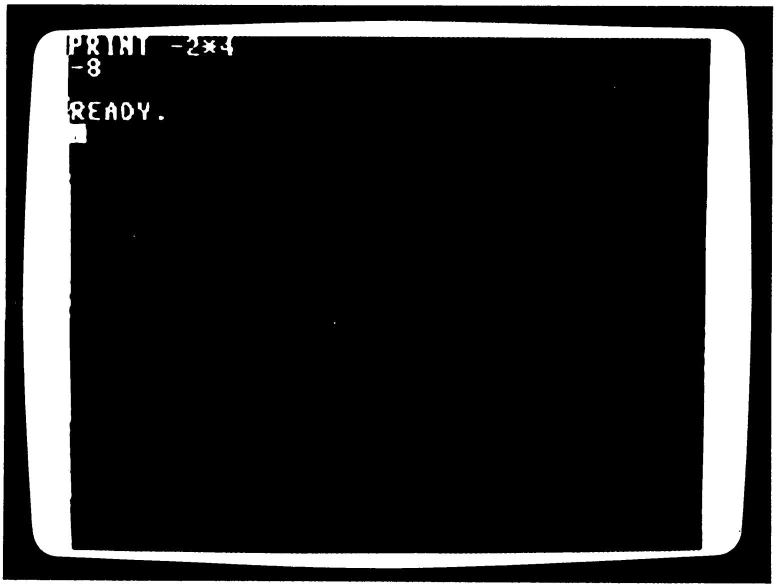
```
PRINT 9+8-7*6/5
```

The computer automatically follows a predetermined order of operations in interpreting arithmetic expressions like the above. First multiplication, then division, and finally addition and subtraction are carried out, in this case to give the result 8.6 on screen. To change this built-in sequence of operations, parentheses are used to direct the computer to work out the values inside them first, like this:

```
PRINT (9+8-7)*6/5
```

for a result that will appear, below the PRINT statement, as 12.

But the PRINT command's usefulness goes beyond simply displaying mathematical manipulations (all of which are listed in Appendix A).



```
PRINT -2*4
-8

READY.
```

Figure 2.4: The effect of the PRINT command.

PRINT is a command you can use often. As well as applying it to direct some internal operation onto the screen, you can use it to express a sentiment, a thought, or a fact on the screen in the midst of some other computer action. The PRINT command is your typist.

You can direct the computer to print on the screen nearly anything you type *in quotation marks* following the PRINT command. For example:

```
PRINT "So glad to be here."
```

The computer's response after you press the RETURN key will be the words,

```
So glad to be here.
```

on the line following the command.

Words and numbers can be combined in a PRINT statement. For instance, you can make the statement:

```
PRINT "The winning number is..." 30
```

and see the computer reply:

```
The winning number is... 30
```

You can even slip a calculation into a printed sentence, like this:

```
PRINT 20 "or" 30 "is more than I'm willing to pay. But" 10 + 5
"sounds right."
```

for the display from the computer:

```
20 or 30 is more than I'm willing to pay. But 15 sounds right.
```

Keystroke Commands in the PRINT Statement

You can load a PRINT statement with keystroke commands, enclosing them within quotation marks, and the computer won't perform them until you press the RETURN key to send the entire PRINT command. One PRINT statement can thus hold directions to do several things. For instance, you can instruct the computer to clear the screen, switch the display to "reverse type" (dark-on-light characters), switch it to yellow, then print a phrase, switch to white, and finally print another phrase, all with a command like the following:

```
PRINT "(press SHIFT-HOME)(press CTRL and 9) (press CTRL and 8) ↑ ↑
↑ TAKE IT(press CURSOR UP/DOWN) (press CURSOR UP/DOWN)
(press CTRL and 2)FROM THE TOP."
```

TAB—Arranging the Screen Display

If you give a command from the keyboard, like SHIFT-CLR/HOME, within quotes in a PRINT statement, the computer will record that command in a coded way, in this case with a heart symbol, but will not carry it out until the RETURN key is pressed. You can use this to delay to tailor a display with the PRINT command. Once you realize that the computer handles text on the screen as a grid of 40 columns by 25 rows, you can arrange displays to suit the eye. A handy little command used within a PRINT statement directs the computer to *tab* (or space over to a specified column) before actually printing on the screen. The form is TAB( ), where the columns are numbered, like a typewriter's, from left to right, the leftmost as 0 and the rightmost as 39. You can add the TAB command before the item to be printed, like this:

```
PRINT "HEY!" TAB(15) "LOOK" TAB(30) "AT THIS ←"
```

The computer acts on this statement by reading it across as a series of commands, and prints:

```
HEY!           LOOK           AT THIS ←
```

If a command is already displayed on the screen you can repeat it by moving the cursor (with the CURSOR UP/DOWN key) to the screen row in which that command is found. If you press the RETURN key then, you will again send the command for the computer to act on. You can even alter a displayed command, and then send it by pressing the RETURN key.

You could, for instance, move the cursor back up the screen to the PRINT command described above, using the SHIFT and CURSOR UP/DOWN keys. Using the CURSOR RIGHT/LEFT key to move the cursor horizontally, and the SHIFT and INST/DEL keys to insert spaces, you can change the statement. One way to change the statement would be so that the display read:

```
HEY!          TAKE A LOOK          AT THIS ←
```

To do this you insert the words TAKE A within the second set of quotes. You can do this by placing the cursor over the letter L in the word LOOK, and there inserting seven spaces, for the words TAKE A and the blank space between them. To insert the spaces, you just press the SHIFT and INST/DEL keys seven times together.

```
(press SHIFT and INST/DEL)(press SHIFT and INST/DEL)(press
SHIFT and INST/DEL)(press SHIFT and INST/DEL)(press SHIFT and
INST/DEL) (press SHIFT and INST/DEL)(press SHIFT and INST/DEL)
```

Then you can make your insertion:

```
TAKE (press SPACE BAR) A
```

The command that remains after these changes is:

```
PRINT "HEY!" TAB(15) "TAKE A LOOK" TAB(30) "AT THIS ←"
```

If you press the RETURN key now, you'll see a changed display result:

```
HEY!          TAKE A LOOK          AT THIS ←
```

You can use such an approach to send any command from the screen into the computer for action. If the command statement takes up two lines on the screen, you can send it by positioning the cursor anywhere on either of the lines, and then pressing the RETURN key.

More Commands . . .

Two punctuation marks can be used as shorthand commands within a PRINT statement. Commas typed between items in a PRINT command will direct the computer to space the items four across the screen, as though following a succession of TAB(2), TAB(12), TAB(22), and TAB(32) commands. Semicolons typed between items will separate them in the statement, but not in the printing. Also, the computer automatically adds a space before and after numbers that it prints.

The computer and its screen provide a more controllable display than the familiar typewriter and paper. You can direct color displays, and even a rapid succession of displays by implanting screen-clearing commands between displays.

By using still more variable commands, you can guide the servile computer in one direction or another. By using commands in combination, you can even send it off to do your bidding on its own.

3 PLANS AND PROGRAMS

SCHEDULING COMMANDS

The computer, long regarded as an awesome accumulator of information and a secret keeper of files, is basically forgetful. That is, it's built to act on commands, and then automatically leave them behind to ready itself for more commands. Human beings often work much the same way, returning to dimly-remembered locations only by the grace of street addresses we recall.

In its own way, the computer can return to "addresses" held in its memory. If commands wait at such addresses, it will then act on them as if you had just sent them. You can provide such addresses by starting each group of commands with a number.

Placed at the beginning of a group of commands, a number tells the computer to retain those commands as part of a *program*, possibly to be used again. For example, to mark a statement of display commands in this way, you could send the following command:

```
10 PRINT "(press SHIFT and CLR/HOME)" TAB(12) "NUMBERS AND
   THEIR USES"
```

You'll notice, when you press the RETURN key after typing a numbered statement like this, that the computer replies with a flashing cursor, without acting on the commands. Actually, the statement is sent into storage in the computer's memory, where it is kept on file by number. Here, the commands will be held as line number 10 until you type the command that tells the Commodore 64 to proceed with them. That command is:

```
RUN
```

which directs the computer to translate and act on the numbered statements in its memory, in numeric order. As the only numbered line of commands in memory in this case, line 10 will be acted on command-by-command. The computer automatically holds numbered commands in memory, even after acting on them, so you can tell the computer to carry out the commands of line 10 as many times as you like by sending the command RUN each time.

Although commands may reside unseen in the computer's memory, you can display them on the screen with a simple command. If you send the command

```
LIST
```

the computer will respond by printing the entire contents of its program memory—in this case, line 10.

You can send another set of commands for the computer to act on after line 10 by labeling the statement with a higher number, like this:

```
20 PRINT "(press CTRL and 9) (press CURSOR UP/DOWN) THIS
   LINE WILL BE REMOVED."
```

The computer now has two lines in its memory and, when sent the command:

```
RUN
```

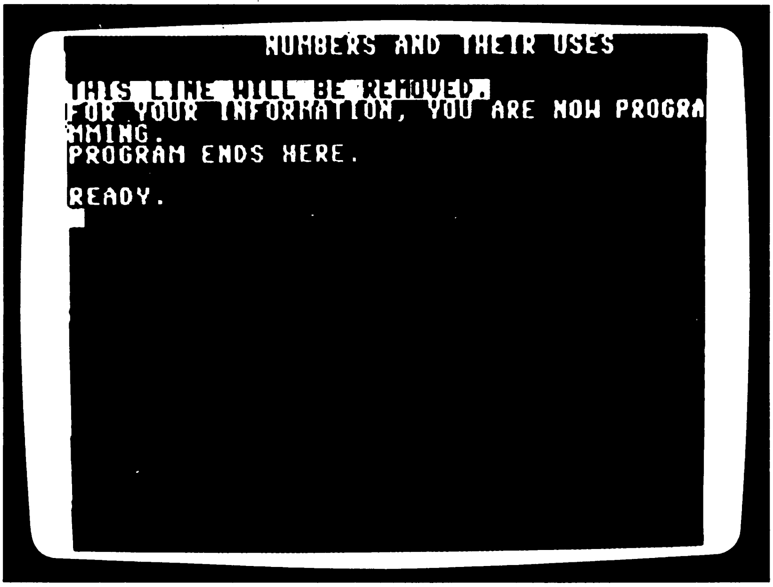
will start with the lower-numbered statement, 10, translate and carry out each command, and then translate and carry out the commands of statement 20.

You can direct the computer, with a series of numbered statements, to switch among different screen printing displays to fashion a result like that produced by these additional statements:

```
30 PRINT "(press CTRL and 8) FOR YOUR INFORMATION, YOU ARE
   NOW PROGRAMMING."
40 PRINT "PROGRAM (press C= and 7) ENDS HERE"
```

If, after sending these statements to the computer, you give the command RUN, you'll see a display of four statements appearing one after another with a final result like Figure 3.1.

Statement lines 10, 20, 30, and 40 make up a program that can direct



```
           NUMBERS AND THEIR USES

THIS LINE WILL BE REMOVED.
FOR YOUR INFORMATION, YOU ARE NOW PROGRA
MMING.
PROGRAM ENDS HERE.

READY.
```

Figure 3.1: The result of running a program of display commands.

the computer, each time you issue the RUN command, to produce the same printing display. Change the statements in the program and you change its results. For instance, the reverse-video row that declares:

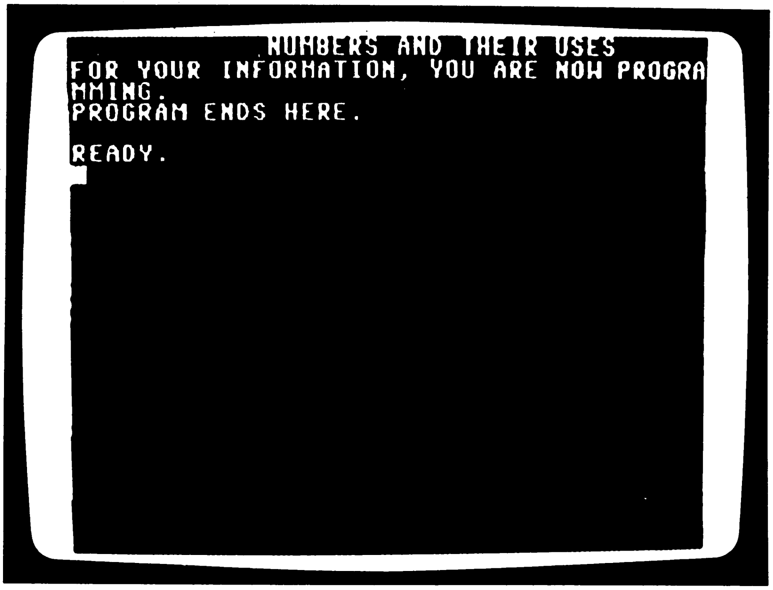
```
THIS LINE WILL BE REMOVED.
```

will be absent from the screen during a program run if you erase line 20, in which you ordered its printing. To erase a line from the computer's program memory, you can simply type the line number and press the RETURN key:

```
20
```

The computer stands ready to accept a new statement, and prepares to store a set of commands under the number 20 when you do this. When you send only the number, it replaces the former statement under this number with the new, blank statement. Since a blank statement gives no instruction, line 20 ceases to exist in the program memory.

When you RUN the program, you'll find the reverse-video text produced by line 20 absent. When you LIST the program, you'll find only lines 10, 30, and 40 remaining in memory, as illustrated in Figure 3.2.



```
                NUMBERS AND THEIR USES
FOR YOUR INFORMATION, YOU ARE NOW PROGRA
MMING.
PROGRAM ENDS HERE.

READY.
```

Figure 3.2: A program listing after one line has been deleted.

The computer follows this simple rule in handling program lines: The contents of a newly-sent numbered line replace the previous contents of the same number. You can add a new statement as line 20 like this:

```
20 PRINT "↑↑↑"
```

to include three arrows in the display after the line

```
NUMBERS AND THEIR USES
```

is printed.

The computer carries out numbered statements sequentially, according to any numbering scheme you devise (of positive, whole numbers up to 63,999). It will follow a program beginning at any number moving up the sequence of numbers, regardless of gaps in numbering between them. Numbering by tens is a handy approach to programming that leaves room between statements to add any program lines that occur to you as afterthoughts or refinements.

The computer retains numbered statement lines in its program memory until you switch off the power, or remove particular lines by typing their line numbers, or command the computer to erase them all. You can direct this erasure by sending the simple command:

```
NEW
```

On receiving this command, the computer prepares for a new program by erasing all program lines from its memory. If you write a new program and don't clear out the old program from memory, old program line numbers not replaced by new ones will tag along from the original program, giving directions within the new program you may not have planned.

STAND-INS FOR NUMBERS

Numbers, you've no doubt noticed, are as useful within commands as they are in marking program lines. As much as the commands themselves, numerical values help control the computer's operations. A change in the value of a number changes the action of the Commodore 64 in, for example, the directions given by a TAB command or the multiplication ordered with an asterisk (*) command.

A single number can be passed around within a program by the computer to be used in different commands, if you use a symbol that stands for the number you'd like acted on in the commands that act on it. If you instruct the computer early in the program that a symbol is standing in for a number, the computer will substitute the number at each command in which it subsequently encounters that symbol, and then act on it.

The handiest symbol to use as a stand-in for a number is a letter. You can use the letter in all commands where you would otherwise have used the number. For example, if you were considering text placement on a screen, you might put such a substitute in a TAB command within a program, like this:

```
20 PRINT "(press SHIFT and CLR/HOME)"
30 PRINT TAB(A) "HOW DOES THIS LOOK?"
```

The letter A substitutes for a number in the TAB command here. If you sent the RUN command to the computer for this two-line program without first assigning a value to A, it would follow built-in directions and replace the letter A with the value zero.

The Assignment Statement

The command by which you give a value to a symbol is known as the "assignment statement." It is represented by an equal sign (=), and to direct the computer to substitute a value of 12 for A in this program, you could send the command:

```
10 A=12
```

When you RUN this three-line program, the result will be the words "HOW DOES THIS LOOK?" printed beginning at column 12.

Having so named a number used in the program, you can change the operation of the program without changing the line that directs that operation. Change the value of A in line 10, like this:

```
10 A=20
```

and the computer will print at column 20. Used in this way, the letter A acts as a *variable* symbol that can stand for whatever number you specify with an equal sign.

You can use a variable in as many commands in your program as you want. If you add the command:

```
25 PRINT "COLUMN A"
```

to the program, the computer will respond when you RUN the program by printing

```
COLUMN 20
```

Then it will print the question starting at column 20 below that line. Again, if you change the value of A and then give the command RUN, the computer will position the printing accordingly, by column number:

```
10 A=5
```

A variable letter like A can be used any number of times in a program to stand in for a number. In this program you can use it to try out different screen-printing placements by retyping only the single short program line 10. In more complex programs, the effect of changing a variable's value by changing an assignment command can be profound.

The assignment command can be translated as a direction to the computer to work internally to replace the symbol on the left of the equal sign (wherever that symbol is encountered) with the numeric value on the right of the sign.

NUMERICAL PREDICTIONS

The computer can act on more than one symbolic variable in a program, if you include an assignment statement for each. This capability gives you enormous power in writing programs to analyze or predict various situations. If you can describe a situation with numbers, you can direct the computer to simulate the situation with any or all of its aspects changed. You can, for instance, compare the results of different salary offers, or the filling of reservoirs behind different dams.

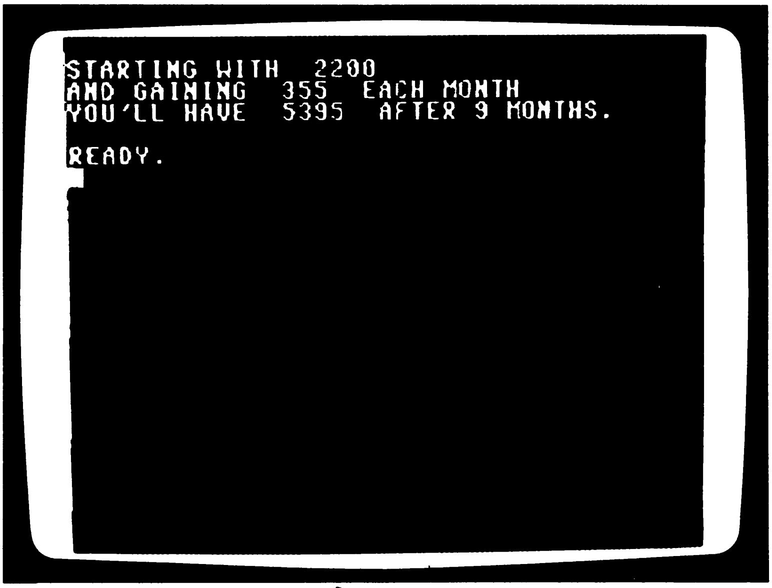
A prediction of the filling of a dam or your bank account could be made by forming a numerical model of the factors known to be at work. Then, sending values for these factors to the computer with assignment statements, you can see what the model predicts as a result. You can easily predict the amount of water in a reservoir or the amount of money in your account after, say, nine months, if you know the amount already there and the average monthly rate at which more accumulates. You can direct the computer to print these values and then to print the calculation of the total amount nine months later with the following program. (Be sure to send the NEW command to clear the last program from memory before typing this one.) In it, the original amount is symbolized by the letter A, and the monthly rate of accumulation by the letter B:

```
NEW
30 PRINT "(press SHIFT and CLR/HOME)"
40 PRINT "STARTING WITH" A
50 PRINT "AND GAINING" B "EACH MONTH"
60 PRINT "YOU'LL HAVE" A+(B*9) "AFTER 9 MONTHS."
```

If you command the computer to RUN this program now, it will automatically substitute zero for the variables A and B. But if you assign values for the amount present and the rate of accumulation, the computer will display those values and, through the directions in program line 60, print a prediction of the future amount. You can give values to the variables with the program lines:

```
10 A=2200
20 B=355
```

The computer will run the complete program with 2200 as the starting amount (gallons of water, or dollars) and 355 as the monthly increase, to produce the display shown in Figure 3.3.



```
STARTING WITH  2200
AND GAINING  355  EACH MONTH
YOU'LL HAVE  5395  AFTER 9 MONTHS.

READY.
```

Figure 3.3: The results of a prediction program.

You can, of course, change the simulation by changing the values assigned in lines 10 and 20, and so make predictions for different circumstances. This program was written to produce a prediction for nine months, but it can be altered by replacing the specific value 9 with a variable, through which you can produce a prediction for any number of months. To do this, you can modify line 60, which produces the display and calculation. You can assign the variable C to the number of months:

```
60 PRINT "YOU'LL HAVE" A+(B*C) "AFTER" C " MONTHS."
```

By adding another assignment command, you can specify another length of time, say 15 months:

```
25 C=15
```

When you run the program through the computer now, it will display the original amount of 2200, and predict an increase of 355 each month, over a period of 15 months.

You can direct the computer to substitute one variable letter for another, or for an "expression," consisting of variables and operators. In this program if you wanted a shorthand way of naming the final amount predicted, you could rename that amount, so far represented by the value of A + (B × C), in a separate statement, calling it F:

```
27 F=A+(B*C)
```

Now that the final value has a simple name, you can order the calculation and display of the difference between the original and final values, A and F, with this simple statement:

```
70 PRINT "THE GAIN IS" F-A
```

You could then streamline the program's appearance a bit with a substitution in line 60:

```
60 PRINT "YOU'LL HAVE" F "AFTER" C "MONTHS."
```

You can again change the details of this model by changing the values in lines 10, 20, and 25.

Although we have used only whole numbers so far, the computer is also designed to work with decimal numbers. It can take values stated in fine detail—for example, A = 2200.35. It can also handle fractions if they're treated as divisions. The value for 10 and 3/4 months, for example, would be stated as C = 10 + 3/4.

COMMANDS IN GROUPS

Commands work perfectly well one to a numbered program line. But you can also combine several together for convenience or for ease of planning.

The three assignment statements from the previous program:

```
10 A=220
20 B=355
25 C=15
```

can be grouped under a single program line number, if you prefer. If commands are separated from each other by colons, the computer will act on each command in a grouped set one at a time, as though each were on its own numbered line. You could replace those three lines with a single line, like this:

```
10 A=2200:B=355:C=15
```

And then erase the two now-redundant lines by sending empty line numbers with the RETURN key:

```
20
25
```

The new line number 10, holding a group of commands, then replaces the earlier lines 10, 20, and 25. The computer will hold the same information in its memory and carry out the program commands exactly the same way. You can group as many commands in a single program line as will fit in the 80 characters of space the computer recognizes for command entries.

Although both methods of grouping commands produce the same results when a program is run, you may prefer one to the other. Within a program written with one command to a statement line, you can more easily find problems that call for rewriting or alteration—the accidental misspelling of a command leading to a "?SYNTAX ERROR" reply during a program run, for example. On the other hand, grouping several commands to a line number may be an easier way to compose a program from multiple command statements. Either way, the choice is yours.

4 DECISION MAKING BY COMPUTER

USING JUDGEMENT

Like a vigilant sentry, you can watch over your computer, deciding when you would like certain commands repeated, or values changed, or when you'd like to interrupt a program run. But that means you're attending to the computer rather than having it attend to you.

Fortunately, such vigilance isn't necessary. Your decisions can be reflected within a program, so that when the computer encounters specific conditions you've described, it will take the action you've directed. The Commodore 64 can examine a numeric comparison (that is, a statement that one value is equal or not equal to, or greater or less than, another value), and "evaluate" it as either true or false. If the statement is true, the instruction that immediately follows will be executed. If not, the computer will skip over that instruction and proceed to the next numbered line.

The command that allows this decision-making is the IF-THEN statement, which takes much the same form as the "if . . . then" conditional statement in everyday English: "If it's raining now, then go by car." *If* the statement is true, *then* take the action named. (If not, then don't.) The computer command looks like this:

```
IF (comparison) THEN (instruction)
```

We'll see some specific examples along the way.

You can use the IF-THEN command in a new program to evaluate a sum of three numbers represented in the following way:

```
10 A=5: B=8: C=87
```

```
20 T=A+B+C
```

You can then arrange for the computer to display a statement if the sum is 100, with the command:

```
30 IF T=100 THEN PRINT "A SUM OF A HUNDRED"
```

which will instruct the computer to print the statement only when the condition T = 100 occurs.

When this program is run through the Commodore 64, the quotation will be printed. If line 10 is changed so that the values add to something else, the computer will skip past THEN and its PRINT command, in this case to the program's end.

On encountering an IF-THEN statement, the computer follows built-in directions to evaluate the condition stated to the right of the word IF. When that condition is true, the computer follows the command to the right of the word THEN. When it is false, the computer skips to the next program line.

You can direct the computer in this decision-making process through any mathematical relation it is built to understand. The following change to the program, for instance, tests the numbers of guests arriving in groups at a gathering:

```
30 IF T>100 THEN PRINT "THE ROOM IS GOING TO BE
   CROWDED"
```

Here, the test is not whether T is *equal to* a particular value, but whether it is *greater than* the value.

When it encounters a number after the word THEN, the computer translates the number as a direction to carry on with the program from the commands in that line number. Thus, besides instantly following a command in the IF-THEN statement, the computer can be sent to another part of the program by line number, as in the following program:

```
NEW
 10 A =20: B=15: C=9
 20 T= A+B+C
 30 PRINT "(press SHIFT and CLR/HOME) WITH GROUPS
    OF"A"AND"B"AND"C"THE ROOM WILL HAVE"T
    "PEOPLE IN IT"
 40 IF T>75 THEN 100
 50 PRINT "(press CURSOR UP/DOWN) NO PROBLEM
    ACCOMMODATING THAT MANY "
 60 END
```

```
100 PRINT "THAT'S MORE THAN THE ROOM WILL
    HOLD!"
```

Using the values assigned in line 10, the computer will print the quotations in lines 30 and 50, then stop carrying out commands on reaching the END statement. It will skip past the command that would redirect it to line 100 from line 40: THEN 100. If you change line 10 so that the sum exceeds 75:

```
10 A = 19: B = 23: C = 37
```

and run the program again, the computer will carry out the command after THEN and skip past lines 50 and 60 to carry out the PRINT statement of line 100.

To direct the Commodore 64 to a line number from an IF-THEN statement, you can also include the optional command word GOTO. The following revision of line 40:

```
40 IF T > 75 THEN GOTO 100
```

produces the same results as its predecessor, but may be a little easier to read and understand.

In directing the computer forward through a program according to values it encounters, or directing it to some immediate command within an IF-THEN statement, you can change the way the computer executes a program. By directing the computer backward to an earlier line number from an IF-THEN statement, you can make it repeat earlier commands, with variations if you like.

SIMULATIONS

An IF-THEN statement with a line number is like a finger pointing backward or forward. This ability to point to different courses of action can be especially useful in simulating different scenarios. You can apply it, for instance, to the problem of filling a party room with guests arriving in groups without exceeding the room's capacity.

In a situation like this, there are three values to consider: the capacity of the room, which you can call the variable C; the number of guests at the party, which you can call G; and the rate at which new guests arrive each trip—call this R. If you want to know how many trips will fill, but not crowd, the party, you can use an IF-THEN statement to instruct the computer to consider trip after trip, and then signal you when the simulated trip that fills the room finally occurs.

You can begin this new program with a statement defining the room size, the number of guests at the simulation's beginning, and the rate of new arrivals:

```
NEW
10 C=200: G=20: R=6
```

The computer will prepare the screen and display the available room from these commands:

```
20 PRINT "(press SHIFT and CLR/HOME)".
30 PRINT "AFTER" N"TRIPS" G "PRESENT & ROOM FOR" C-G
```

In line 30, the variable N will serve as a counter for the number of trips through which the program has directed the computer. You make N a counter and keep G, the number of guests, up to date with these commands:

```
40 N=N+1: G=G+R
```

The crucial command is the one that directs the computer either to continue or to stop considering trips:

```
50 IF G<C THEN 30
```

This line instructs the computer to either return to the sequence at line 30 (if G is less than C) or continue to the final line, which signals you that a limit has been reached:

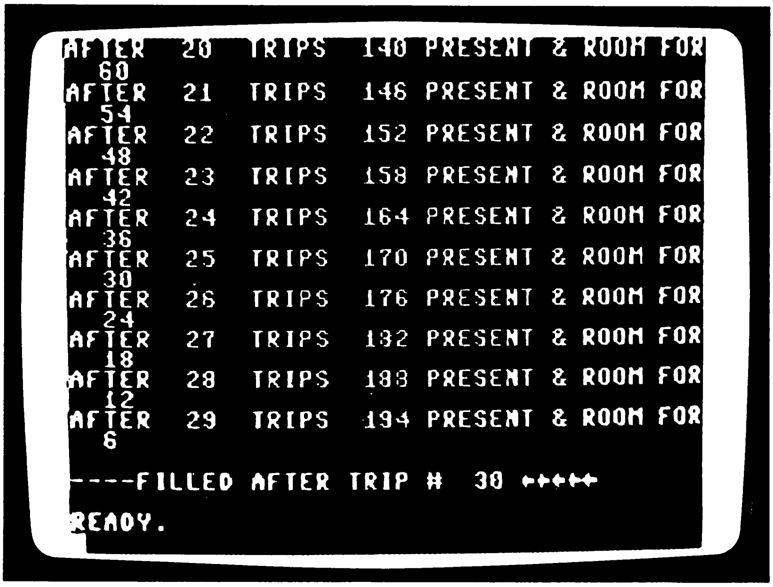
```
60 PRINT:PRINT "----- FILLED AFTER TRIP" N "←←←←←"
```

When you run it, this program will direct the computer to repeat the quotation in line 30 with different values until the room's limit is reached. Figure 4.1 shows the steps that the program run takes.

First, the computer stores the values for C, G, and R (stated in line 10) in its memory, then clears the screen and positions the cursor for the PRINT statement in line 30. When that statement is first encountered, the value of N has been automatically set at zero. This represents the condition of the room before the first trip of newcomers:

```
AFTER 0 TRIPS 20 PRESENT & ROOM FOR 180
```

Encountering line 40, the computer considers trip number 1, which increases the number of guests to 26. Then, at line 50, it compares the number of guests to the room's capacity and, finding on this first pass that the number is smaller, it follows the IF-THEN direction to line 30. Acting



```
AFTER  20  TRIPS  140 PRESENT & ROOM FOR
 60
AFTER  21  TRIPS  146 PRESENT & ROOM FOR
 54
AFTER  22  TRIPS  152 PRESENT & ROOM FOR
 48
AFTER  23  TRIPS  158 PRESENT & ROOM FOR
 42
AFTER  24  TRIPS  164 PRESENT & ROOM FOR
 36
AFTER  25  TRIPS  170 PRESENT & ROOM FOR
 30
AFTER  26  TRIPS  176 PRESENT & ROOM FOR
 24
AFTER  27  TRIPS  182 PRESENT & ROOM FOR
 18
AFTER  28  TRIPS  188 PRESENT & ROOM FOR
 12
AFTER  29  TRIPS  194 PRESENT & ROOM FOR
 6

----FILLED AFTER TRIP #  30 ←←←←←
READY.
```

Figure 4.1: The outcome of a simulation program.

again on line 30, the computer displays the values of N and G after that first trip:

```
AFTER 1 TRIPS 26 PRESENT & ROOM FOR 174
```

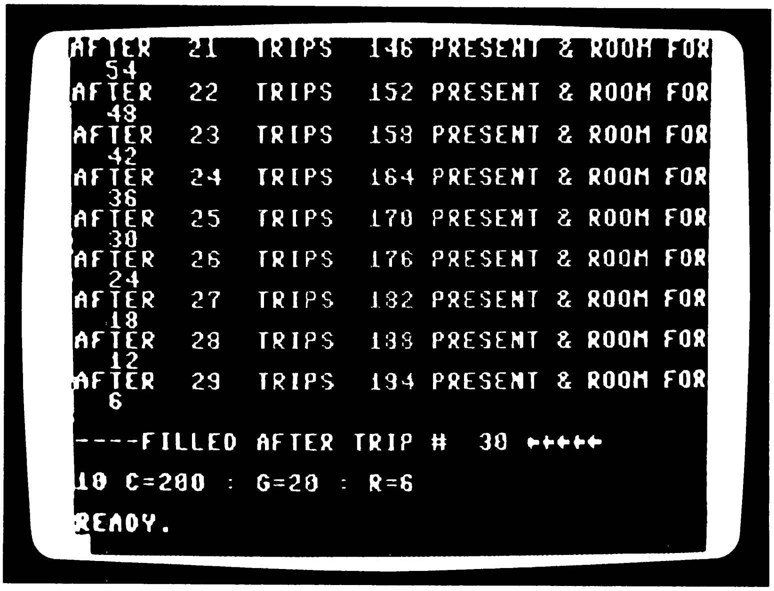
Blind to its poor grammar, the computer goes on to consider the second trip at line 40, and repeats the process. It prints the same statement at each trip until the pass at which the number of guests reaches the capacity, when N is 30. On encountering the IF-THEN command at that point, the computer advances to line 60, printing the message that the room would be filled after that trip.

You can add a handy feature to this program by directing it to display the conditions you stated in line 10:

```
70 LIST 10
```

This command will print line 10, as in Figure 4.2. You can now retype the line to reflect a larger room, C, a different number of guests already present, G, or a different rate of arrival, R.

Under the direction of an IF-THEN command sending it backward, the computer conjures its repetitive powers. Acting under other commands, the computer can be sent in any direction to yet other commands,



```
AFTER  21  TRIPS  146 PRESENT & ROOM FOR
 54
AFTER  22  TRIPS  152 PRESENT & ROOM FOR
 48
AFTER  23  TRIPS  158 PRESENT & ROOM FOR
 42
AFTER  24  TRIPS  164 PRESENT & ROOM FOR
 36
AFTER  25  TRIPS  170 PRESENT & ROOM FOR
 30
AFTER  26  TRIPS  176 PRESENT & ROOM FOR
 24
AFTER  27  TRIPS  182 PRESENT & ROOM FOR
 18
AFTER  28  TRIPS  188 PRESENT & ROOM FOR
 12
AFTER  29  TRIPS  194 PRESENT & ROOM FOR
 6

----FILLED AFTER TRIP #  30 ←←←←←

10 C=200 : G=20 : R=6

READY.
```

Figure 4.2: A simulation program that displays the conditions which directed it.

from one program line to another. Complex routes can be formed between program statements, each of which asks the computer to do something different.

5 PROGRAM CONTROL

A servile machine, the computer can accept and obey scores of commands. It will act on them in its own dull-witted but meticulous way. With these commands, you can put the machine to work on tasks you might otherwise find too tedious or repetitive, too nitpicking or precise, too long or time-consuming. When so commanded in a language it recognizes, your computer can repeat a given task endlessly or as often as you say. It can handle words or numbers, tearing them apart or delicately assembling them, it can carry out the most cumbersome calculations, or it can quickly produce screen displays of a complexity that would otherwise be possible only by days of hand-work.

A PROGRAMMER'S KIT OF COMMANDS

Human beings are imperfect: we change our minds, we make mistakes, and we are prone to forget. The programs we devise are subject to our interesting and imperfect nature, and allow for this nature, you can mark, stop, and restart the programs you use. You can borrow useful statements of commands from other programs, and tinker them into the shape you want.

REM: A Programmer's Notebook

If you put together many programs, you'll find handy a command that marks program lines without affecting the computer's operation. That command is REM, which stands for "remark." You can appreciate its usefulness by considering the following graphics program, which directs the drawing of a checkered pattern like that shown in Figure 5.1. The pattern is made by first drawing vertical lines in white in even-numbered columns (in program lines 20 through 60), and then slicing through them with horizontal lines of black (in lines 70 through 120). Notice that line 70 uses the CLR/HOME key without SHIFT, to send the cursor "home" without clearing the screen.

```
NEW
 10 PRINT "(press SHIFT and CLR/HOME)"
 20 A=0
 30 PRINT TAB(A) "(press C= and +)(press SHIFT and CURSOR
    UP/DOWN)"
 40 A=A+2 : IF A<25 THEN 30
 50 PRINT
 60 B=B+1 : IF B<20 THEN 20
 70 PRINT "(press CLR/HOME)"
 80 M=0
 90 PRINT TAB(M) "(press SPACE BAR)(press SHIFT and CURSOR
    UP/DOWN)"
100 M=M+1: IF M<25 THEN 90
110 PRINT "(press CURSOR UP/DOWN)"
120 V=V+2 : IF V<20 THEN 80
```

What this program does is not immediately obvious to anyone reading it. Instead of running the program through the computer each time you wish to see what it does, you can identify it with a labeling "remark" line, which, as a note to whoever reads the program, is not acted on by the computer. To so mark this program, add a line that starts with REM:

```
5 REM A PROGRAM FOR THE COMMODORE 64 THAT MAKES A
  CHECKERED PATTERN.
```

When the computer encounters this REM line in a program, it takes the command REM to mean: "Move on to the next program line; what follows is a remark for the person reading this program." Although a program may make perfect sense to you when you type it, a few weeks later—or sometimes a shocking few minutes later—it may seem nothing but a baffling assortment of commands unless you've tagged it with a note.

Since the REM command can be put anywhere in a program, you can describe sections or individual command lines of a program if you like. For instance, in the case of the checkered program, you can add notes before the statements that direct the computer to produce the vertical lines,

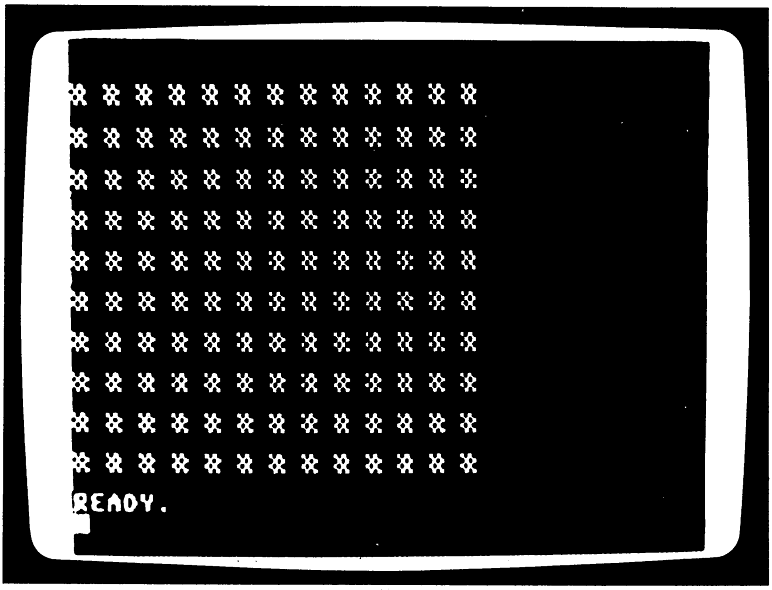


Figure 5.1: A checkered pattern produced by a program.

and also before the statements that produce the horizontal lines:

```
15 REM THE NEXT FIVE PROGRAM STATEMENTS DRAW
   VERTICAL LINES IN CHARACTER COLOR
65 REM THE NEXT SIX PROGRAM STATEMENTS DRAW
   HORIZONTAL LINES IN BACKGROUND COLOR
```

A program marked in this way will be easier to identify and alter, later. REM statements are subject to the same 80-character length limit as other program statements; line 65 is as long as a REM line can be.

Interrupting a Program Run

While this program is running, you might decide you'd like a slightly different pattern. To stop the program in mid-stride, you can use any of three commands. The planned way is to include a command to that effect within the program itself. The program command that will interrupt a run is STOP. You can strategically include it to stop the program run after the vertical lines have been drawn by program statements 20 through 60, like this:

```
65 STOP
```

If you run the program with this additional line, the computer will stop following program commands on encountering the statement, and will inform you that a forced break in the run has been ordered, with the reply:

```
BREAK IN 65
```

The STOP command differs from the END command only in that the computer replies with a message. STOP can be paraphrased as: "Print BREAK IN and this line number, and wait for directions from the keyboard."

An advantage of the STOP command is that you can, at your convenience, command the computer to proceed with the program run, by sending the command:

```
CONT
```

which can be paraphrased "Continue carrying out commands at the program line following the interruption." Of course, if the STOP command is removed from a program (here by removing line 65), the program will run straight through without pause.

The second way to stop a running program is directly from the keyboard, with the command that mimics the STOP command in an immediate, unplanned way. To stop a program at any point in its operation, press the RUN STOP key. After receiving this keystroke command, the computer will reply with the "BREAK IN" messsage. Once stopped in a program run with the RUN STOP command, the computer will also resume carrying out program commands when sent the CONT command. The STOP command, implanted in a program, and the RUN STOP command, sent from the keyboard, both affect the computer's operation in the same way.

You can use either one to stop the computer at line 50 of the current program, just after the bottom edges of the vertical lines have been drawn, to consider varying the pattern. One way to change the display is to select another character or color for the rows drawn by lines 70 through 120. Even a black-and-white screen will show changes in shadings among colors.

Now that you've interrupted the program... Interrupting a program can sometimes give you ideas for interesting variations. For example, you can change program line 120 so as to draw columns of spaces only as far down as row 18:

```
120 V=V+2:IF V<18 THEN 80
```

You can create horizontal rows of circles with this revision:

```
90 PRINT TAB(M) "(press SHIFT and W) (press SHIFT and CURSOR
   UP/DOWN)"
```

Of course, you can keep revising until the pattern is one you like. Perhaps this revision of program lines 80 and 100 will add the finishing touch:

```
 80 M=5
100 M=M+2:IF M<20 THEN 90
```

These three changes will produce a program that displays the pattern of lines and circles shown in Figure 5.2 when run.

The RUN STOP-RESTORE Command

The third and most intrusive way of stopping a running program directs the computer to leave the program completely. The program is retained in memory, but no "marker" is left in the run. To stop a program run with this emergency-brake command, which operates from the keyboard, you press the RUN STOP and RESTORE keys together. Since the computer forgets its place in the program, the CONT command isn't a useful way to pick up the sequence again in this case.

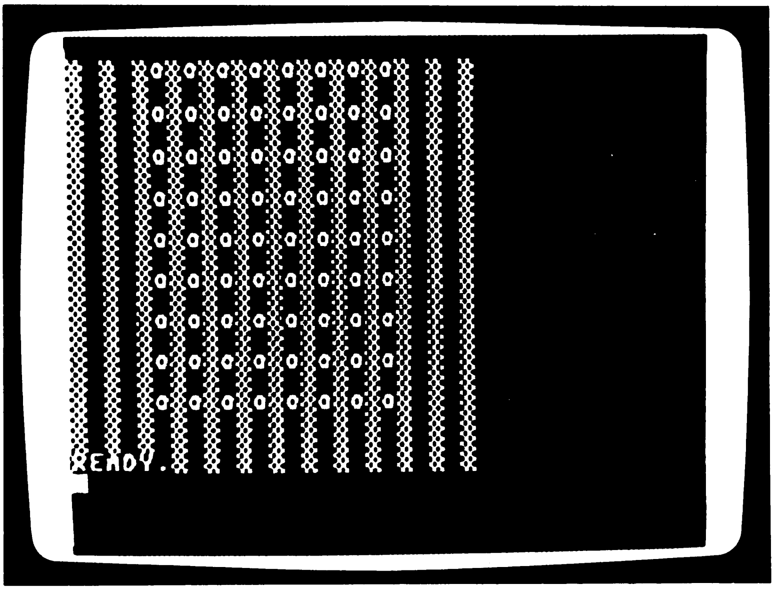


Figure 5.2: The result of revising the program that produced Figure 5.1.

You can usually slow a program, or any response of the computer, by pressing the CTRL key and holding it down. When you release it the computer returns to normal speed.

We've now seen the full set of commands you have for restructuring existing programs in the BASIC language. There are more commands for building the programs themselves.

COMMANDS FOR MOVING AMONG STATEMENTS

The GOTO Command

We've seen some commands that direct the Commodore 64 outside its usual course through higher line numbers in a program; there are still more. One, GOTO, is even simpler in action than the IF-THEN command. We've already used it in a version of that command, IF-THEN-GOTO. You can also use the GOTO command by itself, to direct the computer back to some earlier statement in a program, again and again, creating an unending *loop* of computer action. Ideal for repetitious tasks, this sort of loop can be stopped only with one of the three program-stopping methods described in the previous section.

The following program will, if you let it run, command the computer to produce an "endless" multiplication and division table. (Actually, you would eventually reach a number greater than the Commodore 64 can handle, and get an OVERFLOW ERROR message.) In line 10 the number to be multiplied and divided is set at five.

```
NEW
10 A=5
20 PRINT "(press SHIFT and HOME) ------------THE NUMBER"
   A " -----------"
30 PRINT "N" TAB(5) "MULTIPLIED BY N" TAB(22) "DIVIDED BY N"
40 PRINT
50 N=N+1
60 PRINT N TAB(10) A*N TAB(25) A/N
70 GOTO 40
```

The final command, in line 70, directs the repetition of the program from line 40 on. Without that GOTO direction, the program would simply run three rows of printing—the heading from line 20, the subheading from line 30 and the first pass at the counter number N (when it's raised from zero to one), the multiplication of A × N and division of A/N from line 60. When the computer finishes with the commands of line 60 and translates the command GOTO 40 on line 70, however, it jumps back into the sequence at line 40. Thereafter it automatically follows increasing line numbers until it again encounters line 70, which directs it out of sequence to line 40 again. Each time the computer performs line 50 (N = N + 1), the "counter," a new value is applied to the printing, multiplication and division of line 60.

The untiring computer will perform the commands between program lines 40 and 70 again and again until you stop it with a RUN STOP or RUN STOP-RESTORE command from the keyboard (or, of course, switch off power).

Although you can also direct the computer to skip forward through the program to a higher-numbered line, there's no advantage in doing so, since the commands of intervening statements would be missed forever. While a second GOTO command could send the computer back to an earlier line that had been skipped, that method of programming offers no control that couldn't be achieved more simply by a single GOTO command controlling the type of repeating loop seen above.

In this program, you can assign any value to A in line 10, and then observe the resulting calculated values as they appear on the screen if you're looking for a particular value. If you'd like to know when, for instance, the division drops below some value, a variation of the GOTO command can direct the computer to do that looking for you. This command is fairly rigidly structured, but easy to use once you understand its format. Through the ON-GOTO command, the computer is directed to compare variable symbols with a built-in number range, then is sent to a program line. It works much like an IF-THEN-GOTO statement, except that the condition in the first half of the command is preset to numerical ranges: 1 or greater but less than 2, 2 or greater but less than 3, and so on. The directions in the GOTO half of the command point to line numbers you arrange according to those preset ranges.

If you wanted the computer to signal you when the value of the division A/N, for instance, falls below two, three, and four you could add a command to do that after line 60, like this:

```
65 ON A/N GOTO 100,200,300
```

which directs the computer to evaluate A/N, compare it with the built-in ranges and then to proceed with the appropriate program line—100, 200, or 300—according to that value. In this case, you can put PRINT statements starting at program lines 100, 200, and 300 to alert you to the values produced by earlier lines of the program and A/N. If you had no interest in values less than two, for instance, you could give the following commands to signal and stop the program:

```
100 PRINT TAB(20) "VALUE BELOW TWO"
110 STOP
```

which will print "VALUE BELOW TWO" in the A/N column, and then stop the computer with the message "BREAK IN 110."

Likewise, if you wanted the computer to continue the program run and merely signal you when the value A/N slipped below four and then again below three, you could add statements starting at 300 and 200, respectively:

```
300 PRINT TAB(20) "(press CONTROL and G)(press CONTROL and
    G) VALUE BELOW FOUR"
310 GOTO 40
```

and

```
200 PRINT TAB(20) "(press CONTROL and G)
    (press CONTROL and G) VALUE BELOW THREE"
210 GOTO 40
```

During the execution of this program, each of these pairs of statements signals the appearance of the value you're looking for, then directs the computer back into the multiplication-division-printing loop at line 40.

You can use the ON-GOTO command to search out and act on specific values. By putting variables into a form that falls within the rather particular value ranges of the first half of this command, you can direct the computer into diversions and exits from normal sequential program runs.

You can specify action on any number of ranges. Directions to line numbers, each matched to a particular range, can be used to send the computer to work on many more statements. The ON-GOTO command follows this plan: each line number listed after the word GOTO corresponds in step to progressively rising ranges of the value of the variable after the word ON.

You could use a statement like:

```
ON A/N GOTO 100, 200, 300, 400, 500
```

and so on, to cover larger and larger values of A/N.

REPEATING WITHIN LIMITS

You may want to repeat a useful set of commands, perhaps not endlessly (as done by the GOTO statement), nor conditionally (as done by the IF-THEN statement), but simply for a specific number of times. The Commodore 64 will respond to a pair of commands that do this. One counts off the repetitions, the other directs the computer back to the first command. These commands are FOR and NEXT. You can slip them in, one before a stretch of statements you'd like repeated, the other after.

Here's a program to which you can add this pair of commands in a variety of ways to produce a variety of displays. It directs the computer to draw an arrow shape.

```
NEW
10 PRINT "(press SHIFT and CLR/HOME)"
20 X=5
30 PRINT TAB(X)"        (press SHIFT and M) "
40 PRINT TAB(X)"--------(press SHIFT and Z)"
50 PRINT TAB(X)"        (press SHIFT and N) "
```

Through the value you type for X as a column number in line 20, you can locate the arrow drawn by the directions in lines 30, 40 and 50 in any horizontal position. Now, to draw the arrow several times at different places, the computer needs directions to repeat lines 30, 40, and 50 for different values of X, so that the same pattern is plotted from differing starting points.

You can give these directions with the first statement of the counting pair. In this statement you state a starting value, an ending value, and (optionally) the size of the jumps the computer should make in going from one value to the other. In this graphics program, you can specify a column number represented by X, to be changed at each pass so that the arrow will be drawn at each of several places along a single row, spaced 8 columns apart, with the statement:

```
25 FOR X=1 TO 30 STEP 8
```

The other member of this command pair will automatically direct the computer back to the program line on which the FOR-TO-STEP command is found:

```
55 NEXT X
```

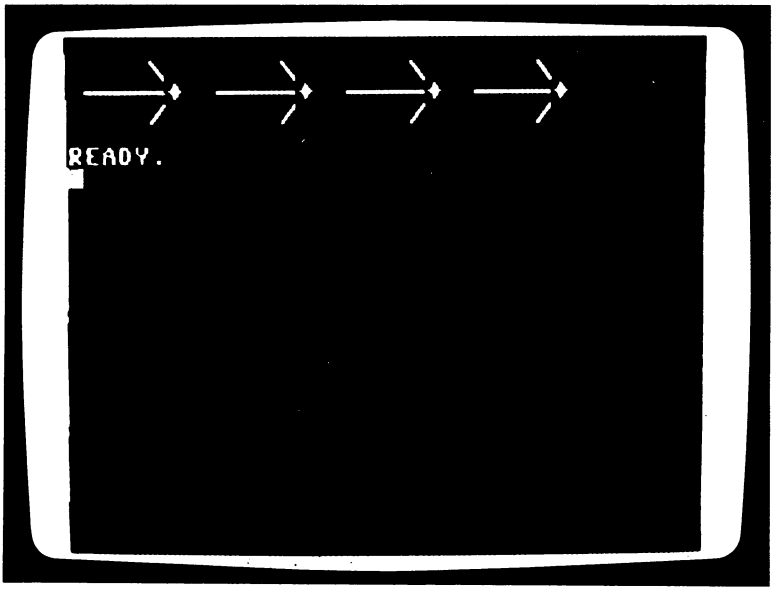
To direct the computer to put each subsequent arrow on the same horizontal line as the one previously drawn, you can add a statement that sends the cursor back up to the line on which it began. Such a statement would take the form:

```
27 PRINT "(press SHIFT and CURSOR UP/DOWN) (press SHIFT and
   CURSOR UP/DOWN) (press SHIFT and CURSOR UP/DOWN)
   (press SHIFT and CURSOR UP/DOWN)"
```

When you run this program, all commands located between the FOR-TO-STEP and NEXT statements at lines 25 and 55 will be repeated until the limit set after the word TO is reached, as in Figure 5.3.

In this case, lines 30, 40, and 50 (the drawing commands), and line 27 (the row-positioning command), are repeated. The computer will begin drawing at each of four locations along a single row at the top of the screen. The first arrow starts from the position set by a TAB of 1, and the last at a TAB of 25. The value of X = 1 assigned in the FOR-TO-STEP statement replaces the initial one from line 20, and each successive value of X replaces the one before it.

Using the FOR-TO-STEP, NEXT command pair, now you have a program that draws a row of arrows. With another pair of FOR-TO-STEP, NEXT commands ranging over values of Y, you can expand the program to draw a similar group of arrows at more rows, and to fill the screen with them. To repeat the first row of arrows, the second pair of commands should bracket the statements that draw the row, including the FOR-TO-STEP and NEXT pair for the values of X. Thus, the arrows at



```
READY.
```

Figure 5.3: A row of repeated arrows drawn by program.

each X location can be drawn for different Y locations (rows). To direct the computer down to draw another row you can use a command that moves the cursor like this:

```
24 PRINT "(press CURSOR UP/DOWN) (press CURSOR UP/DOWN)
   (press CURSOR UP/ DOWN) (press CURSOR UP/DOWN)"
```

And to direct the computer to repeat the drawing at four descending rows, you can add another pair of repeating commands, like this:

```
22 FOR Y = 1 TO 4
60 NEXT Y
```

When it encounters a FOR-TO command without the STEP part, the computer assumes a "default" STEP value of 1. The program now in memory directs the computer to repeat the commands that produced the first row of arrows, to measure down four lines from the first row and draw a row of arrows, then move down another four lines and draw another row of arrows, and so on until a fourth row has been reached. Then the computer stops drawing, and four rows of four arrows each fill the screen, as shown in Figure 5.4.

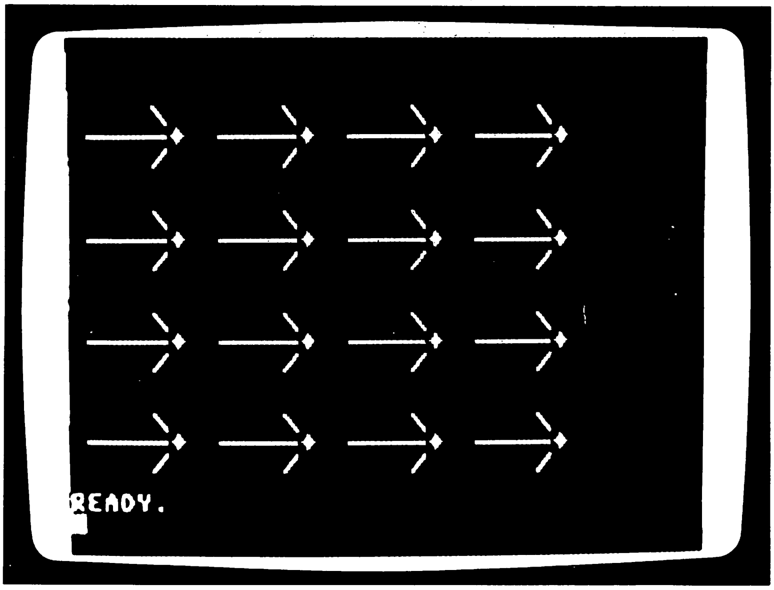


Figure 5.4: Filling the screen with arrows by altering the program of Figure 5.3.

Since the effect of line 20 (which assigned a value to X) has been cancelled by line 25, you can drop that statement as excess baggage:

```
20
```

(If you LIST the program now, you'll see the nested pairs of FOR-TO-STEP and NEXT commands as they appear in Figure 5.5.) The first NEXT statement, at line 55, directs the computer to take action on line 25, where the value of X is changed.

The computer can actually distinguish one NEXT statement's directions from another's even if you don't label each NEXT statement with the variable from its FOR-TO-STEP line; lines 55 and 60 could also have been written as:

```
55 NEXT
60 NEXT
```

However, you can keep track of the FOR-TO-STEP, NEXT groupings more easily if you do label them.

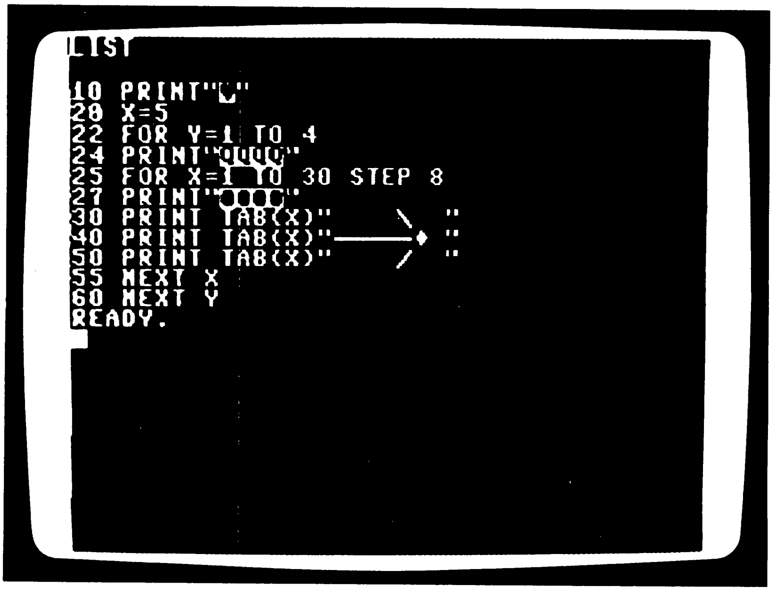
You can envision the X FOR-TO-STEP, NEXT loop as "nested" inside the Y FOR-TO-STEP, NEXT loop. When one loop is nested within another in a program, the commands in the inner loop are carried out first, as many times as detailed in the FOR-TO-STEP command, at the first value of the outer loop, then through the second value of the outer loop, and so on.

Here's a summary of how the computer is directed by this program:

In line 10, the screen is cleared and the cursor is positioned at the upper left corner. In line 22, the computer prepares to substitute a value of 1 for each Y it encounters. In line 24, the cursor is sent down four lines in preparation for drawing. In line 25 the computer prepares to substitute the value 1 for each X. In line 27 the cursor is moved up four vertical rows. The effects of the vertical movement in program lines 24 and 27 come particularly into play in positioning arrows drawn one after another. In lines 30, 40, and 50, the substitutions are made, and the resulting print characters are drawn from the TAB(1) location. That produces the first arrow on the screen.

From line 55, the computer is sent back to the FOR-TO-STEP statement of line 25, where the value 8 is added to the last X, and the new value of X is substituted in lines 30, 40, and 50. The computer draws the second arrow from column number 9 of the first row.

When it next encounters line 55, the computer is sent again to the FOR-TO-STEP statement in line 25, and the process repeats with a value



```
LIST

10 PRINT"♥"
20 X=5
22 FOR Y=1 TO 4
24 PRINT"QQQQ"
25 FOR X=1 TO 30 STEP 8
27 PRINT"QQQQ"
30 PRINT TAB(X)"        \  "
40 PRINT TAB(X)"———————◆  "
50 PRINT TAB(X)"        /  "
55 NEXT X
60 NEXT Y
READY.
```

Figure 5.5: The program that fills the screen with arrows.

of X equal to 17 substituted in the TAB commands.

This goes on until the value of 30 for X is passed. That happens after the fourth arrow is drawn, when the NEXT X statement of line 55 passes the computer on to the following line, number 60. Line 60 sends the computer to the FOR-TO-STEP statement at line 22, where 1 is added to the value of Y. With Y = 2, the computer then moves to line 24, which sends the cursor down four rows.

Then, encountering the FOR-TO-STEP statement of line 25, the computer resets the value of X to 1, and proceeds to line 27. This statement keeps each set of four arrows of varying TAB(X) values on the same row. Without it, the computer would automatically advance to the next print line after drawing each arrow at each TAB(X) position, and each arrow would appear separately on a different row.

Running through the commands of lines 25 through 55 as it did before, the computer draws a second row of arrows, identical to the first, but below it. After the last arrow of this second row has been drawn and X exceeds 30, the computer is sent on to the NEXT statement, dropping through line 55 to line 60, which sends it to line 22.

The FOR-TO-STEP, NEXT Y statements of line 22 and 60 then keep the computer busy on the third, and finally the fourth, rows of arrows until the X and Y values (30 and 4) have both been passed. Thus, you are left with a screen full of four rows of four arrows each.

As you might suspect, if two nested loops of FOR-TO-STEP, NEXT statements can be heaped together into the computer, then three or more are also possible. With the Commodore 64's print graphics, you can use this feature to create moving images of changing colors.

In any program, a loop controls all the statements contained within that loop, including any number of other loops. If commands are added that print blank spaces over an arrow just before another arrow is drawn on the screen, the first arrow will appear to be erased, and the second arrow will appear by itself until it, too, is drawn over, disappearing from the screen to be replaced by the next. The result is an elementary form of animation.

To direct the computer to draw over each arrow in turn, you can add a set of three statements, within the X FOR-TO-STEP loop, that print spaces over each arrow, like this:

```
52 PRINT TAB(X) "(press SPACE BAR 8 times)"
53 PRINT TAB(X) "(press SPACE BAR 8 times)"
54 PRINT TAB(X) "(press SPACE BAR 8 times)"
```

To position the blank printing back on the same row on which the arrow was drawn, you can add a cursor-positioning statement before it, like this:

```
51 PRINT "(press SHIFT and CURSOR UP/DOWN 3 times)"
```

When you run the program now, the results will be as before, except that immediately after each arrow appears it will disappear and the next arrow will be drawn. The result will be an arrow racing from left to right across one row after another.

Another clever use of the FOR/NEXT loop is worth noting, as it adds a delay, which can be useful in any program in which you would like to allow time for someone to view the program's effects. You might have already noticed that the computer, incomprehensibly fast though it is, does take a perceptible time to do things. The more complex or complicated the task, the longer it takes. Knowing this, you can use such a delay to your advantage, by controlling it. You can add a loop containing no real active command. And though there may be no command to repeat in such a loop, the computer will go dutifully through it, taking time to translate and process the FOR-TO-STEP and NEXT commands.

To slow the animation program, then, you can add an empty loop (known as a "timing loop") after the blank-space-positioning line, like this:

```
51 PRINT "(press SHIFT and CURSOR UP/DOWN) (press SHIFT and
CURSOR UP/DOWN) (press SHIFT and CURSOR UP/DOWN)
(press SHIFT and CURSOR UP/DOWN)" : FOR T = 1 TO 200:
NEXT T
```

Once again, the STEP part of the command is optional—if you leave it off, as is done here, the computer automatically takes steps of 1.

In this program, the computer pauses briefly before each erasure as it runs senselessly through its empty loop. This type of FOR-TO, NEXT statement can be inserted anywhere in a program to slow or freeze advancing action.

By judiciously removing statements, you can sometimes create effects quite different from the original program. For instance, by removing the Y FOR-TO, NEXT loop and the accompanying positioning command from this program, you can create a display that shows an arrow descending diagonally across the screen. Removing lines 22, 27,and 60, like this:

```
22
27
60
```

will produce such a program.

6 HANDLING WORDS AND INFORMATION

By its nature, the computer understands how to handle numbers—they direct all its operations, and order commands. But even though it is blind to their meanings, the Commodore 64—with a bit of programming help—can handle words and information in any typed form, treating and even generating them at your direction. Your computer can accept information in several ways, and even stop in the middle of a program run to get it. Words, numbers and graphics symbols can all be objects of manipulations you direct.

STRINGS: WORDS FROM THE COMPUTER

In much the same way as it handles numbers, the Commodore 64 can handle words and keyboard characters. It can print them, and it can take them apart and piece them together. Of course, as human beings, we have an edge over the computer, since we each understand about 40,000 more words than the Commodore 64's meager vocabulary of less than a hundred commands. But, like a blind newstand vendor who can handle magazines he can't read, the Commodore 64 can handle thousands of words it will never understand.

Although the computer can treat keyboard characters—words, numbers, and other symbols—in various complex ways, it groups these symbols into only two categories. When the computer encounters a set of words and symbols enclosed in quotation marks, it treats that group of characters like a busload of passengers, to be carried about as you, the tour leader, direct. But characters typed without surrounding quotes are interpreted as commands, from which the computer takes directions, just as a cab driver picks up passengers from a street corner and takes directions from them.

A group of characters enclosed in quotes is known as a *string.* Variable symbols can be used to stand in for strings, just as they do for numbers. The Commodore 64 recognizes the equal sign as a command assigning a variable to words as well as to numbers. It recognizes these strings by a "tagged" variable symbol. A variable intended for a string is marked with a dollar sign ($), signifying a string of characters, rather than a numeric variable. Since the computer treats spaces as characters, a group of words separated by spaces will be treated as one long word-string by the Commodore 64.

You can command the computer to substitute a sentence (or any group of characters) wherever it encounters a string variable in a program statement:

```
NEW
10 A$ = "WORDS WITHOUT THOUGHTS NEVER TO
   HEAVEN GO."
```

You can then use the string variable, A$, in place of the string itself when you give subsequent commands:

```
20 PRINT A$
```

Encountering line 20 when this brief program is run, the computer prints Shakespeare's words on the screen, just as it would print a number assigned to a numeric variable.

Combining Strings: The Plus Sign

Although strings possess none of the numeric characteristics that the computer was built to work with, they can nevertheless be objects of calculation and manipulation. Using string variables, you can order the computer to combine strings, cut them or derive numeric values from them.

The computer treats string variables in combination much as it treats numbers in addition. In this case, the final result will be a longer string rather than a larger number. To add strings to each other, you use the plus sign (+) as a command to the computer to produce a new string, a combination of the strings on either side of the plus sign.

Adding two more strings to this program, you can prepare the computer to combine them, by assigning them to string variables:

```
12 B$ = "WILLIAM SHAKESPEARE"
14 C$ = "HAMLET, ACT III, SCENE 3"
```

To see these strings printed in combination, you can retype line 20 as:

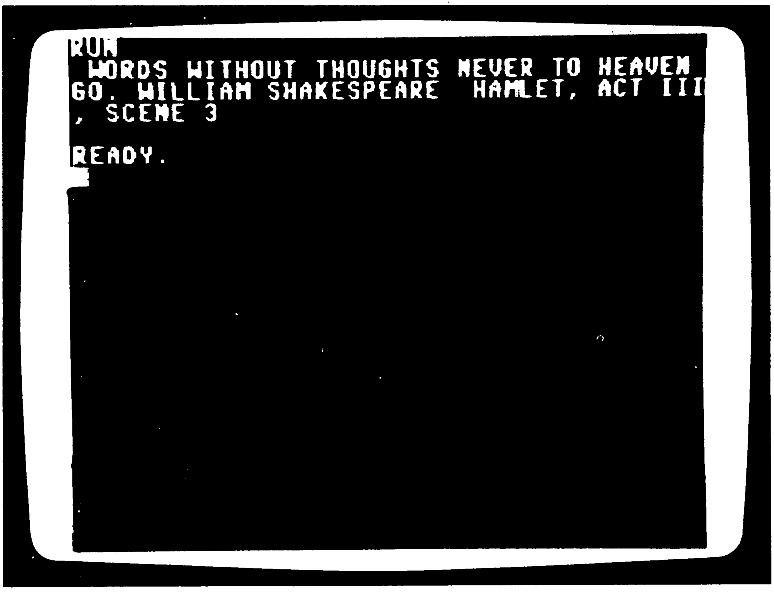
```
20 PRINT A$ + B$ + C$
```

When the program is run, the computer will respond with the display shown in Figure 6.1.

Counting Characters In Strings: LEN

You can extract a numeric value from a string with a command that counts the number of characters, including spaces, within it. Such a command can be useful in any program that manipulates strings. The command that directs the computer to count characters takes the form LEN( ). In a program, you can assign a string length value to a numeric variable with an instruction like this new version of line 20:

```
20 A = LEN(A$)
```



```
RUN
 WORDS WITHOUT THOUGHTS NEVER TO HEAVEN
GO. WILLIAM SHAKESPEARE  HAMLET, ACT III
, SCENE 3

READY.
```

Figure 6.1: The effect of a program that adds word-strings.

This line can be paraphrased as "Set the value of the numeric variable A equal to the number of characters in the string A$." It directs the computer to substitute the number of characters in string A$ wherever it encounters the symbol A in a program run. You can then use this value in another statement, like this:

```
30 PRINT "A QUOTE" A " CHARACTERS LONG BY " B$
```

and also add a statement to display the quotation itself after spacing a couple of rows:

```
40 PRINT: PRINT:PRINT A$
```

Taking Strings Apart: LEFT$, RIGHT$, and MID$

Three commands tell the computer to pick characters from within strings. Using them, you can take apart words or groups of characters, and rebuild them according to your own preferred patterns. The three string-slicing commands are similar in form and each cuts out a segment from the string.

One of these commands, LEFT$, instructs the computer to slice off a given number of characters from the left of a string. Characters are counted from the left quotation mark. If you wanted a new string, called D$, formed from the first five characters of the quote here, you would name and direct its formation with the command:

```
40 D$ = LEFT$(A$,5)
```

You could then display the new string with the command:

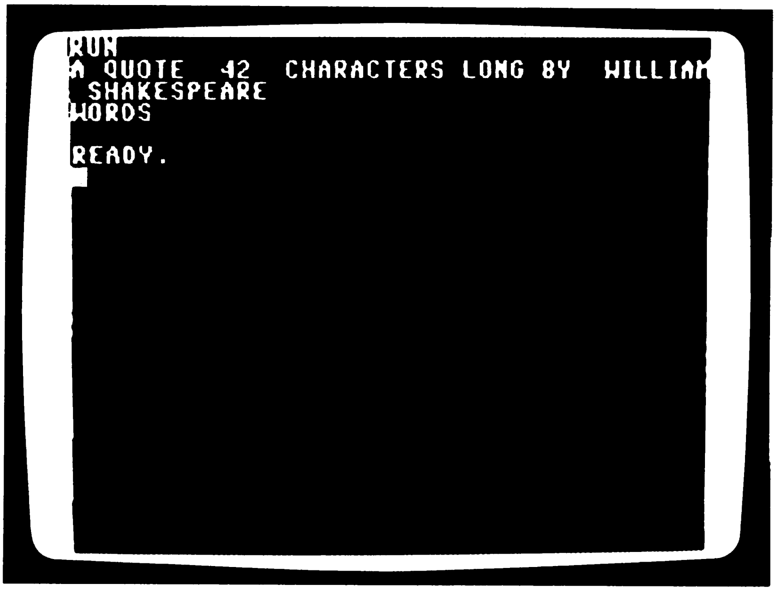
```
50 PRINT D$
```

When the program is run it produces the display shown in Figure 6.2.

You can create new strings of increasing length (up to the full length of A$) out of the original string, by including the LEFT$ command in a FOR loop with a counter variable:

```
35 FOR I = 1 TO A
40 D$ = LEFT$(A$,I)
60 NEXT I
```

The program now "brackets" the string-cutting statement of line 40 and the PRINT command at line 50 in a FOR-TO-NEXT loop that counts from 1 to the full length of the original quotation. When the program is run, the computer will print one row after another; each one begins with



```
RUN
A QUOTE  42  CHARACTERS LONG BY  WILLIAM
 SHAKESPEARE
WORDS

READY.
```

Figure 6.2: The effect of the string-slicing command LEFT$.

the first character of the string and is one character longer than its predecessor. When the last cycle is reached (when I = A), D$ will be assigned the value of LEFT$(A$,A), which is the full length of A$. D$ will be the same as A$, and the last line printed will be the entire quote, as shown in Figure 6.3.

By adding another statement, you can cite the source of the quotation:

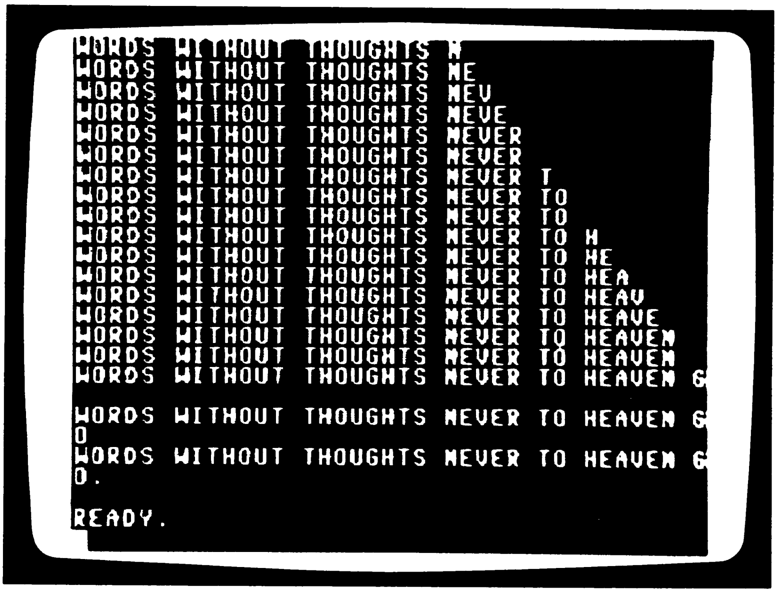
```
70 PRINT C$
```

You can make this program into a test of recognition by slowing printing with an empty FOR-TO-NEXT loop, like this:

```
38 FOR T = 1 TO 500: NEXT T
```

Now, with any quotation assigned to A$, you can run the program, stopping it from the keyboard with a RUN STOP keystroke command and advancing it with the CONT command.

The second string-slicing command, RIGHT$, directs the computer to count off and slice sections of strings from the right. Our current program will print sections of A$ counted from the last letter of the string, if the



```
WORDS WITHOUT THOUGHTS N
WORDS WITHOUT THOUGHTS NE
WORDS WITHOUT THOUGHTS NEV
WORDS WITHOUT THOUGHTS NEVE
WORDS WITHOUT THOUGHTS NEVER
WORDS WITHOUT THOUGHTS NEVER
WORDS WITHOUT THOUGHTS NEVER T
WORDS WITHOUT THOUGHTS NEVER TO
WORDS WITHOUT THOUGHTS NEVER TO
WORDS WITHOUT THOUGHTS NEVER TO H
WORDS WITHOUT THOUGHTS NEVER TO HE
WORDS WITHOUT THOUGHTS NEVER TO HEA
WORDS WITHOUT THOUGHTS NEVER TO HEAV
WORDS WITHOUT THOUGHTS NEVER TO HEAVE
WORDS WITHOUT THOUGHTS NEVER TO HEAVEN
WORDS WITHOUT THOUGHTS NEVER TO HEAVEN
WORDS WITHOUT THOUGHTS NEVER TO HEAVEN G

WORDS WITHOUT THOUGHTS NEVER TO HEAVEN G
O
WORDS WITHOUT THOUGHTS NEVER TO HEAVEN G
O.

READY.
```

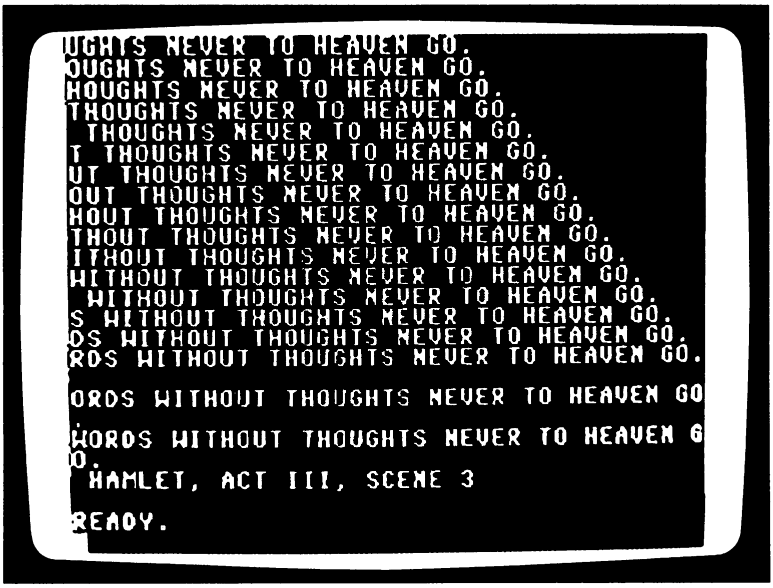
***Figure 6.3:** The effect of a program that generates increasingly longer strings from the LEFT$ command.*

following right-slicing command takes the place of the left-slicing one in line 40:

```
40 D$=RIGHT$(A$,I)
```

Driven by the same FOR loop, the computer will now print each line one character longer than its predecessor, but it will start from the right quote mark, as shown in Figure 6.4.

The third string-slicing command, MID$, directs the computer to snip a section from the center of a string. To use this command, you include the name of the string from which a section is to be excised, the leftmost character's position (as counted from the left side of the string), and the number of characters (to be counted and cut to the right). The command takes the form MID$(A$,s,n), where s is the starting point, and n is the number of characters in the section. If, for instance, you were to cut the eight-letter word THOUGHTS, which begins sixteen spaces from the left quote, from the quotation of line 10, the excising command would be MID$(A$,16,8). If you insert this command in place of the other string-cutters, again at line 40, the computer will produce another list of strings,



```
UGHTS NEVER TO HEAVEN GO.
OUGHTS NEVER TO HEAVEN GO.
HOUGHTS NEVER TO HEAVEN GO.
THOUGHTS NEVER TO HEAVEN GO.
 THOUGHTS NEVER TO HEAVEN GO.
T THOUGHTS NEVER TO HEAVEN GO.
UT THOUGHTS NEVER TO HEAVEN GO.
OUT THOUGHTS NEVER TO HEAVEN GO.
HOUT THOUGHTS NEVER TO HEAVEN GO.
THOUT THOUGHTS NEVER TO HEAVEN GO.
ITHOUT THOUGHTS NEVER TO HEAVEN GO.
WITHOUT THOUGHTS NEVER TO HEAVEN GO.
 WITHOUT THOUGHTS NEVER TO HEAVEN GO.
S WITHOUT THOUGHTS NEVER TO HEAVEN GO.
DS WITHOUT THOUGHTS NEVER TO HEAVEN GO.
RDS WITHOUT THOUGHTS NEVER TO HEAVEN GO.
ORDS WITHOUT THOUGHTS NEVER TO HEAVEN GO
WORDS WITHOUT THOUGHTS NEVER TO HEAVEN G
O.
 HAMLET, ACT III, SCENE 3
READY.
```

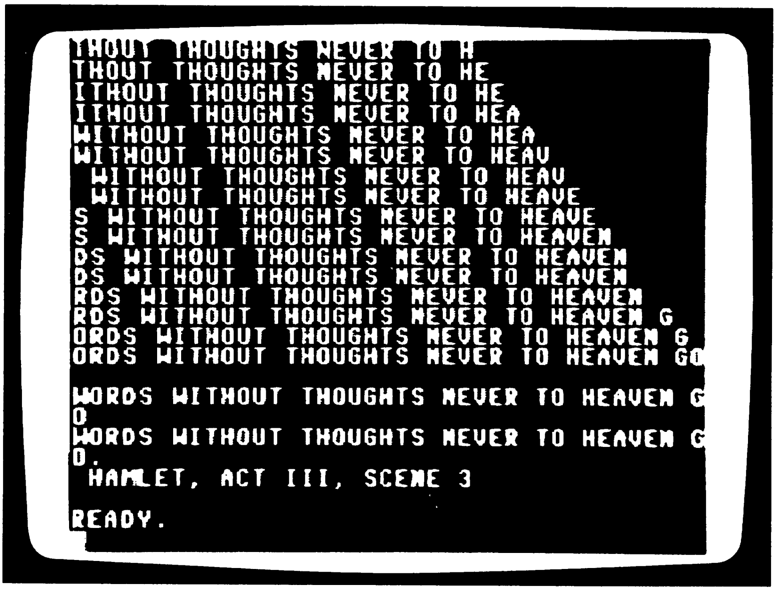
Figure 6.4: The effect of a program that generates increasingly longer strings from the RIGHT$ command.

this time growing outward from the middle of the original string, A$, in both directions.

To use the MID$ command to produce strings growing *symmetrically* from the middle of the original string, however, requires a twist of arithmetic. You can control the length of the string with the variable I, which is given value by the FOR-TO-NEXT statement, as it was in the LEFT$ and RIGHT$ commands. The computer finds the middle of the string A$ by dividing its total length (as found in line 20: A = LEN(A$)) in half: by A/2. The computer finds half the number of characters, I, to be printed at each pass, by I/2. Half of the characters will be printed from the right side, and half from the left of the center. Since the leftmost character of a string is numbered 1, the left boundary can be specified as A/2 - I/2 + 1. Thus, you can direct the computer with the MID$ command to produce strings from the middle of the original quote, A$, with the following command at line 40:

```
40 D$ = MID$(A$, A/2 - I/2 + 1, I)
```

A list of symmetrically growing strings will be printed, as in Figure 6.5.



```
THOUT THOUGHTS NEVER TO H
THOUT THOUGHTS NEVER TO HE
ITHOUT THOUGHTS NEVER TO HE
ITHOUT THOUGHTS NEVER TO HEA
WITHOUT THOUGHTS NEVER TO HEA
WITHOUT THOUGHTS NEVER TO HEAV
 WITHOUT THOUGHTS NEVER TO HEAV
 WITHOUT THOUGHTS NEVER TO HEAVE
S WITHOUT THOUGHTS NEVER TO HEAVE
S WITHOUT THOUGHTS NEVER TO HEAVEN
DS WITHOUT THOUGHTS NEVER TO HEAVEN
DS WITHOUT THOUGHTS NEVER TO HEAVEN
RDS WITHOUT THOUGHTS NEVER TO HEAVEN
RDS WITHOUT THOUGHTS NEVER TO HEAVEN G
ORDS WITHOUT THOUGHTS NEVER TO HEAVEN G
ORDS WITHOUT THOUGHTS NEVER TO HEAVEN GO

WORDS WITHOUT THOUGHTS NEVER TO HEAVEN G
O
WORDS WITHOUT THOUGHTS NEVER TO HEAVEN G
O.
 HAMLET, ACT III, SCENE 3

READY.
```

Figure 6.5: The effect of a program that generates increasingly longer strings from the MID$ command.

INSIDE INFORMATION

Like versatile actors within a play, variable symbols for numbers and characters can play many changing roles within a program. The equal sign directs these variables into their respective roles of the moment. But the Commodore 64 recognizes another way of assigning values to variables, which is somewhat the way a stage company's actors are given their assignments—together. You can assign values to several variables at once with a pair of commands, DATA and READ, in which you list the values to be sent to the computer, and the variables to which they are assigned, respectively. Any number of values can be sent to the computer with the command that handles the information list, DATA. You can use it to send the values of a string and two numbers, CHOKEBERRY, 15, and 30, with the following command:

```
NEW
10 DATA CHOKEBERRY,15,30
```

Then use the command that makes the assignments from the list of values, READ, to assign those values to the variables N$, A, and B, respectively, with this command:

```
20 READ N$, A, B
```

You can now use these assignments, which are the equivalent of the commands:

```
N$ = CHOKEBERRY
A = 15
B = 30
```

in a PRINT statement like this one:

```
30 PRINT N$ "AFTER" A "MONTHS"
```

Now add lines 10, 20, and 30 to a set of commands in which one value, B, is used to control a PRINT display:

```
12 PRINT "(press SHIFT and CLR/HOME)"
40 FOR X = 1 TO B
50 PRINT TAB(X) "(press C= and +)"
60 PRINT "(press SHIFT and CURSOR UP/DOWN) (press SHIFT
   and CURSOR UP/DOWN)"
70 NEXT X
```

The above lines direct the computer to draw a horizontal bar of a length corresponding to the value of B given in the DATA statement of line 10.

Running this program as it is, you'll see the heading:

```
CHOKEBERRY AFTER 15 MONTHS
```

and below it a horizontal bar, 30 columns long.

These two information-handling commands, one forming an information list, the other drawing from that list, can be used anywhere within a program. One command need not precede the other. If you place a READ statement before a DATA statement, the computer will seek out the DATA statement when it encounters READ, and make the assignment of values to the variables. Because it makes substitutions each time it encounters a READ statement, you can use a loop to command the computer to substitute several sets of values taken from a DATA statement for a single set of variables listed in its accompanying READ statement.

The program just run can be modified to handle a group of values, acting on each group, then returning to the READ statement for new assignments for the variables. You can easily add more names and values to the DATA statement:

```
10 DATA CHOKEBERRY,15,30,VETCH,25,14,BEARDWORT,17,27,
   CHICKWEED,9,21,THYME,11,25
```

But if you run the program now, the computer will respond as before, printing only the information about the chokeberry, since it acts on the READ statement only once. To direct it through the READ statement enough times to substitute a variable for each three-piece set of information in the DATA statement, you can add a FOR-TO, NEXT pair of commands to repeat lines 20 through 70:

```
17 FOR I = 1 TO 5
80 NEXT I
```

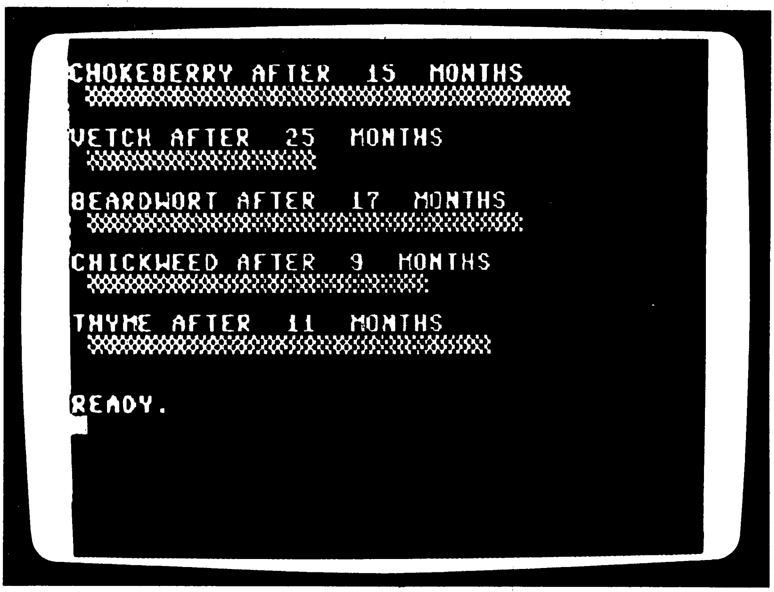
By including the READ-DATA commands in a FOR loop, you can direct the computer to introduce different values from the DATA statement into any often-used series of commands. To slow the plotting and printing pace, you can add an empty loop before the computer moves to another set of values:

```
75 FOR T = 1 TO 2500: NEXT T
```

Finally, because the loop of lines 17 to 80 includes cursor controls, which present a normal line return after each bar is drawn, a blank PRINT statement must be added:

```
25 PRINT
```

When you run this program, the computer clears the screen and moves the cursor to the top left corner. It then assigns to the variables N$, A, and B the first three values in the DATA statement, CHOKEBERRY, 15, and 30, respectively. It next prints the statement from line 30, and then follows the directions for drawing the horizontal bar. After running through the empty FOR-TO, NEXT loop in line 75, the computer is next sent back from line 80 to the READ statement at line 20, which assigns to the variables N$, A and B the next three values in the DATA statement (VETCH,25,33) and repeats the printing, plotting, and pausing. The NEXT command in line 80 once again sends the computer back through



```
CHOKEBERRY AFTER  15  MONTHS
 ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒

VETCH AFTER  25  MONTHS
 ▒▒▒▒▒▒▒▒▒▒▒▒▒▒

BEARDWORT AFTER  17  MONTHS
 ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒

CHICKWEED AFTER  9  MONTHS
 ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒

THYME AFTER  11  MONTHS
 ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒

READY.
```

Figure 6.6: Bar graph produced by using READ and DATA statements in a FOR loop.

the READ statement and another set of values. This process continues until the final run through the FOR-TO, NEXT loop, when the last values of the DATA statement are assigned and the display of Figure 6.6 is completed.

Besides making new assignments for variables, the DATA, READ and FOR-TO, NEXT program structures can be used to put a different name, tagged by a number, on each value in a DATA statement. With a FOR-TO, NEXT loop you can direct the computer to number each variable at passes through the loop. To name the variables so that each now matches a single value, you can change line 20 to read:

```
20 READ N$(I), A(I), B(I)
```

which will number the variables. On the first pass through the loop, when I = 1, the three variables N$(1), A(1), and B(1) will be assigned to CHOKEBERRY, 15, and 30. On the second pass, when I = 2, the three variables N$(2), A(2), and B(2) will be named to represent VETCH, 25, and 33. The computer repeats this process until the last pass, when N$(5), A(5), and B(5) are named and assigned.

You'll now have seven sets of second-named, or *subscripted,* variables.

You can direct the computer to bring them into a program statement by number, with these lines:

```
35 PRINT N$(I) " AFTER" A(I) "MONTHS"
45 FOR X = 1 TO I
```

The resulting program will produce the same results as its unsubscripted predecessor. But the advantage is that you can direct the computer to use these numerically-named variables in various ways, according to numbers calculated or assigned in other program statements. You can, for instance, direct the tabular printing of the plants' names and performances over time. Now the subscripted variables, A(I) and B(I), come in handy in a combination of PRINT and FOR-TO, NEXT statements, which will display all the information:

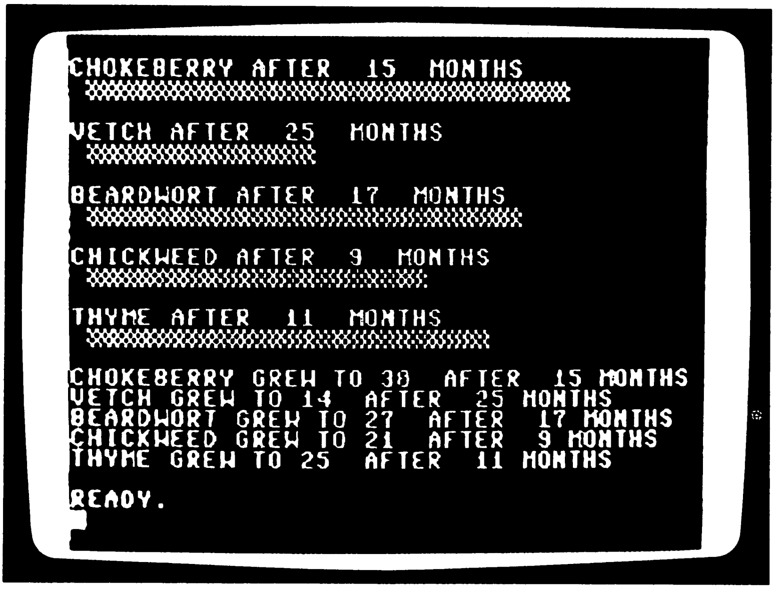
```
 85 PRINT
 90 FOR I = 1 TO 5
100 PRINT N$(I)"GREW TO " B(I) "AFTER " A(I) "MONTHS "
110 NEXT I
```

The program will now direct first the printing and bar graphs as it did before, then the printing of a list of the plants' performances, as shown in Figure 6.7.

If you have a use for variables with more than a single numeric tag each, you can add more tags. The variables N$(I,J), A(I,J), and B(I,J) could be named by wrapping another FOR-TO, NEXT loop, one that counts with the variable J, around the I loops. The numbers for J could stand for the watering interval of each plant, for example, and could range from 1 to 3. You can use several loops in this way to instruct the computer to name a series of variables in the form N$(I,J,K,L,M), in which I would be a code number for the plant, J might be its watering period, K the amount, L the amount of fertilizer, and M the time allowed, for example.

Each loop of FOR-TO, NEXT statements that you use to assign values to subscripted variable names multiplies the number of variables. Each DATA statement ought to contain as many values as requested by the accompanying READ statement. When the computer encounters a READ command asking for more information than is provided by a DATA command, it will stop the program and print the message:

```
OUT OF DATA ERROR
```



```
CHOKEBERRY AFTER  15  MONTHS

VETCH AFTER  25  MONTHS

BEARDWORT AFTER  17  MONTHS

CHICKWEED AFTER  9  MONTHS

THYME AFTER  11  MONTHS

CHOKEBERRY GREW TO 30  AFTER  15 MONTHS
VETCH GREW TO 14  AFTER  25 MONTHS
BEARDWORT GREW TO 27  AFTER  17 MONTHS
CHICKWEED GREW TO 21  AFTER  9 MONTHS
THYME GREW TO 25  AFTER  11 MONTHS

READY.
```

Figure 6.7: Another use of the DATA and READ values from the program that produced Figure 6.6.

When the computer first encounters a subscripted variable that is not numbered, it automatically sets aside space for as many as eleven numerically tagged variables that might later be produced. In this way, it reserves places in its voluminous memory for each of the assignments—A(1) = , A(2) = , and so on—that your program might generate from DATA and READ commands. The eleven variables will be numbered 0 through 10, for each set of subscripts. Thus, whenever the computer encounters a variable A(I), it reserves space for the variables A(0), A(1), and so on to A(10).

If you want the computer to handle variables numbered outside this range, you can direct it to put aside space for them. With the DIM command, you name the variable, and give the subscript the highest number you would like to allow for. To reserve space for the variables N$(0) through N$(50), for instance, you can send the command:

```
DIM N$(50)
```

For example, if you're considering fifty people who can each work one of three different shifts, and you wish to reserve space for all possible arrangements of names, times, and performances, the variables resulting could take any of the forms between N$(1,1), A(1,1), B(1,1) and N$(50,3), A(50,3), B(50,3). You can use the DIM command to save space for all of these variables and values by sending the directions:

```
DIM N$(50,3), A(50,3), B(50,3)
```

in a statement at the beginning of the program. On encountering this command, the computer will set aside an "array" of spaces in its memory, starting with N$(0,0) and ending with N$(50,3). No space problems will befall a program that runs without using all the spaces set aside by a DIM statement, but the program that tries to crowd in variables without reservations—an N$(51,3), for example—will be stopped in its tracks by an intolerant computer, with the reply:

```
? BAD SUBSCRIPT ERROR IN (line number)
```

THE CLOCK INSIDE

Whenever you switch on the computer, even if you don't touch the keyboard, the Commodore 64 secretly busies itself, keeping time. An internal, electronic clock begins running the instant power is switched on, and runs as long as the power is uninterrupted.

The time, in the form of a six-digit number representing hours, minutes and seconds, can be read by directing the computer to display its value. That value is stored, and continually updated, under the name TIME$. To check it at any moment you can give the command:

```
PRINT TIME$
```

The dollar sign in TIME$ indicates that the computer stores this value, though numeric, as a string variable, like a word. You can turn the computer into a displaying clock with this short program:

```
NEW
10 PRINT"(press CLR/HOME)"
20 PRINT TIME$
30 GOTO 10
```

which, when run, will show the seconds ticking by at the top of the screen.

You can include in a program an IF-THEN comparison that directs the computer to check the value of TIME$ (which can also be written as TI$) and act when it reaches a certain value. Such a statement might take the form:

```
IF TIME$ = "013520" THEN PRINT "YOUR TIME IS UP" : STOP
```

which, when encountered by the computer one hour, thirty-five minutes, and twenty seconds after the power is switched on, will declare on the screen YOUR TIME IS UP and stop the program run.

You can also adjust the clock, setting it to the current time much like any other. For instance, you could synchronize the computer's clock with one that gave the time as high noon, with this simple assignment statement:

```
TIME$ = "120000"
```

OUTSIDE CONTACT

Even while it's working on the commands of a program, the Commodore 64 is not entirely blind to what's happening out there where you are. It can respond to directions more subtle and informative than the throttling interruption you give it by pressing RUN STOP or switching off the power. But the computer, unimaginative and servile, has to be pointed in the right direction before it can know what you have to tell it.

The INPUT Command

You can add a command to a program so that the computer will stop, take the information you have for it, and then proceed with the run using what you provided. That command is INPUT. So commanded, the computer can interact with a person at the keyboard or with devices to which it is connected. The INPUT command directs the computer to "prime itself" by setting aside one or more variables, and then "siphon" the values typed at the keyboard. It then proceeds with the program, substituting those values as though it were responding to an assignment command or a pair of DATA, READ commands.

Using a simple form of the INPUT command, you can direct the computer to tell you that it's waiting for information, and then take what you type in (with a press of the RETURN key) and include it in the program.

The command takes the form of a simple request for a variable's value. You can use it to direct, for example, an instant doubling of whatever number you send from the keyboard in this short program:

```
NEW
10 INPUT A
20 PRINT A*
```

The statement, INPUT A, directs the computer to print a question mark on the screen as a signal to you that it's waiting for a reply from the keyboard. If you do nothing, the computer will remain frozen in its operation at line 10, waiting. If you type a number, say 537, and then press the RETURN key, the computer receives that number, makes the assignment A = 537, and then carries out the commands on the next line. In this case, the multiplication 537 × 2 is printed on the next row as 1074.

The INPUT command translates as: "Print a question mark, then take the next value sent from the keyboard, and substitute it for the variable in the statements that follow." The INPUT command can also direct the computer to act on strings in the same way, as in this short program:

```
10 INPUT A$
20 PRINT A$ + A$ + A$
```

which will stammer three times whatever you send.

Like DATA and READ, a single INPUT command can be used to direct the computer to act on both strings and numbers, as in this program:

```
10 INPUT A$, A
20 PRINT TAB(A) A$
```

If you run this program, and then respond with:

```
FROM THE KEYBOARD, 20
```

the computer will then print the words FROM THE KEYBOARD beginning at column 20.

You can use the INPUT command like the DATA and READ commands, typing values after the question mark (as after a DATA command) and setting variables for assignment after the INPUT command (as after a READ command). There is a difference, however: using an INPUT statement, you can send new values during each program run without changing a line of the program.

A second form of the INPUT command allows you to direct the computer in a convenient way, by condensing two commands into one. This form of the command directs printing on the screen, in place of the simple question mark, before waiting for a reply from the keyboard. You can use it by typing, after the word INPUT and within quotes, a question or statement you'd like displayed, followed by a semicolon to separate the variables you're directing the computer to ask for.

After sending a NEW command to clear the last program from memory, you can build a program from this form of the INPUT command, starting with:

```
10 INPUT "WHAT'S THE CODE PLEASE ? ";A$
```

which directs the computer to print your question, "WHAT'S THE CODE PLEASE ?", and then wait for a string that can be substituted for A$. This form of the INPUT statement, although it's more graceful, is exactly equivalent to the following commands:

```
10 PRINT "WHAT'S THE CODE PLEASE": INPUT A$
```

By adding a conditional (IF-THEN) statement, you can program the computer's printed responses to vary according to your "password" replies from the keyboard. You can arrange a program so that the computer encounters a conditional IF-THEN command after an INPUT command has pulled keyboard information into the program to be used in other commands. Doing this, you can make the program a two-way exchange of information, an *interactive* conversation with your computer.

To build such a program from the current INPUT statement, line 10, you can add an IF-THEN command and two possible responses, like this:

```
20 IF A$ = "THE ANCIENT MARINER SENT ME" THEN 40
30 PRINT: PRINT " I DON'T TALK WITH STRANGERS. ": GOTO 10
40 PRINT: PRINT " OK. I'LL TALK TO YOU. THIS COMPUTER
   RUNS ON A 6510 CHIP."
50 PRINT "I'M JUST AN INTERPRETER FOR THE PROCESSOR
   CHIP THAT REALLY RUNS THE"
60 PRINT "SHOW AROUND HERE. WHEN YOU GIVE COMMANDS
   IT'S THE 6510 CHIP THAT"
70 PRINT "MAKES THINGS HAPPEN."
```

When you run a program like the above, the computer is directed to take a value (or in this case a string of characters) from the keyboard, and then compare that value with another and act according to the comparison.

The INPUT command takes the string sent from the keyboard and assigns it to A$. The IF-THEN command compares it against the original string. If there is no match, the next command, line 30, directs a rebuff, "I DON'T TALK WITH STRANGERS", and sends the computer to act on the INPUT statement at line 10, which again asks, "WHAT'S THE CODE PLEASE ?"

When the reply sent from the keyboard (and assigned to A$) makes the IF condition true, the computer is sent by the THEN command to line 40, to PRINT the computer's inner secrets, after which the program ends. When reply after reply from the keyboard doesn't match the condition of the IF-THEN statement, however, the computer is trapped in the loop of lines 10 through 30 through 10, printing the rebuff and requesting the password over and over. The Commodore 64 is particularly stubborn with INPUT statements. Most other programs can be stopped with the RUN STOP command. Not so with programs waiting for a reply to an INPUT command. To stop such a program, you must use the firmer RUN STOP-RESTORE command, which directs the computer to stop the program run and retain the program in its memory.

The GET Command

You determine the computer's actions, of course, when you write a program, but you can also make the action depend on the result of a numeric calculation or on some other value given to the processor through some external device. There is another command, born of the same need for information as the INPUT command, that directs the computer in a similar, but terser, manner: GET. With it, you can command the Commodore 64 to act on a key pressed at the keyboard, without waiting for the RETURN key. It directs immediate substitutions and can order values for several variables at once, but it prints no message in quotes, or question mark. The main limitation of this command is that it takes only one character—number or string—for each variable.

You can use the command in a program of its own, like this:

```
NEW
 5 PRINT "(press SHIFT and CLR/HOME)"
10 GET A$
20 PRINT "(press CLR/HOME)" A$
30 GOTO 10
```

When this short program is run through the Commodore 64, the computer assigns the first key pressed to the variable A$, then moves the cursor to the upper-left corner of the screen, and prints the character of whichever key is pressed. If more than a single key is pressed, as in this reply:

```
TO ERR IS HUMAN
```

the computer will take the first letter typed, T, and display it. Then, because of the GOTO command at line 30, the computer will return to lines 10 and 20 to read and display each letter as it's typed, one at a time. The letters will replace each other in the corner immediately, since the computer reads much faster than any person can type. In fact, to use GET, you will *always* have to put it into a loop to make it repeat continually. Unlike INPUT, a single GET command will not wait long enough for you to input even a single character. To see this for yourself, delete line 30 and try the program again.

The features that make the GET command either useful or bothersome, depending on your point of view, are two: it directs the computer to ignore anything more than the variable it's set to assign, and it acts without waiting for the RETURN key to be pressed.

The ASC Command

Each key provides information to the computer each time it is pressed. That information travels as a numeric code, on which the computer takes some action: relaying the letter Z to the screen display (press Z), or stopping a program in progress (press RUN STOP), for example. You can see the number each key sends by using a command that directs the reading of each key numerically. That command is ASC(" "), and it produces the code number of the first character of the string typed in parentheses. For instance, ASC("Z") is 90. You can use the GET statement in a program that will display the code number of each key as you press it, like this one:

```
NEW
10 GET A$ : IF A$ = "" THEN 10
20 PRINT A$ TAB(5) ASC(A$)
30 GOTO 10
```

When you run this program, the character and the code number of each key you press will be printed on the screen as you press it.

Line 10 directs the computer to search for the next key pressed. If none is pressed during the instant it takes to perform a single GET operation, then A$ has no value (A$ = "") and the computer is sent back to search again. When a key is pressed, A$ is given a value and the computer moves to line 20, where it's directed to print the key pressed and then, spacing over, the numeric code. Line 30 restarts the process.

As this program runs, you'll find that each keystroke combination (except the RUN STOP and RESTORE keys, which actually stop the operation of the program) produces a code number. In fact, four keys will show a code number each, but no value. These four, marked f1 f2, f3 f4, f5 f6, and f7 f8, are known as *function* keys, and they were put on the keyboard not to affect the screen display, but to serve as programmable switches you can use to signal the computer while keeping the other, display keys free for use.

If you press f1, you'll see that it corresponds to the code 133, and that SHIFT-f1 (which you can think of as f2) corresponds to 137. Knowing this, you can use any of the function keys as a trigger for some action while a program is running. By adding the following statements to the program above, you can direct the computer to break out of the loop and take another action:

```
25 IF ASC(A$) = 133 THEN 40
40 PRINT "(press SHIFT and CLR/HOME)" TIME$
```

You can use other keystrokes in similar ways. For example, to make further use of the function key you can add:

```
27 IF ASC(A$) = 137 THEN 50
50 PRINT "NOW YOU'VE DONE IT!"
60 GOTO 50
```

The PEEK Command

The Commodore 64 can reach beyond the keyboard for information to act on. It does this through its circuitry, but you need not call an electrician or take out a soldering gun. In fact, if you plug a pair of game paddles into the side of the Commodore 64 at the jack marked CONTROL PORT 1, shown in Figure 6.8, you can open another line of information.

The Commodore 64 has the ability to electronically sense the positions of

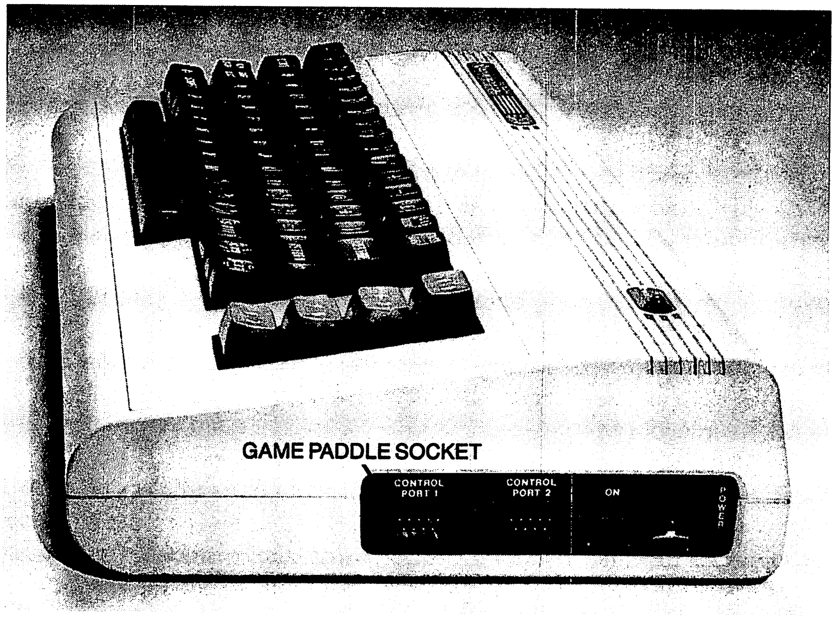


Figure 6.8: CONTROL PORT 1, the game-paddle socket.

the paddles or joystick, and to interpret those positions as numbers, from 0 to 255. You can direct the computer to take values from the paddle positions, for example, with the commands that introduce them into a program, PEEK(54297) and PEEK(54298). Then, acting as if those values had been assigned in any of the other ways already seen, the computer can treat them as it could any other value. The computer can read one numeric value from each paddle. Using the PEEK command to read the electrical status of a paddle, you can direct the computer to assign that value to a variable, which can then be acted on in the program.
The PEEK commands can be used in a statement like the one below, which assigns the value of each of the paddles to the variables X and Y:

```
NEW
10 X=PEEK(54297):Y=PEEK(54298)
```

The commands PEEK(54297) AND PEEK(54298) each translate as: "Generate a number from the electrical value of the paddle. When the paddle is turned completely counterclockwise, the value is 0; when it is clockwise, the value is 255; generate values in between according to position."

The statement in line 10 prepares the computer to substitute the value from one paddle for the variable X, and the value from the other paddle for variable Y. You can find out which is which by trying them with a program like the one that follows. You can add printing, screen-clearing, and positioning statements, and a repeating loop command, like this:

```
 5 PRINT"(press CLR/HOME)"
20 PRINT TAB(10) X TAB(20) Y
30 GOTO 10
```

When you run this program, the paddles feed in values through the PEEK commands. Program line 20 directs the computer to print the value of each paddle's position near the top of the screen. Incidentally, values below 100 will appear in three digits, as 990, 980, etc., on the screen.

You can add a statement to erase the previous value and so keep only the current value displayed, like this:

```
15 PRINT "(press SHIFT and CLR/HOME)"
```

On encountering the GOTO statement, the computer is directed back to the command that assigns the paddle values to variables. Thus, the values printed on the screen keep pace with the current paddle positions.

As this program is run, the values you see on the screen are generated from a physical device acting as an extension of the computer. This is the essence of computer monitoring.

You can use these values within a program just as you'd use values generated or supplied within it. With two so easily manipulated values, you can set in motion all kinds of action. For instance, you can draw directly with the paddles by deriving TAB and vertical spacing values from paddle positions. Building from program line 10 again, you can use the following program to do that:

```
 1 PRINT "(press SHIFT and CLR/HOME)"
 5 PRINT "(press CLR/HOME)"
10 X = PEEK(54297) : Y = PEEK(54298)
20 H = 30/255*X : V = 20/255*Y
30 FOR I = 1 TO V : PRINT : NEXT I
40 PRINT TAB(H) "(press SHIFT and Q)"
50 GOTO
```

The assignment commands of line 20 scale the values taken from the paddles to the dimensions of the screen. The loop formed by the commands of line 30 carries out a spacing function according to the value of one of the paddles.

7 ECONOMY CLASS STORAGE—TAPES

USING PROGRAMS WITH CASSETTE TAPES

People travel across country by both airplane and bus. One way is faster and costs more; the other is less convenient and slower, but is relatively inexpensive. The same sort of choice is available when it comes to storing and using programs and information files, both pre-packaged and your own. A disk system will move programs and information more quickly; a cassette tape system more slowly but at a much lower cost.

You can put your programs on common cassette recording tape through a Commodore cassette recorder. This is a specially-wired tape recorder (called a DATASSETTE ™) that plugs into the back of the Commodore 64, as shown in Figure 7.1. By operating it according to the instructions that appear on the screen whenever you use the recorder, you can store anything that resides in the computer's program memory on inexpensive cassette tapes.

The Commodore 64 recognizes simple commands for transferring programs to and from cassette tape. What's required on your part is a bit of button-punching on the recorder as on-screen directions guide you.

You can use the cassette tape in the same way as if you were recording speech or music. To make full use of a cassette tape, make sure the recording-protect notches at the edge of the cassette case are covered or blocked, and that the tape has been rewound to the beginning, but shows the dull brown tape through the window at the business-end of the cassette.

Slide the cassette into the plastic tracks of the recorder's lid. Then close

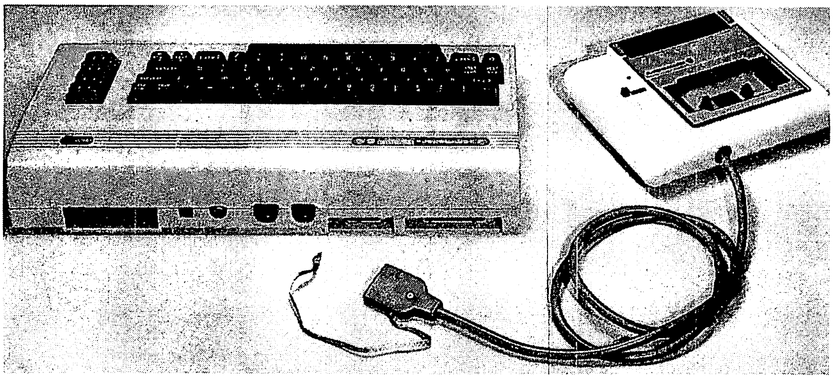


Figure 7.1: The connection between the computer and the cassette recorder.

the lid and you're ready to use the tape for program storage. The power to drive the recorder comes from the computer itself. You can use the counter to keep track of where your programs are by making a note of program and number, just as you might on any tape recorder. This is a good idea, since the tape system of storage won't automatically make a directory of programs contained on the tape. The programs will instead be treated much like a series of short songs or speeches recorded one after another through a microphone.

The procedure for recording a program is fairly direct. If you have a program in the computer's memory that you want to save under the name ENCHANTMENT, for example, you can give the command:

```
SAVE"ENCHANTMENT"
```

The computer will respond, in turn, with the prompt:

```
PRESS RECORD & PLAY ON TAPE
```

If you then press those two keys on the recorder together, the computer will quickly respond with:

```
OK
SAVING ENCHANTMENT
```

The screen will be blanked, the tape recorder's motor will turn, and a red light on the recorder will light. When the computer has finished recording the program, it will return the display screen and present a READY prompt and a flashing cursor. At this point, unless you want to save another program on tape, you can press the STOP key on the recorder. The program now resides in two places—on the tape and in the computer's memory.

You can bring a program from tape into computer memory just as simply. The computer will search an entire tape for that program by name and will load it when it's found. You can load the program ENCHANTMENT back into the computer from any point on the tape before the location of the program by giving the command:

```
LOAD "ENCHANTMENT"
```

The computer will respond with directions for you:

```
PRESS PLAY ON TAPE
```

When you do this the computer will respond quickly with:

```
OK
SEARCHING FOR ENCHANTMENT
```

If any programs precede ENCHANTMENT on the tape, the computer will blank the screen and then pause at each one it encounters, printing the message

```
FOUND program name
```

but it will continue its search for ENCHANTMENT. If, for instance, programs named AGGRAVATION, CHALLENGE, and OBSTACLES were recorded on the tape before it, the computer would blank the screen as it searched, and list each program it encountered, until it found the one you asked for. You would then press the C= key to actually load the program into memory. Incidentally, each time the computer stops the tape at a program in its search, you can load that program instead of the one you sent it off searching for, by pressing the C= key.

If you let the computer finish the search, it will eventually load ENCHANTMENT when it finds it. You would see this display finally:

```
FOUND AGGRAVATION
FOUND CHALLENGE
FOUND OBSTACLES
FOUND ENCHANTMENT
LOADING
READY.
```

That is the computer's way of telling you it encountered, but passed over, three other programs before locating the program ENCHANTMENT and loading it into its memory.

Incidentally, it's best not to "overlap" program names. The computer will stop at any program name that contains, within its first few characters, the name of the program being searched for. This would happen if you saved an additional program on the same tape, this one under the name ENCHANT, and then, after rewinding the tape to its beginning, commanded the computer to LOAD "ENCHANT". Finding the letters ENCHANT in the program name ENCHANTMENT, the computer would stop and load ENCHANTMENT.

You can use the searching ability of the Commodore 64 to generate a directory of the programs on a tape. By directing the computer to search for a program you know it won't find, you will put it through its searching and printing routine. If you send a command like

```
LOAD "ZZZ"
```

for a tape that has no such program, the computer will pause and print on screen the name of each program it encounters before reaching the end of the tape.

FACTS AND FILES ON TAPE

Although programs come and go on cassette tape, they're really no more than directions for operating the computer. Sometimes you may want to store and find information—addresses, names of places, people, or just bare facts. You can direct the tape-computer system to take in items typed on the keyboard and store them as an information file.

Since the computer has no convenient set of directions for working with information files (as it does with programs), you'll need a program that tells it what to do. The one that follows uses three new commands—OPEN, PRINT, and CLOSE—which control the flow of information between tape and computer.

```
100 REM A PROGRAM THAT CREATES FILES ON TAPE
110 DIM A$(100) : INPUT"NAME OF FILE";F$
120 OPEN 1,1,2, F$
130 INPUT A$(I)
140 "IF A$(I) = "CLOSE FILE" THEN 170
150 PRINT#1,A$(I)
160 I = I + 1 : GOTO 130
170 CLOSE 1
```

You can use this program in the following way. First, send the command:

```
RUN
```

The computer will respond with the question:

```
NAME OF FILE ?
```

If you then type in a name of your choosing, say WORMS, and press RETURN, the computer will prompt you to ready the tape recorder:

```
PRESS RECORD & PLAY ON TAPE
```

Then, after you've done so, the screen will go blank, the recorder's red light will glow, and the tape will turn. At this point the computer is directing the name WORMS onto the tape as a file name. When this has happened, the screen will reappear with printing and the new lines:

```
OK
?
```

The question mark is a prompt from the computer for the first item in the file. If you type an item, like NIGHT CRAWLER, and then press the RETURN key, the computer will prompt you for the next item with another question mark:

```
?
```

If you type another item, say SILKWORM, and send it with the RETURN key, the question-mark prompt will again appear. This program can accept 100 items. (Line 110 created an "array" of 100 variables for answers.) You can send item after item until you've put in the number you want. At that point, you just reply to the question-mark prompt with CLOSE FILE, and the computer will encode the entire list of items on the tape.

The screen will go blank, the red light on the recorder will come on, and the tape will turn. When the last item has been loaded, the tape will stop and the READY prompt will appear on the screen. Your list will be on the tape.

You can put any number of files on the tape this way, one after the other. Of course, encoded as they are, these files are of little use to you. But with another program to direct the tape system, you can pull the items out of a file and display them on the screen. The following program will retrieve a file by name from tape and list each item on the screen.

```
200 REM A PROGRAM THAT RETRIEVES A FILE FROM TAPE
210 DIM A$(100)
220 INPUT"NAME OF FILE"; F$
230 OPEN 1,1,0,F$
240 INPUT#1,A$(I)
```

```
250 PRINT I TAB(5) A$(I)
260 I=I+1
270 IF ST > 64 THEN 240
280 CLOSE 1
```

Notice the INPUT command in line 240. This command is a variant form of INPUT, used to read items from tape or disk memory instead of from the keyboard.

When you run this program to see the items of a tape file, such as the WORMS file we've created, you'll see the question:

```
NAME OF FILE ?
```

Once you type the name of the file you're looking for, and press RETURN, the computer will give you directions to operate the recorder:

```
PRESS PLAY ON TAPE
```

After you do that, the screen will go blank and the tape will turn until the file name and the items have been found. If you run the program when the tape is wound to its beginning and the file is at the end of the tape, the computer will search over each name until it finds the name you requested. (If the name is contained within a longer file name, the computer will retrieve that file instead of the one you requested. For example, the file HAPPENCHANCE, if encountered first, would be retrieved if the file HAPPEN was requested.) Finally the tape will stop, and the computer will display the number assigned to each item in the list, and the item itself, as it prints a list on the screen.

You can use these programs separately to store and gather files with tape, and you can incorporate them into your own programs, as long as your programs assign variables to items in the form A$(I).

8 HIGH SPEED STORAGE—DISKS

Soon after its appearance, a gallant and effective Pony Express yielded its service of the growing frontier to the faster and technologically superior telegraph. Likewise, in many computer applications, the tape storage system we discussed in Chapter 7 is giving way to a new system. A technological improvement over the tape system of storage, a disk system offers a speed of access that can be important when frequent exchanges of programs and information must be made between computer and storage medium.

A MARRIAGE OF MACHINES

A disk system operates under the electrical and logical control of the Commodore 64 as an extension of the computer's own memory, a warehouse of programs and information. Each element of this system provides more than twice the memory capacity of the Commodore 64, which is 64 kilobytes. With a disk storage system, you can put aside and use again your own programs and other programs already written for the Commodore 64.

More complex than the cassette tape system, the disk storage system works through a box about the size of the computer that contains its own circuitry and small memory. Working under the computer's direction, this *disk drive* magnetically encodes programs and information onto the thin plastic circles known as *disks,* each contained loosely within a square plastic envelope.

These disks are coated with a magnetically-sensitive material and respond like recording tape to an electromagnetic device that scans their surfaces inside the disk-drive box. The disk drive includes a motor for spinning the disks within their envelopes, phonograph-style. Since these disks can be run in a drive one at a time, and then removed and replaced by other disks, they form an interchangeable and unlimited set of repositories for programs and information passing to or from the computer.

Figure 8.1 shows a disk drive and some disks.

Once connected to the computer electrically, the disk system also responds to commands that aren't part of the vocabulary of the Commodore 64. Through these commands, you can direct the computer to put aside a program or information from its memory on a disk and to pull from a disk into its memory.

The Commodore 64, which communicates with one disk at a time, will accommodate five disk drives, but you can handle most disk uses conveniently with only two. In fact, a single disk drive will suffice for most disk uses, although less conveniently, at times requiring more inserting and withdrawing of disks on your part.

The disk system and the computer can exchange control of each other interactively. Your commands can direct the computer to a particular disk drive (if more than one is installed) and to a particular program or file on the disk in the drive. In converse fashion, a program directing the

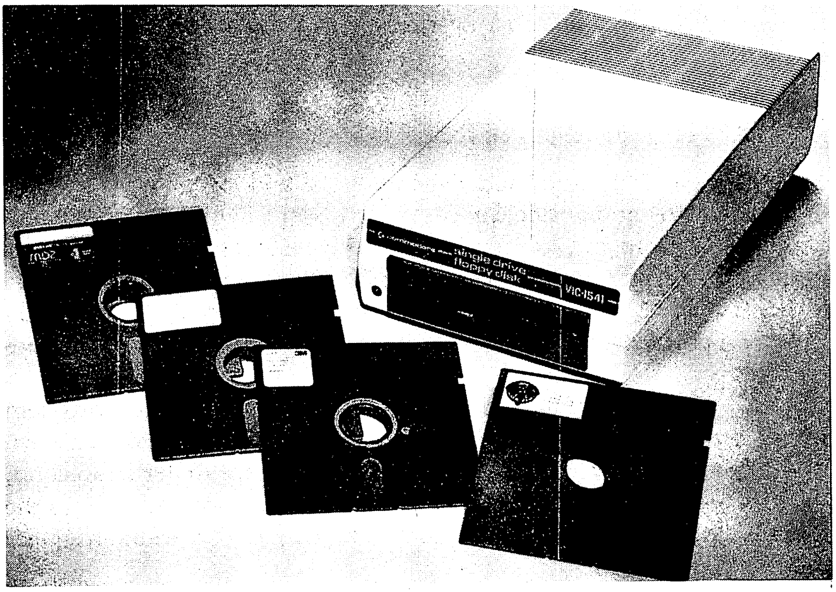


Figure 8.1: A disk drive and some typical disks.

computer's action from a disk can guide it to carry out commands or sequences just as you would from the keyboard.

Each program or information file brought out from a disk is first fed into the computer's memory before any action is taken on it. The information makes its way between disk drive and memory by means of a simple electrical cable.

Each disk drive contains the circuitry for running its motor and moving the magnetic drive head. In fact, the disk drive has its own small parcel of memory space for retaining programs and information, and its own power supply as well.

You won't need an electrician to connect things. Here's what to do: First, turn off the power to the computer. With the power off, you can't damage the machine. The cord packed with the disk drive has a metal sleeve at each end, surrounding six pins. Plug one end of this cord into the back of the drive box, into the jack just above the fuse. You then plug the other end into the round jack furthest from the power light at the back of the computer. The final connection to make is the 3-pronged power cord. Plug the boxy end into the the back of the disk drive, and the standard end into a power outlet. The connections are shown in Figure 8.2.

After you switch on first the disk drive power (at the rear of the drive box), then the computer power, the electrical connection between computer and disk drive system is complete. You can position the parts of the system—computer, disk drives and monitor or television—wherever the wires will reach.

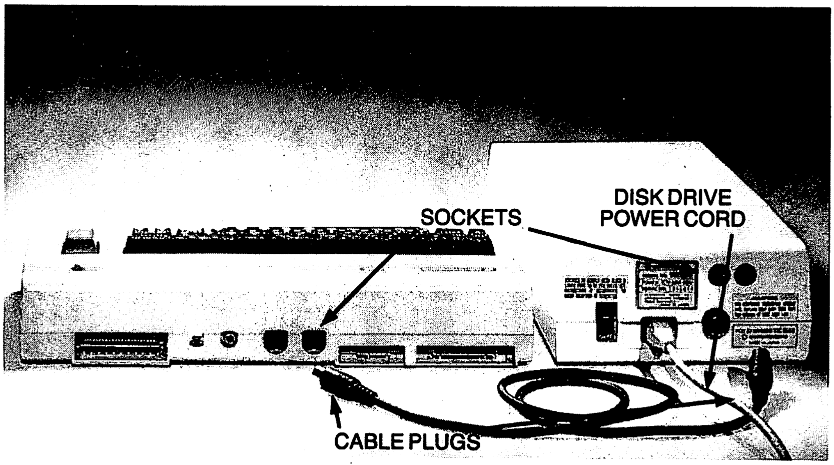


Figure 8.2: The connection between computer and disk drive.

USING PRERECORDED DISKS

First, a word about how to treat disks. Like paper, disks come in two forms, blank and written on. You can copy recorded disks, write on blank parts of recorded disks, and write over, or erase, parts of recorded disks.

You can also protect the contents of a disk, as you might put written paper in a binder so that it won't be unexpectedly written on. The square envelope around the disk protects the magnetic writing from erasure or alteration in the drive if that envelope is completely sealed around all edges and lacks a cut-out notch, called a *write-enable* notch. If such a notch is exposed on an envelope, as illustrated in Figure 8.3, the disk drive will be switched out of a reading-only mode of operation. You can then direct the computer to record your typing on the disk.

You can "freeze" all the programs and information on a disk bearing such a notch, by covering that notch with a sturdy piece of foil tape, which is usually provided with blank disks. When the mechanisms of the disk drive sense that this space is covered, the drive will be switched to the read-only mode.

As their nearly total enclosure suggests, disks are fairly fragile. Even when a disk has endured some mishap with no visible damage, its magnetically-sensitive surface may have been damaged. Magnetic fields, extremes of temperature (those outside our comfort range), bending or

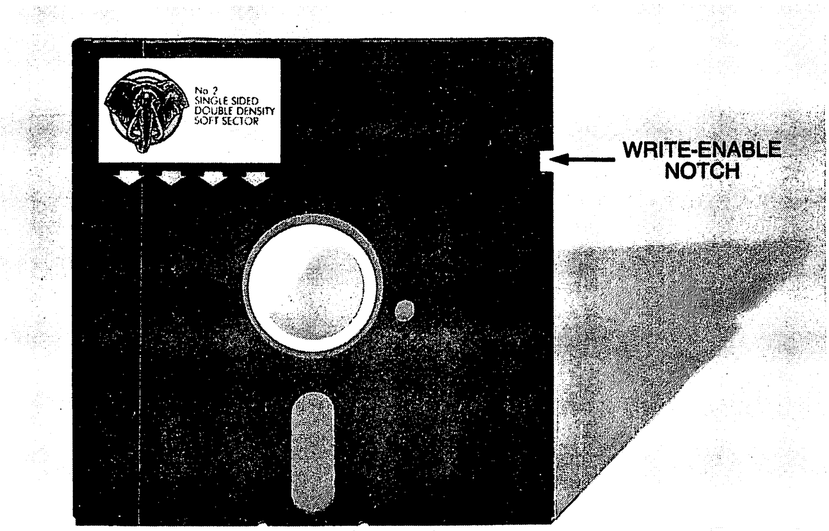


Figure 8.3: The write-enable notch of a disk.

folding, and contamination by dirt, dust, skin oils, and the like are all hazards. Information and programs recorded on the disk surface in invisible magnetic patterns can be cripplingly altered by an imperceptible scratch or contamination. Hidden in its envelope, nearly all of the disk is protected from contamination at any one time; you can handle the envelope without concern. But the drive head reads where the disk surface itself peeks through an oval slot, and through that slot can pass dust and contaminants as well.

You can easily avoid common disk problems by taking two simple precautions: Always remove disks from the drive before turning off the disk drive or the computer; and always mark labels for disks before you stick them onto the disk envelopes.

The disk and its plastic envelope are stored in a paper jacket, which doesn't quite cover the envelope. This paper sleeve shields the otherwise exposed disk surface at the oval cutout when it's out of the drive.

The magnetic head in the disk drive, recording and reading along a short track, makes contact with the disk through the envelope slot while the disk itself is spinning inside the envelope. Because the disk envelope is square and two-sided, there are eight possible ways to put it into the drive. Only one way will position the disk correctly.

The drive head is in the rear half of the drive, so the slotted end of the disk envelope goes in first. One side of the disk envelope is unlabeled and relatively unfinished-looking; the other side is labeled and smooth. If this labeled side faces the little door at the front of the drive, the magnetic side of the disk will face the drive head.

Thus, you insert the disk into an empty, opened drive box like this: slot in first, labeled side facing the door. Then all you do is slide the disk gently all the way in, until it catches in the recess. You can then push down on the drive door until it catches to close over the center of the opening.

You remove the disk by pushing the door until it's pulled up by spring action. The disk itself will slide halfway out of the drive box under the same spring action.

Inserting a disk in a drive box and closing the door physically completes the computer-disk system. But unless it's given the special commands that direct the disk drive, the system is about as useful as an illiterate librarian. Even when it is physically and electrically connected with a disk system, the computer must still be told what to do with this apparatus.

The computer can respond to your disk commands, as well as to the BASIC programming commands built into it. Disk directions automatically compile a listing of programs and file information as they are put onto the disk. The TEST/DEMO disk, supplied with Commodore disk drives, holds several programs already, though it's not nearly filled. We'll use the TEST/DEMO disk to get started. You can command a listing of the programs on that (or any other) disk, and the computer will respond with a display showing the name given to the disk and a listing of its programs and files. Each is preceded by a number indicating the amount of space it occupies on the disk.

You can direct the computer to display this list with the command:

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

The computer will respond with a message on the screen, and the disk drive's red light will glow as a signal that the drive is in operation. You can also take this glowing red light as a reminder not to remove the disk at this time.

If the computer-and-disk-drive system doesn't recognize the disk command sent to it, the red light on the drive will flash, as a signal that the last disk command hasn't been carried out. The system will subsequently accept any command it recognizes and then extinguish the light after performing the command.

If you've sent the command above, the message you'll see is:

```
SEARCHING FOR $
```

The dollar sign, $, is a shorthand name for the directory. Once the directory is found, the computer will reply with the message:

```
LOADING
```

That means the computer is copying information from the disk (in this case, the directory listing) into its own memory. When the computer has finished copying, it will display the familiar signal:

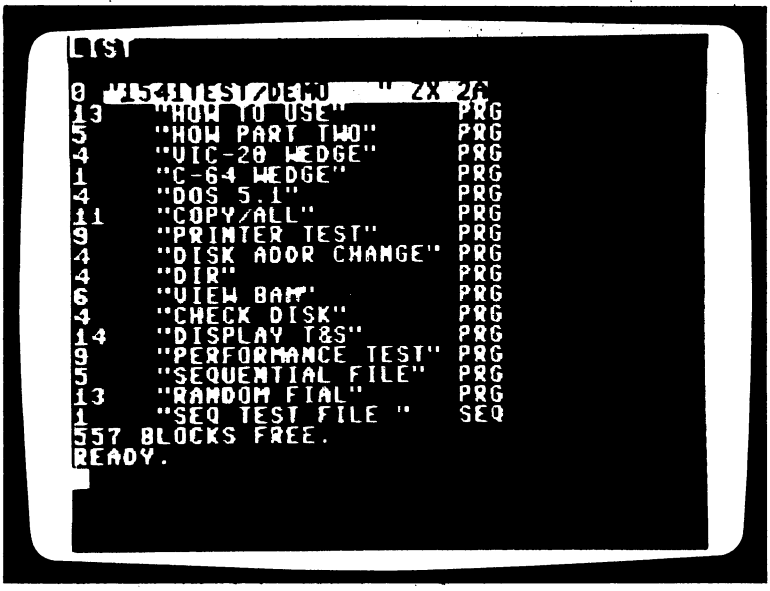
```
READY.
```

You can then direct the computer to print on-screen the directory just loaded, with the command:

```
LIST
```

The screen will fill with a list like that shown in Figure 8.4.

A directory listing can have more items than will fit on a single screen.



```
LIST

0 "1541TEST/DEMO       " ZX 2A
13   "HOW TO USE"          PRG
5    "HOW PART TWO"        PRG
4    "VIC-20 WEDGE"        PRG
1    "C-64 WEDGE"          PRG
4    "DOS 5.1"             PRG
11   "COPY/ALL"            PRG
9    "PRINTER TEST"        PRG
4    "DISK ADDR CHANGE"    PRG
4    "DIR"                 PRG
6    "VIEW BAM"            PRG
4    "CHECK DISK"          PRG
14   "DISPLAY T&S"         PRG
9    "PERFORMANCE TEST"    PRG
5    "SEQUENTIAL FILE"     PRG
13   "RANDOM FIAL"         PRG
1    "SEQ TEST FILE "      SEQ
557 BLOCKS FREE.
READY.
```

Figure 8.4: The TEST/DEMO disk directory.

The Commodore 64 will display the entire listing by "scrolling" the first items of the list up and off the screen to make room for later items printed after the first screenful. Once it has displayed the final item of the directory, the computer signals its readiness for your next command with the READY prompt. You can stop the listing at any point with the RUN STOP keypress command.

You control the disk drive through the computer with commands given in the same form as the LOAD command above. The computer will recognize these commands if they each follow the pattern: first the command, then the name of the program in quotes, then a comma, then the number 8, which opens a path to the drive connected to the computer.

These commands increase in complexity as you order more complex tasks. However, one of the programs on the TEST/DEMO disk can simplify the commands you give to operate the disk drive. After you've directed the computer to load and then run this program, the Commodore 64 will respond to either those rather cumbersome disk commands it's built to act on, or another, shorthand set of symbolic commands. The directions for these shorthand commands are stored in an area of the computer memory apart from the usual program memory, so you can use them over and over without interfering with the programs or files you put in the computer.

You'll find this program listed in the diectory under the name C-64 WEDGE. The C-64 WEDGE program will control the disk drive so that a set of directions corresponding to these shorthand commands is stored in the "reserved" part of the computer's memory. These directions will respond to your shorthand disk commands, and will remain in memory even after the command NEW erases whatever program you've put into memory.

With the TEST/DEMO disk in the drive, you can put the C-64 WEDGE program into effect by directing the computer to load it:

```
LOAD "C-64 WEDGE",8
```

Once the computer has completed its search and loaded the program into memory, you'll see the familiar READY prompt. You can then direct the computer to run the program, and put the directions for its shorthand commands aside in memory, by commanding:

```
RUN
```

As the program runs you'll see a three-line display of title and credits, after which the familiar READY prompt appears as a signal that disk directions for the shorthand commands are available in memory.

The computer will now respond to either set of commands. The longer, more cumbersome commands are always available from the computer. The shorthand C-64 WEDGE commands are there whenever you load and run that program from the TEST/DEMO disk.

Since you may be using either type of command, both are shown in the explanations that follow, although you may prefer to use the WEDGE commands for convenience. The longer commands are shown here in *italics* as alternatives.

You can now load and list the directory with a single command:

```
@$
```

which gives the same directions as:

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

You can run the first program of the TEST/DEMO disk, a display program called HOW TO USE, with the WEDGE and program commands:

```
/HOW TO USE
RUN
```

The slash symbol , /, serves here as an abbreviated form of the command

LOAD. You could get the same results with the standard disk commands:

```
LOAD "HOW TO USE",8
RUN
```

As this program runs, you'll see a description of the programs on the TEST/DEMO disk and directions on their use. You can stop the display as you could any program, by pressing the RUN STOP key.

The / and LOAD commands each direct the computer to erase its program memory and to copy onto it a program from the disk. Their effect is like typing NEW and a program. (You can use another symbolic command, %, to run programs written in the machine's own numeric language, not in BASIC.)

The WEDGE program contains directions that allow you to give both the LOAD and the RUN command together, in an even more abbreviated form. It takes the form ↑, and can be used to give the same results as above, though more quickly. You can direct the computer to immediately begin acting on the commands of a program from the disk drive by giving that single command. If you want the computer to find, load, and run a program like HOW TO USE on the TEST DEMO, you can give the command:

```
↑ HOW TO USE
```

The computer will then skip a line and signal you with the statement:

```
SEARCHING FOR HOW TO USE
```

Then another line will tell you the computer is bringing the program into its memory:

```
LOADING
```

After this, the computer will immediately begin carrying out the commands of the program. In this case, it displays the screen titled DISKETTE INSTRUCTIONS.

The / , @ , and ↑ commands are all you need to use prerecorded disks operating under the directions of the TEST/DEMO WEDGE commands.

PROGRAMS FROM COMPUTER TO DISK

You can record programs and information on any disk that has a recognized magnetic pattern etched onto it. A disk in the drive without this pattern is as useless to you using the computer as a bare desk to a writer with a pen. With them, the disk is like a blank sheet of lined paper, lying ready on the desktop.

You can direct this magnetic etching of a disk with a single command from the WEDGE program. In the process, you will erase any information already on the disk, and put on it a sectioned pattern, in which programs and information can be stored. At the same time, the disk will be given a name. The computer-disk system also looks for a two-character code on each disk as it is first scanned. You can provide this code within the same simple command.

To prepare a disk this way, giving it the name THE FIRST, and the code A1, you can send the following command:

```
@N:THE FIRST,A1
```

or

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

then

```
PRINT #15,"NO:THE FIRST,A1"
CLOSE 15,8,15
```

The computer will take control of the disk drive, and you'll hear the sound of magnetic formatting (like fabric tearing) for several moments. When all is quiet again and the red disk drive light goes off, the disk has been "initialized," and is at your service. The newly-prepared disk can hold program and information files, just as the TEST/DEMO disk can. You can selectively put useful programs from another Commodore 64 disk (like the TEST/DEMO disk) onto any initialized diskette.

To record a program on a disk, you first type it into the computer. Any program will do—the *Hamlet* quotation from Chapter 6, or one of your own invention. You can type in a program you'd like to store on disk, just as you would before running it, or you can load a program from another disk or from a cassette tape. You can, of course, run the program to see that the computer does what you want with it, and make any necessary changes. When you've got the program as you like it in memory, everything is ready for the disk command that will preserve it outside the computer.

All that remains is for you to name the program. The disk directions tell the computer to handle names up to 16 characters long. If you want to call your program COLD STORAGE, for example, you can direct the computer to store it under that name by the command:

```
← COLD STORAGE
```

or

```
SAVE "COLD STORAGE", 8
```

The disk drive will make its fabric-tearing sounds. When the commotion stops and the red light goes off, the program exists in two places. Magnetically encoded, it's on the disk. And as before, it's in the computer's program memory. It hasn't moved, it's been copied.

If you type LIST, the program statements will appear on the screen. You can type NEW to erase the program memory and still run the program, this time from the disk, with the command:

```
↑ COLD STORAGE
```

or

```
LOAD "COLD STORAGE", 8
```

then

```
RUN
```

It will then be freshly loaded into the program memory of the computer. You can remove the disk, turn off the computer, send the disk round-trip to Chicago if you like, and—on its return—put that disk in the drive box, turn on the computer and type:

```
↑ COLD STORAGE
```

or

```
LOAD "COLD STORAGE", 8
```

then

```
RUN
```

and the program will manifest itself.

The computer was quietly following another disk direction when you commanded ← COLD STORAGE. It put the name of the program, preceded by a number indicating the amount of space it takes, on the directory. If you now send the directory command:

```
@$
```

or

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

then

```
    LIST
```

at the end of the directory, you'll see:

```
    (a number like 2)        COLD STORAGE     PRG
```

You can change the name of a program with another disk command. Directed by this command, the computer looks for the program under the first name listed, then substitutes for it the second name.

To change the name of the program COLD STORAGE to STOWAWAY, for example, you can send the command:

```
    @R:STOWAWAY = COLD STORAGE
```

or

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

then

```
    PRINT #15,"R0:STOWAWAY = COLD STORAGE
    CLOSE 15,8,15
```

So directed by the RENAME command, the computer changes only that program's name, leaving the program itself and its position in the CATALOG unaffected.

There's no need to clutter a disk with a program you no longer want. To erase a program once and for all, you can give a command like this:

```
    @S:STOWAWAY
```

or

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

then

```
    PRINT #15,"S0:STOWAWAY"
```

The computer, under disk directions, will operate the drive to find the program, erase it entirely, and remove its name from the directory.

You can move programs individually from one disk to another using the commands that load (/), save (←), and delete (@S:). First you load the program from one disk into the computer's memory, then save it by sending it from the program memory to another disk, and finally delete it from the disk on which it originated.

To move a program like STOWAWAY from one disk to another using a one-drive system, you can first bring it into the computer's program memory:

```
/STOWAWAY
```

then remove its disk from the drive, insert a second disk—one equipped with its own magnetic etching—and send the command:

```
← STOWAWAY
```

then reinsert the original disk and have it erased with the command:

```
@S:STOWAWAY
```

Through the save, load, delete, and rename disk commands, you can juggle and reshape programs on disks quite freely. Any magnetically-prepared disk can hold program and information files, just as the TEST/DEMO disk can. You can selectively put useful programs from another Commodore 64 disk (like the TEST/DEMO disk) on any initialized diskette.

You could copy a program like SEQUENTIAL FILE, for instance, with the command:

```
/SEQUENTIAL FILE
```

or

```
LOAD "SEQUENTIAL FILE", 8
```

Then, replacing the TEST/DEMO disk with an initialized one on which you'd like to store the program SEQUENTIAL FILE, you could give the command:

```
← SEQUENTIAL FILE
```

and the computer would duplicate that program on the disk you had initialized.

Using this procedure, you can duplicate selectively and move programs around your diskettes as you find useful. In particular, you can copy the information- and file-handling program of SEQUENTIAL FILE onto a disk that will be used for a special purpose, like holding information files.

USING MORE THAN ONE DISK DRIVE

Just as several disks can serve you as a reservoir of programs and information, several disk drives expand the number of pathways for items moving in and out of the computer memory. With two drives, for instance, you can load programs from, or store them on, the disks in either drive.

To connect a second drive to the system, simply take the six-pronged cord that comes with the drive, and plug one end into the socket over the fuse holder on the second drive; then plug the other end into the socket remaining on the drive you've already connected directly to the computer. The electrical connection will then run from the computer directly to one drive, and from that drive to the second drive. The power cord from the second drive connects to an outlet in the same way as for the first drive.

The computer can now communicate with either drive. If you turn on the computer and either one of the drives, leaving the other turned off, the computer will carry out the commands described above to operate the drive which is powered.

If you switched on both drives and gave a command, like that to load the directory, the computer would be stymied. Without instructions for which of the two disk drives to search for the directory, it may arbitrarily choose one or the other, or it may lock up, giving you the message that it's searching for the directory, $, when in fact it's not controlling either drive. In that case, you'd have lost control of the computer by sending it off to make a decision for which it hadn't been programmed.

The key to avoiding this problem is to realize that built-in directions tell the computer that each drive is identified by the same number. This device number is automatically set at 8, and you can see it in a command like LOAD "$",8. Any drive connected to the computer, directly or through another drive, is identified by this number automatically. But the computer can be directed to label any drive with a number of your own choosing through commands that re-label each drive. These commands are number-laden and abstruse, and involve setting up command channels and files, but you don't need to understand the details of how they work to use them.

To change the device number by which a connected disk drive will be called from the computer, here's what you can do. First, switch off each drive except the one you wish to renumber.

Send the following statement (with the RETURN key, of course):

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

Then send this statement, if you want the disk drive to be numbered 9, for example:

```
PRINT #15,"M-W"CHR$(119)CHR$(0)CHR$(9 + 32)CHR$(9 + 64)
```

If you wish to renumber another drive, switch it on and repeat the commands above, substituting another number for 9 in the last two sets of parentheses. If, for instance, you wish to label another drive as 22, you can turn it on and follow this sequence:

```
OPEN 15,8,15
PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$(22+32)
     CHR$(22+64)
```

The device number you assign to each disk drive will then be recognized by the computer until it is again changed, or until power is switched off.

Since the Wedge commands assume a drive numbered 8, they will be effective only for the drive automatically given that number, or for one assigned that number. If numbers 8 and 9 are assigned to two drives, for example, you can use the long form of the disk commands to control each of the drives, and the Wedge commands to control drive 8.

You could copy a program from one disk in drive 8, for instance, to another in drive 9 with a sequence of commands like this. First type

```
LOAD "SEQUENTIAL FILE",8
```

then wait for the READY prompt, and type

```
SAVE "SEQUENTIAL FILE",9
```

FACTS AND FILES ON DISK

Programs are useful enough, but let's face it, what you really want the computer to do is handle information—data, facts, figures, names, addresses, times, places, quantities, descriptions.

Any information that can be typed can be put on a disk. You can store it under a name on the directory listing and bring it out through the computer to the screen. You can direct the computer to work on it, manipulating it according to program statements.

Like files in a cabinet, information on a disk is at your disposal to pull out for reading, or to use in programs in place of DATA and INPUT statements. However, the computer needs more detailed directions for putting information that's not organized as a program on disk, and for pulling that information back into its memory.

One of the programs on the TEST/DEMO disk handles information files on disk. It has two functional parts. One part directs the computer to put keyboard information on an initialized disk; the other part directs it to read information from a disk.

More elaborate programs, which direct the computer in complex manipulations of information files, are commercially available on prerecorded disks. If so inclined, you can write some yourself by building from the TEST/DEMO disk file program. It provides a set of commands that direct the computer in information transfers.

In carrying information files—also known as *text files*—to and from a disk, the Commodore 64 treats them just as it treats strings of words in a program. Whether you direct numbers, words, commands, or declarations of emotion into these text files, the computer will treat them as simply a group of characters.

You can use the file program without studying how it works. You can also list the program and strip away all but the essential file features to fashion a streamlined sequence of file-handling statements. This sequence can then be added to other programs for manipulating information and perhaps making new files, if you like.

The program that directs the computer in forming information files is listed on the TEST/DEMO disk as SEQUENTIAL FILE. You can use it most easily if you first copy it onto a disk you've "initialized" with a magnetically-etched pattern. Then, when you command:

```
↑SEQUENTIAL FILE
```

or

```
LOAD "SEQUENTIAL FILE",8
RUN
```

the computer clears the screen and prints descriptions and directions for the program's use. At this point you can ignore the first couple of lines and follow the directions after them, ENTER A WORD, COMMA, NUMBER. If you want to enter, as first on the list, the name AMBROSE and the number 1, you can then type after the prompt A$,B:

```
AMBROSE,1
```

and then send it to the computer by pressing the RETURN key. You can then send another name and number similarly:

```
BRET,357
```

and a third name and number:

```
MARK,2
```

If you'd like the list to contain just these three items, you can signal the computer to close the list with the response:

```
        END
```

sent by a press of the RETURN key. A pair of question marks will appear as a prompt. If you press the RETURN key again, the computer, acting under the SEQUENTIAL FILE program's commands, will take it as a signal that you've finished the file. Following the directions from the second part of this program, the Commodore 64 will print the line, READ SEQ TEST FILE. At this point, SEQ TEST FILE is the name the program has given to the list of items you've typed. Finally, the computer will print on-screen the list of items and numbers.

You'll find, after running the SEQUENTIAL FILE program, that the computer has added the file to the end of the directory listing under the name SEQ TEST FILE. The file listing appears as:

```
        1 " SEQ TEST FILE " SEQ
```

The file information has been copied item-by-item into the computer from the disk, as though in response to keyboard typing after an INPUT statement, and is still present on the disk.

9 BUILDING AND REBUILDING PROGRAMS

A program can have a rather uncertain existence. It can be run repeatedly, performing the same function each time, or altered to fit different occasions. The clever and creative way to use your computer is not to write a new program for each new project, but rather to adapt and build from programs and parts of programs that have proven their usefulness. How much easier it is to start with groups of commands whose effects are known than to begin each project facing a blank screen!

ADDING PROGRAM PARTS

You can direct the Commodore 64 to piece together programs (or parts of programs) you find particularly useful or interesting. Program parts can be added within the computer's memory. You can do this even if you understand little more than the connections among programs or program parts. To create your own style of programming, you can call on ready-made, often-used statement sequences. Using such an approach, you can take sequences you're confident of, and splice them into a program you design to take advantage of each part.

For example, to display a name at the bottom center of the screen, you can direct the computer with this two-line program:

```
NEW
  10 DATA BONAPARTE THE AMBITIOUS
1000 READ A$: PRINT "(press SHIFT and CLR/HOME)"A$
```

You can use the screen-clearing, positioning and printing features of line 1000 again, by directing the computer to carry out the commands of that line for other names (i.e., other values of A$). The pair of commands, GOSUB and RETURN, that direct the computer to line 1000 and back can be added to the program, with these statements:

```
  20 GOSUB 1000
1010 RETURN
```

A third statement, END, will stop the computer from acting on line 1000 again. It can be placed anywhere before line 1000, like this:

```
100 END
```

On encountering the GOSUB command, the computer skips any intervening lines of statements and acts on line 1000 immediately. Then, in normal sequence, it performs the commands of any lines that follow 1000. When the computer encounters the command RETURN, it's directed back toward the GOSUB statement and the line immediately after it. Having carried out the commands between line 1000 and the RETURN statement, the computer resumes the sequence at line 100, which directs it to stop.

The GOSUB command can be translated as: "Mark the line number of this statement in your memory, then carry out the commands beginning at the line number indicated." The RETURN command translates as: "Return to the marker made by the GOSUB command that sent you, and then continue action with the next line number following it."

Using this pair of commands, you can direct the computer to a group of statements anywhere in the program. The GOSUB command can be used several times in a program to direct the computer to a group of statements, as in the following sequence:

```
 10 DATA BONAPARTE THE AMBITIOUS
 14 GOSUB 1000
 20 DATA ALBERT THE UNDERRATED
 24 GOSUB 1000
 30 DATA MELVILLE THE MAGNIFICENT
 34 GOSUB 1000
100 END
```

At each place where it appears, the GOSUB 1000 command directs the computer to act on line 1000 and any lines that follow it. The RETURN statement directs it back to the main sequence of DATA and GOSUB statements. If you add an empty loop to line 1000, to delay the change of displays:

```
1000 FOR T = 1 TO 500 : NEXT T: READ A$ : PRINT
     "(press SHIFT and CLR/HOME)" A$
```

and then run this program, you'll see the name given in each DATA statement printed at the top left corner of the screen, one after another.

You can direct the computer to act on any number of command groups—known as *subroutines*—in any order. You can add a graphing function to this program with another group of commands:

```
2000 N$ = "(press SHIFT and W)"
2010 FOR I = 1 TO 6
2020 READ A(I)
2030 PRINT "(press CLR/HOME)"
2040 Y = 23 - A(I) : X = 5*I
2050 FOR V = 1 TO Y : PRINT "(press CURSOR UP/DOWN)" ; :
     NEXT V
2060 PRINT TAB(X) N$
2070 NEXT I
2080 PRINT "(press CLR/HOME)" : RETURN
```

This group of commands will direct the computer to READ values from a DATA statement, and then, by a little arithmetic manipulation of X and Y values, to plot those values.

You can now make further use of the GOSUB command. If you expand the DATA statements in lines 10, 20 and 30, you can put the commands starting at line 2000 to work on those values:

```
10 DATA BONAPARTE THE AMBITIOUS, 5.5,7,11,17,9,1
20 DATA ALBERT THE UNDERRATED, 2,3,16,17,18,19
30 DATA MELVILLE THE MAGNIFICENT, 5,10,7,15,17,12
```

These values could stand for yearly weights, or productivity, or size of vocabulary, for instance.

To direct the computer to the graphing subroutine you can add, after each of the GOSUB 1000 printing commands, a GOSUB 2000 command:

```
18 GOSUB 2000
28 GOSUB 2000
38 GOSUB 2000
```

When run, this program will direct the computer first to print the name, then to plot the corresponding set of numbers.

Because it sends the computer to another line number, a GOSUB command can be used within one subroutine to direct the computer to another. In such a case, the computer will pass through the first group of commands to the second. Encountering the RETURN command of the second group, it will be sent back to the first group. When it encounters the RETURN command of the first group, it will be sent back to the main program sequence.

Using the GOSUB-RETURN command pair in this way, you can add subtleties and complexities to programs in a simple, "modular" fashion. For example, you can enhance the graphing feature of this program by adding grid lines against which to compare each of the plots. A sequence of commands that will do this could be grouped as two subroutines, one that draws vertical lines starting at line 3000:

```
3000 PRINT "(press CLR/HOME) (press CURSOR UP/DOWN)"
3010 FOR V = 1 TO 20
3020 FOR I = 0 TO 39 STEP 5
3030 PRINT TAB(I) "(press C = and G)(press SHIFT and CURSOR
     UP/DOWN) (press SHIFT and CURSOR UP/DOWN)
     (press SHIFT and CURSOR UP/DOWN)"
3040 NEXT I
3050 PRINT
3060 NEXT V
3070 RETURN
```

and another sequence of commands that draws horizontal lines and a number scale, starting at line 4000:

```
4000 PRINT "(press CLR/HOME) (press CURSOR UP/DOWN) (press
     CURSOR UP/DOWN)"
4010 FOR Y = 0 TO 20 STEP 5
4020 FOR X = 3 TO 35
4030 PRINT 20 - Y TAB(X) "(press C = and @) (press SHIFT and
     CURSOR UP/DOWN)"
4040 NEXT X
4050 IF Y = 20 THEN 4070
4060 FOR S = 1 TO 5: PRINT: NEXT S
4070 NEXT Y
4080 RETURN
```

You can direct the computer to draw the grid, just after it reads the values from directions in the subroutine beginning at line 1000, by adding this statement:

```
1007 GOSUB 3000 : GOSUB 4000
```

When you run this program, first the screen will clear and the name "BONAPARTE THE AMBITIOUS" will appear at the top. Then the computer will draw grid lines. Next it will plot the values from the first DATA statement, as shown in Figure 9.1. The screen will clear again, and the process will begin for the values from the next DATA statement. It will continue until each set of values from the three DATA statements has been handled, and the computer stops at the last display.

The computer acts on GOSUB and RETURN commands in pairs. If it encounters a RETURN statement without having first encountered a GOSUB statement, it will follow its built-in directions to signal you with an error message:

```
RETURN WITHOUT GOSUB ERROR IN (line number)
```

This problem would result from the computer's usual progress through higher line numbers to a high-numbered subroutine. You can avoid it by putting an END or STOP statement after the main sequence but before the first of the subroutine statements. In this case, we've put an END at line 100.

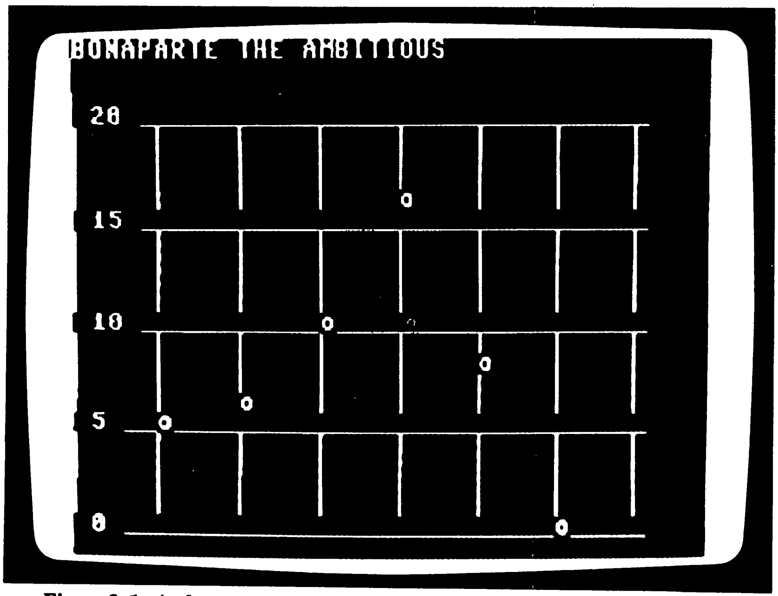


Figure 9.1: A plot created by four subroutines.

Like the GOTO command, GOSUB directs the computer to the line number listed in the command. You can use a variation of the GOSUB-RETURN command pair to direct the computer, in a conditional way, in and out of the main sequence of statements. Just as an ON-GOTO command directs the computer to depart from the normal sequence on certain numeric conditions, so an ON-GOSUB command directs the computer to particular subroutines. You can use this command to direct the computer to one of several command groups, from which it will later be sent back to resume the main sequence again.

The ON-GOSUB statement can be used in the plotting program, for instance, to selectively draw parts of the grid pattern against which the information will be graphed. One group of commands, like those beginning at line 3000, can direct the computer to draw vertical lines. Another group, like those beginning at line 4000, can direct the drawing of horizontal lines.

By changing program line 1007 to a statement whose directions to the computer are conditional you can, according to the choice made during each program run, either use or pass over the subroutines beginning at 3000 or 4000, or a new one at line 5000. This subroutine produces the full grid, by sending the computer to draw vertical, then horizontal, lines. The full grid could be drawn through these statements:

```
5000 GOSUB 4000
5010 GOSUB 3000
5020 RETURN
```

The command that sends the computer to each of those subroutines, on certain values of the variable D, is:

```
1007 ON D GOSUB 3000, 4000, 5000
```

This directs the computer like the ON-GOTO command, comparing the value of the variable with the ON-GOSUB command's built-in ranges of value: one or greater but less than two, and so on. You can instruct the computer to request the value of D, by adding a prompt and an INPUT statement, like these:

```
1005 PRINT "PRESS THE NUMBER OF THE DISPLAY YOU'D LIKE:"
1006 INPUT "1-VERTICAL; 2-HORIZONTAL; 3-FULL GRID";D
```

Running the program, you'll see the INPUT request at the top of the screen. If a choice in the range 1, 2, or 3 is sent, the computer will first draw the graph lines according to choice, then plot the values from the DATA statement. If a choice outside the range is typed, the computer will simply move to the next program line and no graph lines will be printed; the DATA values will be plotted on an unlined screen.

You can build a library of useful subroutines, giving each of them high line numbers to avoid interference with a lower-numbered main sequence. In fact, your main program can sometimes be nothing more than a series of requests for subroutines, along with, perhaps, some INPUT or DATA sequences. In assembling such a program, the only elements that must match are variables that are common to the subroutines and are acted on in each subroutine. You can be sure of this match by checking to see that a given variable name stands for the same quantity wherever it is used. The main sequence of such a program could be as simple as:

```
 10 GOSUB 1000
 20 GOSUB 2000
 30 GOSUB 3000
  .
  .
  .
100 END
```

The final statement, END, stops the computer from running on to the subroutines that follow. Following this structure, you can save yourself work by taking advantage of subroutines already written by you or someone else, to build programs giving you greater control over your computer. You can concentrate on arranging existing elements, rather than on writing detailed program commands.

REBUILDING BORROWED PROGRAMS

Not all useful programs are created in the plodding process of adding one command after another. You can borrow from existing programs, alter them to suit your needs, and so make use of the original programming effort that went into those programs. By borrowing a disk program—for example, the SEQUENTIAL FILE program of the TEXT/DEMO disk—you can use its disk-handling abilities in programs of your own.

The SEQUENTIAL FILE program, which actually performs two distinct file-handling tasks, making information files on disks and then reading them, can be tailored into two separate parts for use in your programs. Many of its statements do nothing more than describe the program functions, and can be removed. By elimination of those commands that you don't need, you can trim a bit of excess from the original.

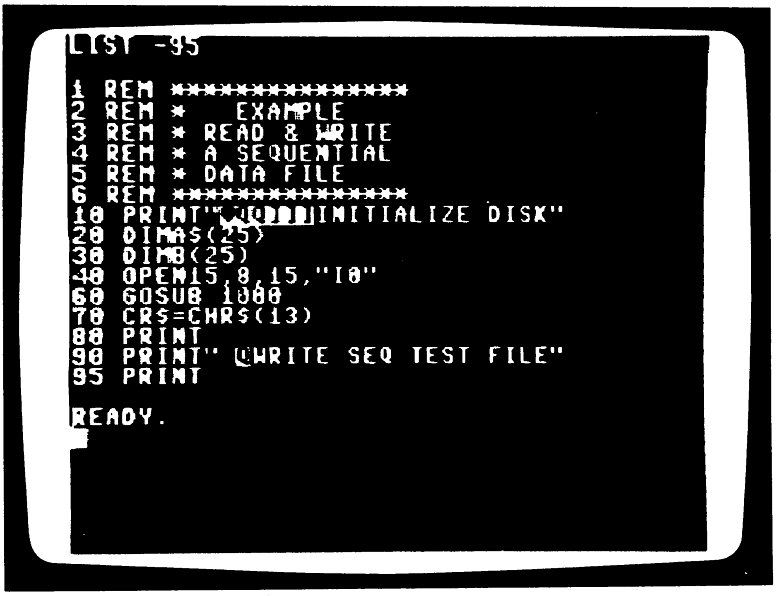
You need not understand all the details of the program to make use of it and its parts. For instance, if you look at the entire listing of SEQUENTIAL FILE, a screenful at a time, you can see that the program is divided into four parts, each separated by the notes of a group of REM statements.

The first group of REM statements, numbered 1 through 6, describes the function of the program that follows, AN EXAMPLE READ & WRITE A SEQUENTIAL DATA FILE. The statements that follow that group of remarks, lines 10 through 95 (shown in Figure 9.2), hold commands common to both functions of the program.

The next group of remarks, numbered 100 through 105, describes rather tersely the part of the program that directs the formation of a disk file, WRITE SEQ TEST FILE. Those statements occupy lines 110 through 160. (See Figure 9.3.)

The remarks from lines 200 through 205 describe the statements, lines 206 through 420, that direct the reading into the computer of information from the file, SEQ TEST FILE, produced by the first half of the program, READ SEQ TEST FILE. (See Figure 9.4.)

Finally, the notes in lines 1000 to 1006 describe a sequence of contingency commands in a subroutine, statements 1010 through 1060, that will

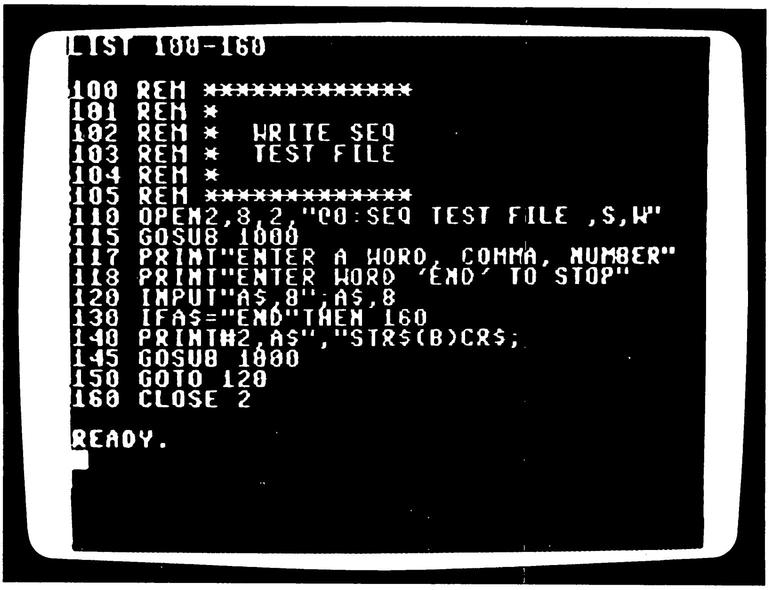


```
LIST -95

1 REM ****************
2 REM *    EXAMPLE
3 REM * READ & WRITE
4 REM * A SEQUENTIAL
5 REM * DATA FILE
6 REM ****************
10 PRINT"INITIALIZE DISK"
20 DIMA$(25)
30 DIMB(25)
40 OPEN15,8,15,"I0"
60 GOSUB 1000
70 CR$=CHR$(13)
80 PRINT
90 PRINT" WRITE SEQ TEST FILE"
95 PRINT

READY.
```

Figure 9.2: The statements of the SEQUENTIAL FILE program that prepare the computer to handle a file.

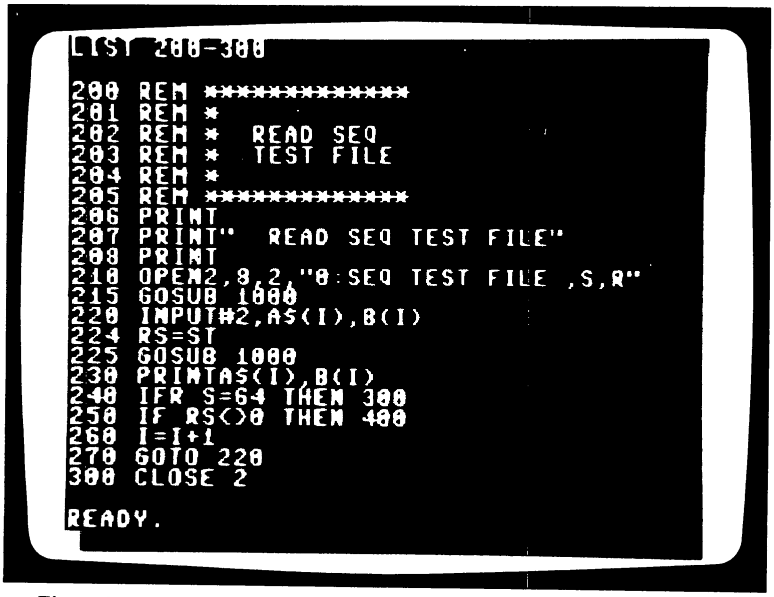


```
LIST 100-160

100 REM *************
101 REM *
102 REM *  WRITE SEQ
103 REM *  TEST FILE
104 REM *
105 REM *************
110 OPEN2,8,2,"@0:SEQ TEST FILE ,S,W"
115 GOSUB 1000
117 PRINT"ENTER A WORD, COMMA, NUMBER"
118 PRINT"ENTER WORD 'END' TO STOP"
120 INPUT"A$,B";A$,B
130 IFA$="END"THEN 160
140 PRINT#2,A$","STR$(B)CR$;
145 GOSUB 1000
150 GOTO 120
160 CLOSE 2

READY.
```

Figure 9.3: The statements of the SEQUENTIAL FILE program that send information to a disk file.

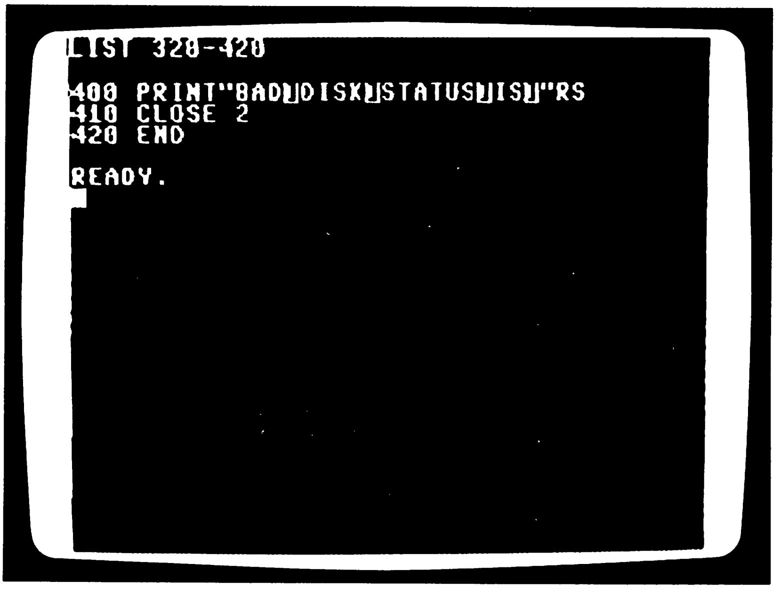


```
LIST 200-300

200 REM *************
201 REM *
202 REM *  READ SEQ
203 REM *  TEST FILE
204 REM *
205 REM *************
206 PRINT
207 PRINT"  READ SEQ TEST FILE"
208 PRINT
210 OPEN2,8,2,"0:SEQ TEST FILE ,S,R"
215 GOSUB 1000
220 INPUT#2,A$(I),B(I)
224 RS=ST
225 GOSUB 1000
230 PRINTA$(I),B(I)
240 IFR S=64 THEN 300
250 IF RS<>0 THEN 400
260 I=I+1
270 GOTO 220
300 CLOSE 2

READY.
```

Figure 9.4: The statements of the SEQUENTIAL FILE program that take information from a disk file.



```
LIST 320-420

400 PRINT"BAD␣DISK␣STATUS␣IS␣"RS
410 CLOSE 2
420 END

READY.
```

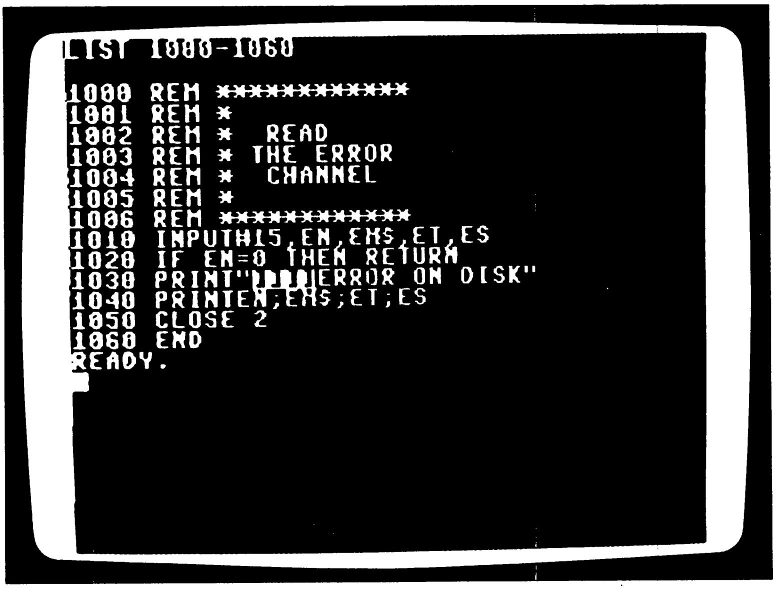
Figure 9.4, continued.

display a code for problems that might develop from unrecognized commands between the computer and disk drive, READ THE ERROR CHANNEL. (See Figure 9.5.)

To cut out that part of the program that creates the file in the first place, you really only need the lines up to 160, and the subroutine from line 1010 on. You can begin to tailor this program to your file-making needs by deleting the program lines between 200 and 410. You do that, you recall, by sending each line number with the RETURN key:

```
200
201
 .
 .
 .
400
410
```

The program remaining in the computer's memory will prompt you to form a disk file of up to 25 word-number combinations, which it will name SEQ TEST FILE. It's a program that has several statements in excess of those that actually direct this function. You can trim off excess statements by deleting the remarks of lines 1 through 6, the redundant



```
LIST 1000-1060

1000 REM ************
1001 REM *
1002 REM *  READ
1003 REM * THE ERROR
1004 REM *  CHANNEL
1005 REM *
1006 REM ************
1010 INPUT#15,EN,EM$,ET,ES
1020 IF EN=0 THEN RETURN
1030 PRINT"[illegible]ERROR ON DISK"
1040 PRINTEN;EM$;ET;ES
1050 CLOSE 2
1060 END
READY.
```

Figure 9.5: The statements of the SEQUENTIAL FILE program that act on a disk problem.

PRINT statement of line 10, and the remarks and spacing of lines 95 through 105. (Keep in mind, however, that you or another user might later appreciate having the REM lines preserved.)

The program that remains after this trimming will direct the computer to form a disk file through your interaction. If you run it more than once to create different files, however, it will give each set of data the same name, SEQ TEST FILE. By altering two statements, you can modify the program to assign a name of your choosing to each file you create with this program.

At line 110 is the statement that names the file to be put on disk. You can make it more general by simply changing the part that names the file to a variable. To do this requires the kind of addition that you can perform on word strings using a plus sign. Using the variable F$ to represent the file name, you can replace line 110 with the statement:

```
110 OPEN 2,8,2,"@0:" + F$ + ",S,W"
```

Now the program directs the computer to give each file the name assigned to the variable F$. You can make this assignment when the program is run, if you include a statement that asks you for a file name. An INPUT statement in place of the PRINT statement of line 90 would do that:

```
90 INPUT "NAME OF FILE TO BE MADE"; F$
```

Now the computer holds a versatile program that can create disk file after disk file in response to your entries, and can assign to each a different name. Using this program you can create many files, each of which will be listed by name on the directory of the disk it is stored on.

The resulting program you'll see on sending the command LIST (as shown in Figure 9.6), is trimmed and to the point. It will direct the computer to ask you for the name of a file you wish to produce, and then, through the prompts of its INPUT statement, will collect and finally send to the disk drive the elements you supply from the keyboard.

You can save this program as it is, by putting it on a disk on which you'd like to store files. You might put it there under the name MAKE FILE, for instance:

```
←MAKE FILE
```

or

```
SAVE "MAKE FILE",8
```

On the other hand, you can include this program in a larger program, if you like, or make some further adjustments to it.

As it was orginally written, the file-handing program reserves space for only 26 entries. Lines 20 and 30 make this allocation:

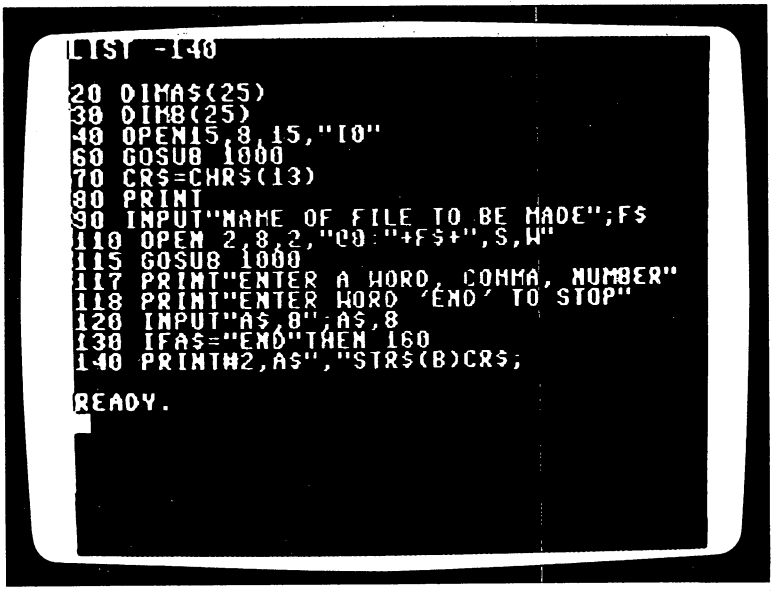
```
20 DIM A$(25)
30 DIM B(25)
```

You can extend the number of elements to be put in the files you create, by modifying those statements to reserve more space in computer memory. If you wanted to reserve room for 100 items, for instance, you could simply send new statements to reserve space:

```
20 DIM A$(100)
30 DIM B(100)
```

Likewise, the format of the items entered can be changed by changing the INPUT statement of line 120 to accept any number of items.

By the same thoughtful tailoring, you can produce a file-reading program from another part of the SEQUENTIAL FILE program. In this case, you load the SEQUENTIAL FILE program into the computer

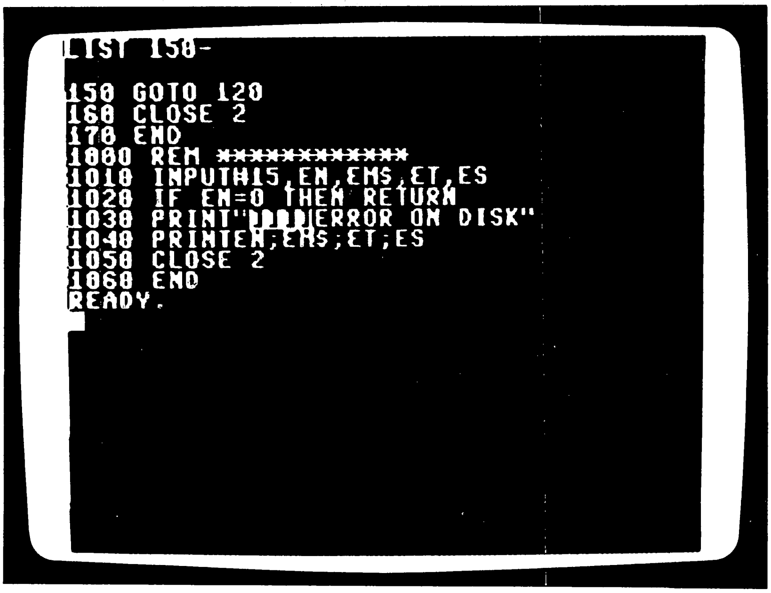


```
LIST -140

20 DIMA$(25)
30 DIMB(25)
40 OPEN15,8,15,"I0"
60 GOSUB 1000
70 CR$=CHR$(13)
80 PRINT
90 INPUT"NAME OF FILE TO BE MADE";F$
110 OPEN 2,8,2,"@0:"+F$+",S,W"
115 GOSUB 1000
117 PRINT"ENTER A WORD, COMMA, NUMBER"
118 PRINT"ENTER WORD 'END' TO STOP"
120 INPUT"A$,B";A$,B
130 IFA$="END"THEN 160
140 PRINT#2,A$","STR$(B)CR$;

READY.
```

Figure 9.6: A tailored file-writing program, derived from the SEQUENTIAL FILE disk program.



```
LIST 150-

150 GOTO 120
160 CLOSE 2
170 END
1000 REM ************
1010 INPUT#15,EN,EM$,ET,ES
1020 IF EN=0 THEN RETURN
1030 PRINT"███ERROR ON DISK"
1040 PRINTEN;EM$;ET;ES
1050 CLOSE 2
1060 END
READY.
```

Figure 9.6, continued.

memory and begin trimming it to that single function. Again, lines 1 through 10 can be dropped:

```
1
2
.
.
.
10
```

Lines 70 through 206, which direct file formation, can also be dropped:

```
70
80
 .
 .
 .
206
```

Statements 1001 through 1006, which are simply remarks, can be dropped as excess baggage, too:

```
1001
1002
1003
1004
1005
1006
```

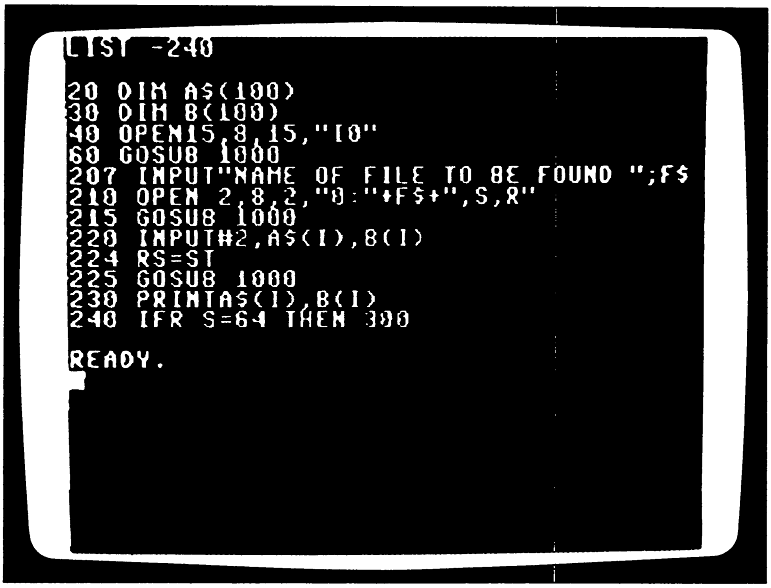
As it stands now the program will direct the computer to find and read into memory the items from a file on disk, named SEQ TEST FILE. By changing two statements, you can tailor the program to ask you for the file name, and then to find that file by name. The line that directs the computer to search the file for SEQ TEST FILE, line 210, can be changed so as to direct it to search out a file of any name, stored under the variable F$, like this:

```
210 OPEN 2,8,2,"0:" + F$ + ",S,R"
```

You can direct the computer to ask you for a file name, with the statement:

```
207 INPUT "NAME OF FILE TO BE FOUND"; F$
```

The program now in memory (Figure 9.7) will search a disk for the file

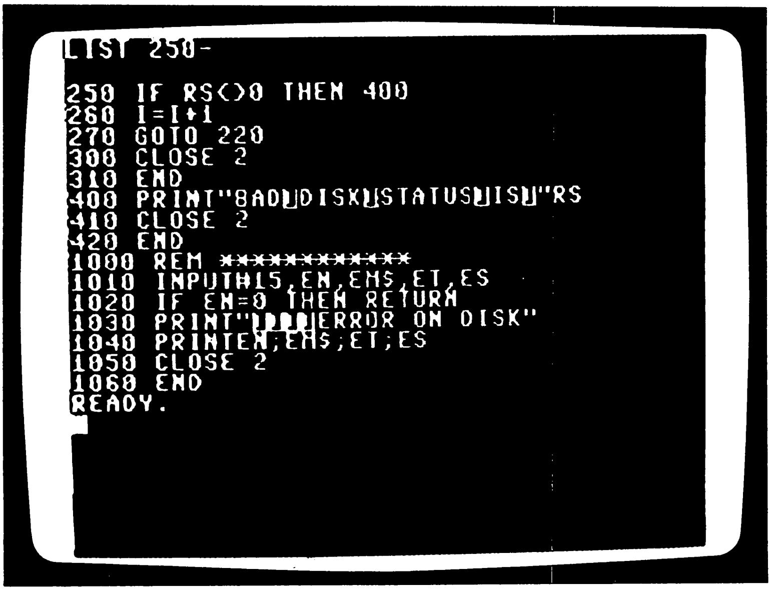


```
LIST -240

20 DIM A$(100)
30 DIM B(100)
40 OPEN15,8,15,"I0"
60 GOSUB 1000
207 INPUT"NAME OF FILE TO BE FOUND ";F$
210 OPEN 2,8,2,"0:"+F$+",S,R"
215 GOSUB 1000
220 INPUT#2,A$(I),B(I)
224 RS=ST
225 GOSUB 1000
230 PRINTA$(I),B(I)
240 IFR S=64 THEN 300

READY.
```

Figure 9.7: A tailored file-reading program, derived from the SEQUENTIAL FILE disk program.



```
LIST 250-

250 IF RS<>0 THEN 400
260 I=I+1
270 GOTO 220
300 CLOSE 2
310 END
400 PRINT"BAD DISK STATUS IS "RS
410 CLOSE 2
420 END
1000 REM ************
1010 INPUT#15,EN,EM$,ET,ES
1020 IF EN=0 THEN RETURN
1030 PRINT"ERROR ON DISK"
1040 PRINTEN;EM$;ET;ES
1050 CLOSE 2
1060 END
READY.
```

Figure 9.7, continued.

that you name in response to the computer's query, and then print out each element of the file as directed by line 230:

```
PRINT A$(I),B(I)
```

You can use this program to search for and list files, or you can further alter it to use those elements in different ways. You can add any statements you desire, so long as they take into account the form of the variables in the INPUT and allied statements. Conversely, you can change these variables to match your program if you like.

This approach to program tailoring can be used whenever you want the computer to act on information it receives from a disk file. By eliminating statements you find to be superfluous, you can streamline and adapt any prerecorded program the computer will list.

10 THE INSIDE STORY

To the person who has gained facility in using programs and commands, the computer is not the unmanageable contraption it often first appeared. Even though by learning to control the Commodore 64 you can gain an understanding of the logic behind the machine, many of its possibilities lie buried in its memory and design. There are ways these secrets can be uncovered; you can go further into the computer without finding yourself elbow-deep in microchips and tangled wires. More commands will lead you into the maze of number codes that direct the computer. An assortment of extra circuitry will make the Commodore 64 into something more than the machine it was built to be. And a look inside will reveal the guts of a personal computer undreamed of just a few years ago.

A STEP PAST THE TRANSLATOR

The computer's *translator* acts as a bridge between you and the *processor,* which directs the flow of information through the computer's circuits. These circuits are etched on microchips, like the one shown in Figure 10.1. The processor and the information reservoirs (the "memory chips") sit quite apart from each other in the computer. The computer's memory has been organized so that each piece of information is stored according to a number, which represents the location of that data. Although the processor may change the information in a memory location, that numbered location continues to exist, regardless of the value contained in it—a fixed address with changing tenants.

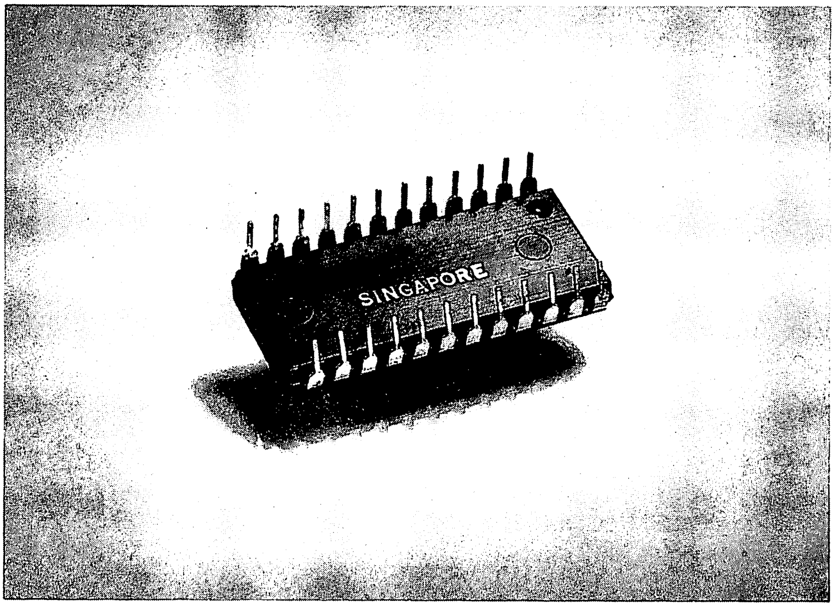


Figure 10.1: An encased microchip of the type used in all microcomputers.

The PEEK and POKE Commands

Although it expects commands in the computer language BASIC, the computer can also be directed to internal addresses. You can alter the computer's automatic responses by changing numeric values in these locations. One command will direct values into memory locations, circumventing the usual translating process.

Developing a mastery of the computer's own numeric language is a study, demanding and tedious, to which other books are devoted. Unless you have a passion for giving your computer directions that go beyond language commands like those in BASIC, it's a study not recommended.

The computer's numeric language is also expressed in a vocabulary of character codes, consisting of numbers and letters. These codes can represent every possible memory location and value in the usable memory of the Commodore 64. In this numeric language, you can send the computer to a particular memory location, A65E for example, where a set of directions for the SHIFT-HOME keystroke command tell the computer to clear the screen and position the cursor.

Programs of these encoded directions can form seemingly endless lists that blur the vision and dull the wits of all but the most determined enthusiasts. This system of notation is known as *hexadecimal*.

Fortunately, there is an easier way. There is a command available in BASIC that uses the more familiar decimal numbers. You can use it to change the computer's built-in drections to other directions you specify. This command is POKE, which sends the computer to a numbered location, where the numeric value you direct is changed.

To see how the command can be useful, consider an example of changing the screen display. The Commodore 64 automatically prints characters in light blue on a dark blue background surrounded by a light blue border. This screen display is determined by values copied into the active memory from the built-in unchanging memory every time you turn the power on. By changing the values in these active memory locations, you can change the printing pattern on the screen. One location sets the color of the window, another the color of the border.

The computer follows its own built-in directions in setting the background and border colors. By changing those directions, you can form displays of any color combinations. To do that, you can use the POKE command, which slips past the translator to talk more directly with the processor. You can command the processor to set a new color for the background by directing it to put another value in memory location 53281, where a screen color value for dark blue is automatically stored. If you want the color to be black, code number 0, you can use the command:

```
POKE 53281,0
```

This POKE command translates as: "Seek out memory location number 53281 and store the value of 0 at that address."

You can likewise direct the processor to set the border of the display to black. The processor looks for directions for this part of the display at memory address number 53280. You can use the command:

```
POKE 53280,0
```

In combination, these two commands and the values within them (background color 0, and border color 0) direct the computer to present a black background for the entire screen. They let you effectively seize control from the built-in automatic directions, and so switch the computer into a mode of operation tailored to your special needs. Any of the Commodore 64's 65,536 available memory locations will be opened by the POKE command for your new values.

As a tool for "customizing" and replanning the computer's performance, the POKE command in a program gives you absolute control, directing the computer to carry out changes before advancing to the next line.

You can use the PEEK command, which we discussed in Chapter 6, to look at the value in any memory location. It will direct the computer first to seek the numbered address you have given, as the POKE command does, but then to examine the value there and leave it unchanged. The value can then be used as part of some other command. For example, the computer finds the paddle-position values at memory locations 54297 and 54298, so a statement like:

```
PRINT PEEK(54297) TAB(20) PEEK(54298)
```

will display these values on screen. This combination of commands directs the computer in two stages, one command after the other. The PEEK commands send the processor to find the values, which the PRINT command then displays.

You will find the PEEK command most useful if you become familiar with the memory locations used by the processor to store directions. As the computer is sent commands to carry out, the values in memory locations change. With the PEEK command, you can see what those values are at any point, or use them to further direct the computer.

Some memory locations hold two values instead of just one. For these locations, a more complex form of the PEEK command is needed, to pry one value from another. These more complex forms require a detailed knowledge of the organization of the memory, and we won't discuss them here.

You can also use the memory-invading POKE and PEEK commands within programs, as you would other commands.

The SYS Command

Another command, SYS, sends the computer to specific memory locations, where it finds the directions that make each BASIC command work. For instance, the directions for the keystroke command RUN STOP-RESTORE begin at location A65E. You can send the computer to that location, where it will begin carrying out those directions, with the command:

```
SYS A65E
```

Driven by a SYS command, the computer will go to directions at any location in its memory. At some of these are automatically stored the little numeric subroutines that serve as directions for other BASIC commands. You can store your own machine-language routines at others. If you do that, keep in mind that the address marks the *beginning* of a routine. If, with a

SYS statement, you send the computer to an address in the middle of one of these sequences of directions, the results may be mangled. But after gaining some familiarity with these locations, you can avoid such surprises.

WHAT IS THE COMMODORE 64?

The Commodore 64 is based on Commodore's latest version of one of the microcircuitry chips that launched the personal computer. The circuit that gives the Commodore 64 its operating character is the 6510 processor. The 6510 is directed by same built-in instructions as its predecessor, the 6502, a processor chip found in the earlier VIC-20, Apple II, and Atari computers.

In fact, the same programs written in the BASIC computer language for the VIC-20 can be used in the Commodore 64, provided they don't refer through POKE, PEEK, and SYS commands to the different memory addresses in each, or to differences in screen display (the VIC-20 has a screen 22 columns wide by 24 rows deep). Furthermore, since the BASIC language consists of essentially the same elements in different computers, programs written for one computer can sometimes be adapted so as to be used in another.

As processor and internal foreman, the 6510 chip controls the flow of information through the other chips. A single circuit-board holds most of the electronics in the Commodore 64. On this board sit eight other chips, apart from the processor, as shown in Figure 10.2. Each of these chips holds 8192 (or 8K) locations of memory, which can be either occupied or empty according to the processor's directions. These eight chips together hold the Commodore 64's total adjustable memory of 65,536, or 64K locations.

Two other chips in this computer each perform special tasks. To one, the VIC chip, is given the duty of handling fine-point graphics and programmed illustrations, known as "sprites." These sprites, once stored in the computer's memory, can be positioned anywhere on the screen and made to interact with each other. Another chip handles the production of sound by the computer; with it, synthesized music and sound effects can be created.

To make use of either of these chips requires either elaborate programming of memory locations or the use of commercially prepared programs

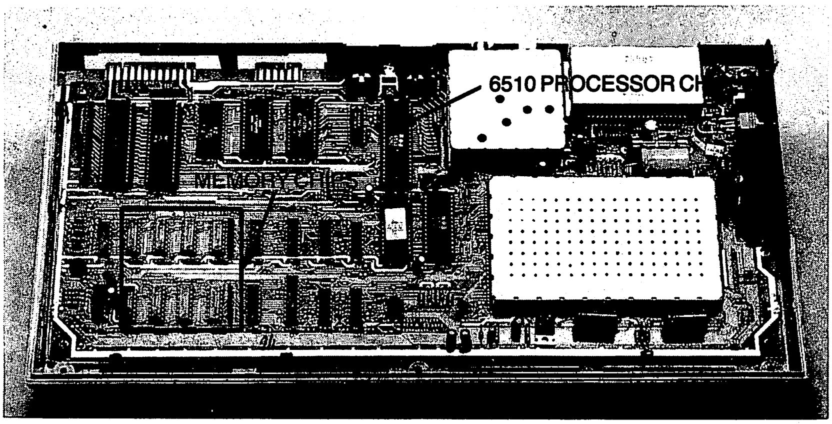


Figure 10.2: The processor and memory of the Commodore 64.

that translate simplified keyboard commands or paddle or joystick movements into changes in these memory locations.

EXPANDING THE COMPUTER

Your Commodore 64 wasn't finished at the factory. Look at the empty connector slot, sockets, and card at the back and power-switch side, and you'll see this. You can finish construction by adding circuitry cartridges and plugging in cables until the machine before you more closely matches your needs.

Some of the cartridges—containing "cards" packed with electronics—can change the way your Commodore 64 works. One crowds 80 characters of type on a video screen where only 40 would fit before. Another cartridge adds a different processor and its attendant chips, and so grafts a second computer into the the Commodore 64. Still another sends information and programs out from the computer into telephone lines and distant receivers. Some cables and circuitry route the screen display to printers, which can turn the video screen images into print on paper.

And, of course, there are game cartridges.

The equipment you add to the computer can be combined to produce a computer system of your own preference. The 80-column cartridge is one of several links to *word processing,* a puffed-up phrase that simply means moving words around in a computer's memory, instead of on a sheet of paper.

Another link in the chain, a set of directions for word-processing commands, comes most conveniently on disk in the form of commercially available word-processing programs. If the words so processed are to eventually take physical form as ink on paper, a printer will be at the end of the chain. The link that completes the connection between printer and computer takes the form of a cable—and, depending on the printer, an extra bit of circuitry attached to that cable. These circuits act as gateways between printer and computer.

The Commodore 64 operates under the directions built into its 6510 processor chip, by taking commands and programs that "speak to" that chip. Another processor chip, the popular Z80, has been well provided with available programs. By plugging in a cartridge containing the Z80 and its entourage of circuits, you can put a second processor into the Commodore 64. This one can use the Commodore 64's memory and can take commands from the keyboard or disk. A Commodore 64 so equipped can be switched back and forth between the resident 6510 chip and the Z80 chip, to run programs written for either processor. In effect, you'll be switching between two computers in one box.

The 6510 chip processes commands and information according to directions contained in the Commodore 64's permanent memory. Another set of directions for controlling processors (known as an "operating system") is often fed into chips that accept it, like the Z80, from a disk. This system is known by the acronym CP/M® and, as a tool for controlling the processor's operation, has become very popular. In fact, many programs written on different machines using the CP/M operating system are transferable from one type of computer to another with no or few revisions. With a Z80 cartridge installed in the Commodore 64, you can load CP/M into the computer. For the Commodore 64, CP/M is packaged as a card containing a Z80 chip, along with a diskette.

Another cartridge, the modem, plugs into the Commodore 64 and forms a link to telephone lines. With a modem installed, your computer becomes an electronic teletype, connecting you to people and data banks in ways that were once the exclusive privilege of large corporations and governments.

Perhaps the most promising feature of a personal computer like the Commodore 64, however, is the adaptability of its *master* to the possibilities of this newly-arrived tool. The computer is a means to an end; once you've learned to direct it, you can use it as an extension of your own interests and ideas. You need not study the electronics or logic of the chips that direct the machine to realize your own capabilities in using it. Rather, you will add one capability to another, to build a truly personal computer system, one that is directed by your own human intelligence.

A
SPECIAL-PURPOSE COMMANDS

The built-in vocabulary of the Commodore 64 includes commands of more specialized usefulness than those presented in the preceding chapters. These commands mostly involve mathematics and number treatment. Many of them are grouped below according to function. An underline indicates where you can (and usually must) include a number or variable in each command.

Mathematical and Trigonometric

ABS (___): Turns a negative number positive, leaves positive number unchanged.

ATN (___): Gives the trigonometric value arctangent of a number given in radians.

COS (___): Gives the trigonometric value cosine of a number given in radians.

EXP (___): Gives the value of *e,* 2.7182818, raised to the power of a number.

INT (___): Gives the value of rounding down to the nearest integer number.

LOG (___): Gives the logarithm of a number.

RND (___): Gives a random value between 0 and 1.0, regardless of number.

SGN (__): Gives the sign of a number: − 1 if number is negative, + 1 if positive, 0 if 0.

SIN (__): Gives the trigonometric value sine of a number given in radians.

SQR (__): Gives the square root of a number.

TAN (__): Gives the trigonometric value tangent of a number given in radians.

__^__: Raises the number before it to the power of the number after it.

__E__: Multiplies the number before it by 10 to the power of the number after it.

FN __ (__): Gives a number according to the mathematical relation established with a DEF command. See DEF FN. FN Z(A), FN Z(B) and FN Z(5), for example, each call on the same relation set as FN Z(__).

DEF FN __(__) = : Defines a function; that is, it establishes a mathematical equivalence between the variable in parentheses and the mathematic relation after the equal sign. For example, DEF FN X(A) = A + B establishes a function FN X(__), which adds the value of B to the value or variable in parentheses.

Comparisons

Comparison operators produce values that correspond to the truth or falsehood of the statement made with them. When the statement is true, a value of − 1 is given; when it's false, a value of 0. As their name implies, the kind of statement these operators make is always a comparison of values.

__ > __: (Greater than) Gives a value of − 1 if the number before it is greater than the number after it, 0 if otherwise.

__<__: (Less than) Gives a value of − 1 if the number before it is less than the number after it, 0 if otherwise.

__ = __: (Equal to) Gives a value of − 1 if the number before it is the same as the number after it, 0 if otherwise.

__<> __ or __> <__: (Not equal to) Gives a value of − 1 if the number before it is not the same as the number after it, 0 if it is.

__> =__ or __=<__: (Greater than or equal to) Gives a value of −1 if the number before it is greater than or equal to the number after it, 0 if it is less.

__<=__ or __=<__: (Less than or equal to) Gives a value of −1 if the number before it is less than or equal to the number after it, 0 if it's greater.

Logic (Boolean)

__AND__: Gives a value of 1 if the number or relation before it and the one after it are both greater than 0, 0 if one is 0.

__OR__: Gives a value of 1 if either the number or relation before it or the one after it is 1 and the other is 1 or 0, 0 if they are both 0.

NOT__: Gives a value equal to the number or relation plus 1, multiplied by −1.

Other

ASC ("__"): Gives a code number, called the ASCII code, for the first character of the string of characters.

CLR: "Clears" all the variables in a program; replaces their values with 0.

STR$ __: Directs the computer to treat a number as a string character, instead of a numeric value.

VAL__$: Directs the computer to treat a numeric string character as a number.

B JARGON PHRASE GUIDE

Jargon is the curse of the specialized, and every specialty has its jargon. You will encounter computer jargon among enthusiasts, salespeople, and programmers. Jargon speakers, who are often more knowledgeable, can provide useful information not obtainable from speakers of the mother tongue. The phrase guide that follows can help in making their dialect intelligible.

Applications program: A set of commands, usually available on disk, that directs the computer in performing some task whose results appear in a recognizable form outside the computer's memory.

Back-up: Not directions given to a truck driver, but a copy kept in case the item in use should fail. A back-up copy of a disk can be kept, or a back-up copy of a file or program can be kept in another form or on the same disk.

Bit: Not the past tense of bite, but the smallest piece of information the computer can handle. A bit always has a value of either 0 or 1.

Boot a Disk: Not an act of aggravation, but the act of sending directions that operate a disk system into the computer from a disk in a drive. (You boot the disk before sending disk commands to the computer.)

Boot DOS: See *Boot a Disk.*

Buffer: A specially controlled memory that will hold data or commands being moved between the computer and other electronic devices. It's useful when the rate of flow out of one is greater than the rate into the other, like a reservoir collecting a river's waters faster than it lets them through flood gates.

Bug: Not a crawling six-legged creature, but any unplanned response that interferes with the expected operation of a machine. The term usually refers to an unplanned command problem within a computer program.

Byte: The basic package of information sent through the computer circuits, in which each of eight bits has a value of either 0 or 1. Each memory location in the Commodore 64 holds one byte.

COBOL: A programming language in widespread use in business since before the rise of smaller, personal computers.

Copy-protected: Describes a program on a disk that includes directions designed to thwart any efforts to copy that program onto another disk.

CPU: Central Processing Unit, the processor in the computer.

Crash: In a program run, what happens when the computer stops acting on program commands suddenly, in an unplanned way, usually because of directions the computer can't follow.

Data-base Manager: Not a baseball coach, but a program that organizes information according to a preset plan.

Default ("dear Brutus, lies not in our stars but in ourselves"): That value chosen automatically by the computer when a choice is not made by the computer user.

Enter: To send data or instructions into the computer, usually from the keyboard, and followed by a subsequent press of the RETURN key.

Error: Any command or data given to the computer that it is incapable of acting on.

Escape Sequence: The use of keyboard typing to stop the running of a program or part of a program.

Fatal Crash: A *crash* that results in the loss of program commands or valuable information from computer memory.

Firmware: Those directions stored in built-in memory that can't be changed from the keyboard. (Compare with *hardware* and *software.*)

Format: To etch a magnetic pattern onto a disk, so that programs and data can be stored on it.

FORTRAN: A computer programming language used by scientists and engineers since the days of punch cards.

Function: A command that manipulates values, often in some unseen way. The commands +, −, *, and / represent arithmetic functions.

Hardware: The physical parts of a computer system, as opposed to the directions and values stored in it. (Compare with *firmware* and *software.*)

Hex: Not a witch's curse, but an abbreviation for hexadecimal, a way of representing numbers by a hybrid of the first six (hence hex) letters of the alphabet and the ten (hence decimal) numerals.

High-level Language: Not the King's English, but a set of commands that direct the computer in relatively complicated ways to perform a complex task with a single command. BASIC is a high-level language.

Housekeeping: Not dusting and mopping, but the chores done by the computer user, or the computer itself, to operate a system or a program.

Initialize: To format a disk and add an initial program of commands that will be automatically run by the computer when the disk is first scanned.

Instruction: Any command or direction given to the computer.

Interface: As a noun, the circuitry that serves as electronic connection to the computer, or a method of commands that connects different logic systems. As a verb, to connect.

I/O: The flow of data or commands to and from the computer, from Input/Output.

K: When referring to the number of memory locations, an amount roughly equal to a thousand (its exact value is 1024). 64 K is equal to 65,536.

Logo: A programming language designed for ease of use and learning, often a child's introduction to computers.

Loop: A series of commands through which a computer is directed repeatedly.

Low-level Language: Not locker room talk, but a set of commands that direct the computer in relatively unsophisticated ways, requiring several commands to direct a complex task.

Machine language: The codes for memory locations particular to any computer.

Menu: Not a selection of entrees at a restaurant, but a list of choices given in a program, from which its user selects different operations. Programs using them are called *menu-driven.*

Microcomputer: A computer built around a microprocessor. The Commodore 64 is a microcomputer.

Microprocessor: The electronic circuitry chip that controls the rest of the computer. The 6510 chip is the Commodore 64 computer's microprocessor.

Mini-Floppy Diskette: The formal name for the type of disk, 5¼ inches across and flexible, used with the Commodore 64 and most other personal computers.

MMU: (Memory Management Unit) The part of the computer that organizes the flow of information into different parts of memory according to a preset pattern.

Modem: A device that couples a computer to telephone lines.

Parallel: Travelling concurrently, as in the way data or command impulses can be sent from a computer through a set of parallel wires. (Contrast with *serial.*)

Pascal: A computer programming language that is particularly popular in academic circles.

PC Board: (Printed-Circuit Board) A hard plastic or fiber board on which strips of bonded foil serve as electrical pathways in place of wires, and on which electronics are seated.

Peripheral: Any device attached to a computer that controls it, or is controlled by it.

Pixel: Not a mincing dwarf, but the smallest dot a computer can control on a video screen.

Powerful Language: Not epithets, but a set of commands, each of which can give directions that would require several commands in a "weaker" language.

Powerful Program: A program that does a lot with little prompting or control by its user.

RAM: An acronym for Random-Access Memory, the computer memory locations whose values can be changed from the keyboard.

Read: To draw information from an encoded form, such as a disk.

Return a Value: What the computer does when it displays, on screen or printer, numbers or characters in response to commands.

ROM: An acronym for Read-Only Memory, the computer memory locations with values fixed in manufacture. It provides all the computer's built-in directions.

Serial: One after another, as in the way data or command impulses can be sent from a computer through a single wire. (Contrast with *parallel.*)

Software: The directions, such as those available as programs on disk, that can be added to and removed from the computer system, as opposed to its physical, fixed parts. (Compare with *firmware* and *hardware.*)

Spreadsheet: Not bed linen hanging to dry, but a visual system of organizing and calculating data, often business or financial, in rows and columns. Several such systems are available as programs on disks.

Utility: Not the gas company, but a program that directs the computer in performing some internal task, like moving data about its memory.

Write: To put information in a form in which it can be stored and then retrieved, as in to write to a disk.

Write-enable Notch: A section of the disk envelope that, when uncovered, allows the disk drive to add or alter information on the disk.

Write-protection: The physical feature of a disk that prevents it from having information added or altered. The absence of, or taping over of, a notch on the disk envelope results in write-protection.

INDEX

FOR YOUR COMMODORE 64!

THE EASY GUIDE TO YOUR COMMODORE 64

This is the book that will show you how to use your Commodore 64 in a matter of hours. Right away you will become familiar with the keyboard, video screen, and add-on devices.

You can learn how easy it is to write your own BASIC programs. Or, if you prefer, you can skip programming and learn how to get started with commercially available software.

You will quickly learn to:

- Create video graphics
- Use your disk drives
- Make forecasts and simulations
- Customize pre-recorded programs to your needs
- Expand your system with useful accessories
- And much more

Everything you need to know to get started with your Commodore 64 personal computer is revealed in a friendly, jargon-free style.

ABOUT THE AUTHOR:

Joseph Kascmer is a professional writer whose works include articles and instructional materials in health sciences, anthropology, politics, regional history, and computers in business and education.

ISBN 0-89588-126-8