A SPECTRUM BOOK

COMMODORE 64™
VIC 20™
BASIC

RICHARD HASKELL and THOMAS WINDEKNECHT

Haskell/Windeknecht
COMMODORE 64™/VIC 20™ BASIC

CONTENTS

PREFACE

Anyone planning to teach a BASIC programming course using Commodore 64™/VIC 20™ personal computers is faced with the problem of selecting an appropriate text. Programming manuals provided by the manufacturer are generally intended for reference rather than for teaching purposes. Standard textbooks on BASIC programming will describe a BASIC language that varies enough from Commodore 64/VIC 20 BASIC to lead to considerable frustration on the part of the student. In addition, such texts will be of no help in learning to use the graphics capability of the Commodore 64/VIC 20. This is unfortunate, since most interesting programs on the Commodore 64/VIC 20 involve graphics.

This book is designed to be used as a text for learning to program in BASIC using a Commodore 64/VIC 20. It is suitable for introductory programming courses at the high school, junior college, and university levels. It can also be used for self-study with a Commodore 64/VIC 20 computer.

Companion texts entitled *Apple BASIC, TRS-80 Extended Color BASIC, Atari BASIC, PET/CBM BASIC, TI BASIC, and IBM PC BASIC Programming* are also available for use with these computers.

The strategy of this book is "learning by doing." The student is led step by step through all aspects of BASIC programming on a Commodore 64/VIC 20. All examples are illustrated with actual photographs of a Commodore 64/VIC 20 screen. Many of the fundamental programming ideas are developed using examples involving graphics. This has the advantage of providing a direct visual picture of what the program is doing. In addition it provides examples that will be useful to anyone wishing to write Commodore 64/VIC 20 programs for specific applications.

Chapter 1 introduces the Commodore 64/VIC 20 keyboard and the drawing of simple graphic figures. Getting the Commodore 64/VIC 20 to draw figures by using the PRINT statement with string variables is accomplished in Chapter 2. Chapter 3 discusses the general nature of BASIC programs and the operation of the cassette tape recorder and floppy disk drive.

Chapter 4 covers numerical variables, arithmetic expressions, Commodore 64/VIC 20 built-in functions, and more graphics. The INPUT statement is covered in Chapter 5, with examples including the drawing of custom checkerboard patterns. Chapter 6 introduces the IF...THEN statement and relational and logical operators. The checkerboard patterns now become random.

The important topic of loops is introduced in Chapter 7 and continued in Chapter 8, where the American flag is displayed on the screen. The use of

subroutines is covered in Chapter 9, where the student learns to produce his or her name in three-dimensional letters on the screen.

The READ...DATA statement is introduced in Chapter 10, where the drawing of bar graphs is covered. Chapter 11 describes the GET statement and uses it to draw pictures interactively on the screen and to move a fighter plane around the screen and fire the plane's phasers.

Arrays and the ON...GOSUB statement are discussed in Chapter 12. String functions are described in detail in Chapter 13, with examples given for dealing a hand of cards. Chapter 14 describes the use of the PEEK and POKE statements and includes more graphics, the use of lower-case letters, and the production of sound. Chapter 15 describes the development of two complete programs: the hangman word game, and a Commodore 64/VIC 20 organ that plays three octaves of musical notes.

Students who complete this text will gain a solid foundation in fundamental programming techniques and will acquire the particular skill of being able to program the Commodore 64/VIC 20 computers using BASIC. Special thanks go to Sharon Rix, who typed the manuscript with skill, patience, and good humor.

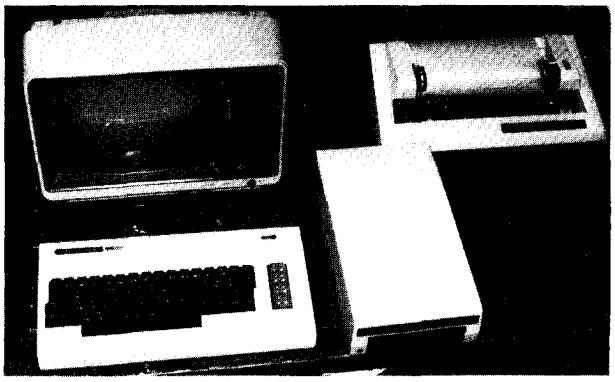
1 LEARNING TO USE THE COMMODORE 64/VIC 20 KEYBOARD

There is only one way to learn how to program a computer: you must write programs and run them on the computer. It is not possible to learn to program by reading about it. Programming is an action activity. You must do it. This book is designed to help you learn how to program in the BASIC programming language while using a Commodore 64 or VIC 20 computer.

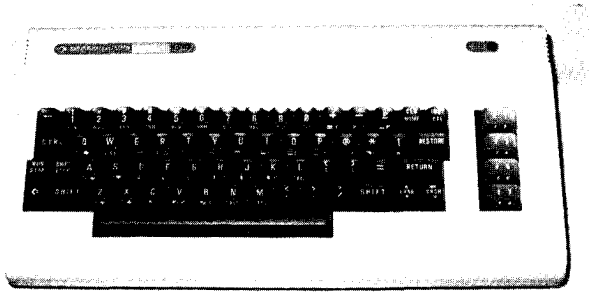
The VIC 20 and Commodore 64 computers are two of several popular personal computers (some others are the Apple II, the TRS-80, and the Atari) that are finding their way into an increasing number of homes and schools. All of these personal computers will run programs written in the BASIC programming language. However, the BASIC programming language is implemented somewhat differently on each of these computers, particularly with respect to graphics programming. This means that a BASIC program written for an Apple II computer will not, in general, run on a Commodore 64 or VIC 20 without some modification. (Fortunately, the Commodore 64 and VIC 20 are rather similar and fewer modifications are required to convert programs back and forth). If you are learning BASIC for the first time, it will be easier for you if you use a book written specifically for the kind of computer you are using. In this way you will not become frustrated by all of the little "exceptions" that apply only to your computer.

This book assumes that you have a Commodore 64 or VIC 20 computer available for your use. These two computers are shown in Figure 1.1. Almost all of the

Figure 1.1 (a) VIC 20.



(b) Commodore 64.



programs described in this book can be run on either computer. In this chapter you will become familiar with the use of the Commodore 64/VIC 20 keyboard. In particular you will learn the meaning of the special keys shown in Figure 1.2, and you will learn how to draw simple graphic figures in various colors.

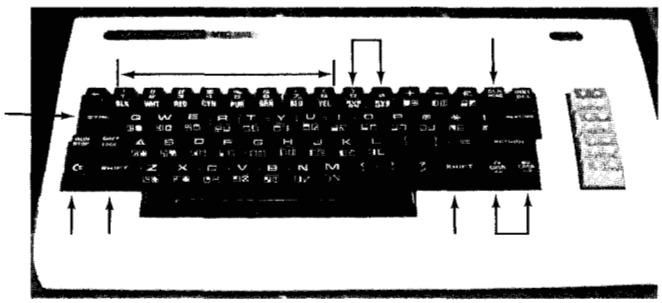
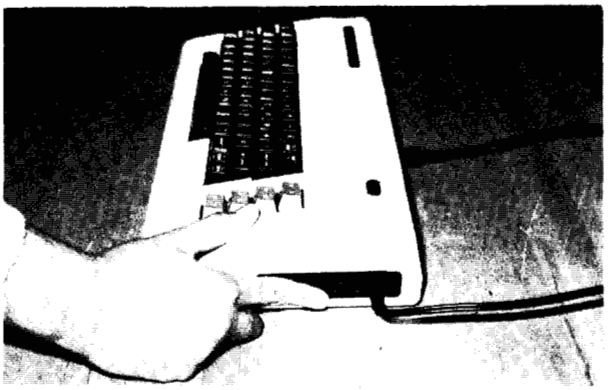


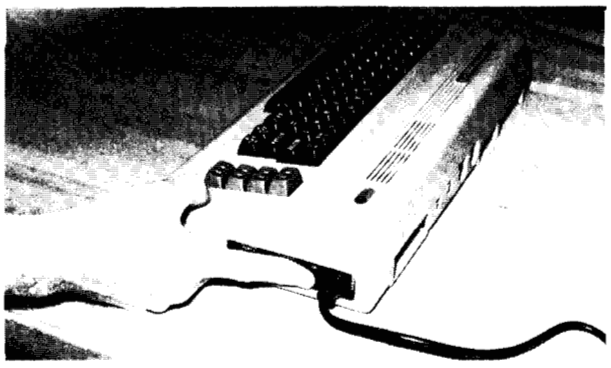
Figure 1.2 Special keys discussed in this chapter.

Begin by turning on your Commodore 64/VIC 20. This is done with the switch on the right side of the computer (see Figure 1.3). After a few seconds of warm-up the screen should look either like Figure 1.4a or 1.4b. Note the difference between the two screens. The VIC 20 produces fewer characters per row and column and, hence, each character appears larger on the screen. The VIC 20 actually produces blue

Figure 1.3 (a) Turning on the VIC 20.



(b) Turning on the Commodore 64.



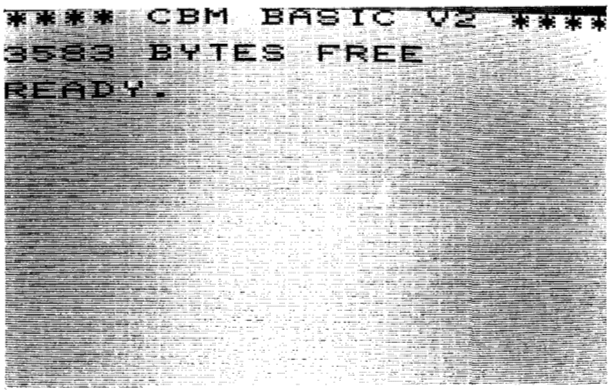
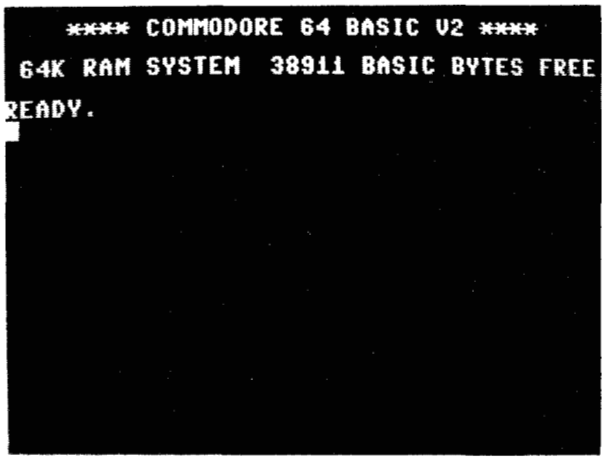


Figure 1.4 (a) Screen display for a VIC 20.



(b) Screen display for a Commodore 64.

characters on a white background and the border color is cyan. The Commodore 64 produces light blue characters on a blue background and the border color is also light blue. In the black-and-white photographs in this book, the VIC 20 screen appears to be black characters on a white background (like typewritten material) whereas the Commodore 64 screen is reversed, white characters on a black background. Most of the photographs we shall use are of the Commodore 64 screen. We do this because they give a higher quality of reproduction.

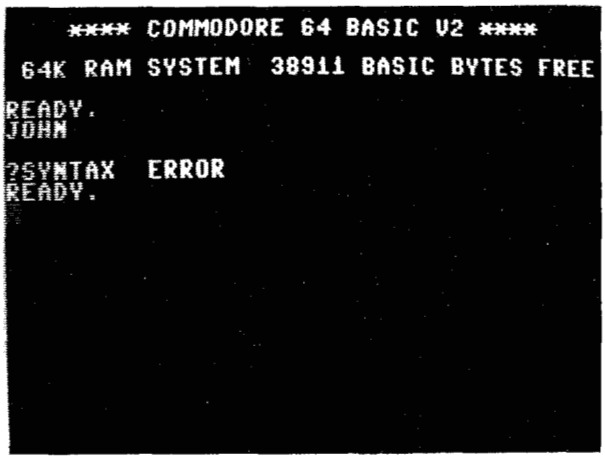
Your Commodore 64/VIC 20 computer contains a special read only memory, called a ROM. The ROM contains the BASIC interpreter program, which is ready to run when you turn on the power. In addition to the read only memory (ROM) your computer contains some read/write memory called RAM, or random-access memory. ROM is also random-access memory, but the difference between ROM and RAM is that you can change the contents of a RAM location, while the contents of a ROM location are fixed and cannot be changed. In addition, when you turn off the computer's power the contents of the RAM locations are lost, while the contents of the ROM locations are

retained. This is why the BASIC interpreter, located in ROM, is there every time you turn on your computer. On the other hand every program that you write is stored in RAM and is lost whenever you turn the power off. This is why you must save your programs on cassette tapes or floppy diskettes if you wish to run them again without having to type them in.

The amount of RAM that you have depends on which computer you are using. The more RAM you have, the larger the programs you can run and the more data you can store in the computer. A VIC 20 without a memory expansion contains 5,120 bytes of RAM. (A byte is eight bits; a bit is a 1 or a 0. Thus, for example 10101101 is a byte. It takes one byte to store a character. 1K = 1,024 bytes.) When you turn on a 5K VIC 20, you should see the message 3583 BYTES FREE at the top of the screen (see Figure 1.4a). This is the amount of memory available for you to use. The space used by the VIC 20 for its BASIC interpreter and other built-in programs accounts for the difference between 5,120 and 3,583. A Commodore 64 contains 65,536 bytes of RAM and the message 38911 BASIC BYTES free should appear when you power up the computer.

The last word that is displayed when you turn on your Commodore 64/VIC 20 (see Figure 1.4) is the word READY with a *blinking* cursor displayed beneath it. The cursor identifies where your typing will begin to appear on the screen. Try typing your name, and then press the RETURN key. If your name is JOHN, your screen should look like the one shown in Figure 1.5.

Figure 1.5 A SYNTAX ERROR occurs when you type an invalid BASIC command.



Note that the message **?SYNTAX ERROR** appears on the screen. This is because JOHN is not a valid BASIC command, and the computer can respond only to BASIC commands that it understands.

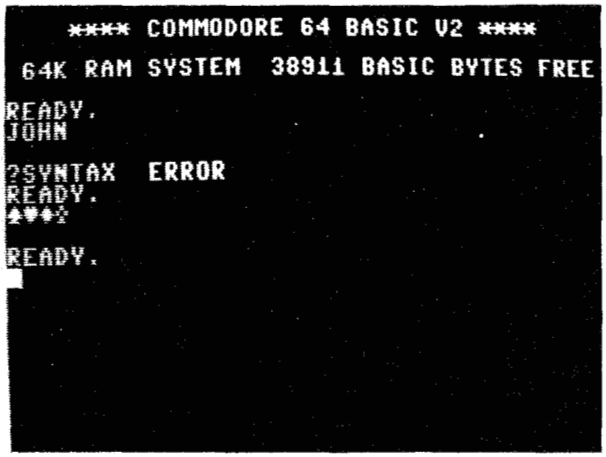


Figure 1.6 Playing card symbols.

All of the valid BASIC commands are described in this book. Whenever you type a misspelled or otherwise invalid command, the Commodore 64/VIC 20 will respond with a **?SYNTAX ERROR** message and will allow you to type in a correction.

GRAPHIC KEYS

On the front face of most keys on the keyboard are two graphic symbols. You can type these symbols on the screen by holding down either the LOGO key (at the lower left with the Commodore logo on it) or a SHIFT key (there are two and both function in the same manner) while pressing the particular graphic key. The LOGO key causes the left graphic symbol to be printed whereas the SHIFT key causes the right graphic key to be printed. For example, if you press keys A, S, Z, and X in sequence while holding down the SHIFT key, you will display the spade, heart, diamond, and club card symbols, as shown in Figure 1.6.

Note that when you press RETURN after typing only graphic symbols you do not get a **?SYNTAX ERROR** message. Try typing some of the graphic symbols to see what they look like.

CURSOR KEYS

There are three keys, one at the upper right and two at the lower right, that are used to control the movement of the cursor. These three keys, shown in Figure 1.7, are the CLEAR/HOME, CURSOR UP/DOWN, and CURSOR LEFT/RIGHT keys. The CLEAR/HOME is at the upper right and the two CURSOR keys are at the lower right.

Pressing the HOME key causes the cursor to move to the upper left-hand corner (home position) of the screen. Pressing this same key while holding down the

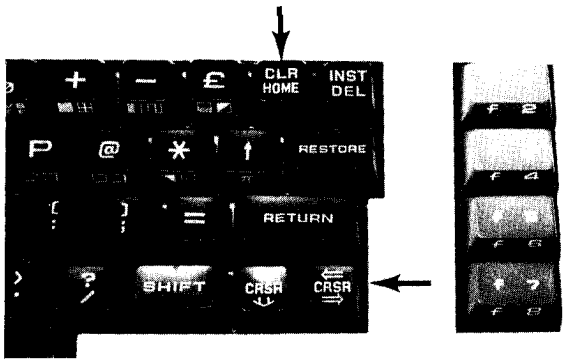


Figure 1.7 The CLEAR/HOME and cursor keys.

SHIFT key will return the cursor to the home position and clear the screen. Try out this key.

Pressing the CURSOR UP/DOWN key will cause the cursor to move down. Pressing this key while holding down the SHIFT key will cause the cursor to move up. Try this. Note that if the cursor is on the top line of the screen, attempting to move it up has no affect. On the other hand, if the cursor is on the bottom line of the screen (line 25 on the Commodore 64/line 23 on the VIC 20), attempting to move it down causes the entire screen to scroll up, with the cursor remaining on the bottom line. In order to see this, type anything on several lines of the screen and then continue to press the CURSOR UP/DOWN key.

Pressing the CURSOR LEFT/RIGHT key will cause the cursor to move to the right. Pressing this key while holding down the SHIFT key will cause the cursor to move left. Try this. Note that if the cursor is in the left-most column on the screen (column 0), attempting to move it left will cause the cursor to move to the right-most position (column 39 on the Commodore 64/column 21 on the VIC 20) on the preceding line. (If the cursor is in the home position, nothing will happen). Note also that, if the cursor is in the right-most column on the screen (column 39 on the Commodore 64/column 21 on the VIC 20), attempting to move it to the right will cause the cursor to move to the left-most position (column 0) of the next line. (If the cursor is at the right-most position of the last line on the screen, attempting to move it to the right will cause the screen to scroll).

MAKING GRAPHIC FIGURES

The graphic keys and the cursor keys can be used together to form graphic figures. For example, clear the screen and then move the cursor down near the center of the screen. Now press the following keys. (The notation SHIFT U means press the U key while holding down the SHIFT key).

SHIFT U
SHIFT I
CURSOR DOWN
CURSOR LEFT
CURSOR LEFT
SHIFT J
SHIFT K

This sequence generates a circular figure as shown in Figure 1.8.

Other shapes can be made using other graphic keys. For example, the larger square in Figure 1.9 is made by using the right graphics characters on keys O, P, L, and @. The smaller square is made by using the left graphics characters on keys A, S, Z, and X. This requires pressing the LOGO key rather than a SHIFT key. The diamond is made by using the right graphics characters on keys N and M. Try making these figures.

REVERSE VIDEO

The RVS ON, RVS OFF, and CTRL keys shown in Figure 1.10 are used to turn the reverse video on and

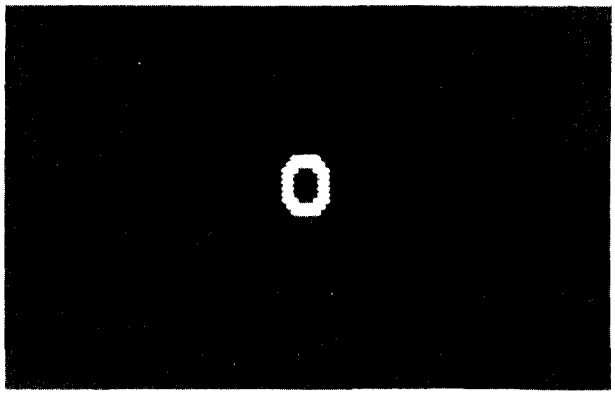
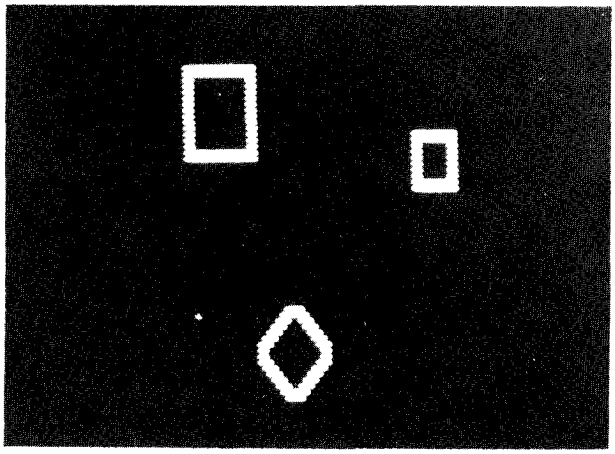


Figure 1.8 Circular figure generated using keys U, I, J, and K.

Figure 1.9 Large square generated using O, P, L, and @. Small square generated using keys A, S, Z, X. Diamond generated using keys N and M.



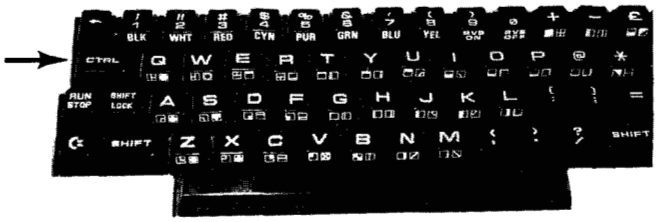


Figure 1.10 Reverse video keys & the CTRL key.

off. With reverse video, white characters appear on a blue background on the VIC 20 and blue characters appear on a light blue background on the Commodore 64. Clear the screen, press the RVS ON and CTRL keys simultaneously, and then type **THIS IS REVERSE VIDEO.** The result should be as shown in Figure 1.11. To turn the reverse video off, press the RVS OFF and CTRL keys simultaneously.

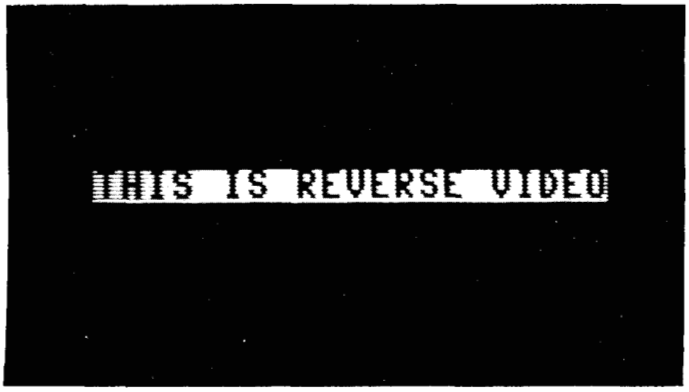
The reverse video keys can be useful when making certain graphic figures. For example, the graphic figure in Figure 1.12 can be made by using the following keys:

SHIFT POUNDSIGN (£)
LOGO *
CURSOR DOWN
CURSOR LEFT
CURSOR LEFT
CTRL RVS ON
LOGO *
SHIFT POUNDSIGN(£)

USING THE COLOR KEYS

On the VIC 20, the characters are normally blue on a white background and, on the Commodore 64, they

Figure 1.11 Example of reverse video.



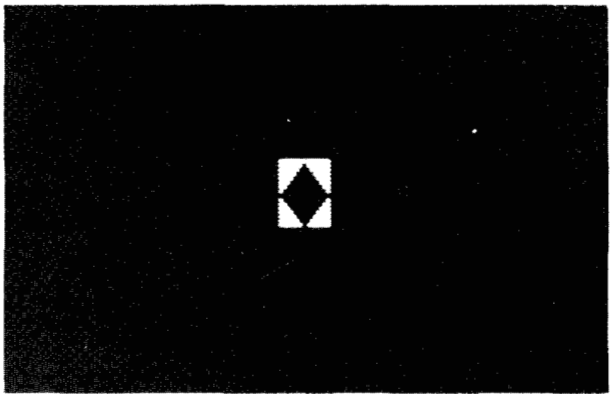


Figure 1.12 Graphic figure generated using reverse video.

are light blue on a blue background. On a color TV, however, the VIC 20 can produce characters in eight different colors and the Commodore 64 can produce characters in 16 different colors. Eight colors on both the VIC 20 and Commodore 64 can be chosen by pressing CTRL together with one of the numeric keys 1 through 8. See Figure 1.13. For example, if you press CTRL and 4 together, you will see the cursor change to cyan. Now, if you type in some characters, they too will appear in cyan. Try it.

The eight colors produced by the numeric keys and CTRL are shown in Figure 1.14. To see them, clear the screen and position the cursor about 1/3 of the way down from the top. Then, type in the following keys:

CTRL RVS ON
CTRL BLK, SPACE, CURSOR DOWN
CTRL WHT, SPACE, CURSOR DOWN
CTRL RED, SPACE, CURSOR DOWN
CTRL CYN, SPACE, CURSOR DOWN
CTRL PUR, SPACE, CURSOR DOWN
CTRL GRN, SPACE, CURSOR DOWN
CTRL BLU, SPACE, CURSOR DOWN
CTRL YEL, SPACE, CURSOR DOWN

You can erase the screen and return the character colors to normal by pressing STOP and RESTORE.

Figure 1.13 The eight color keys.

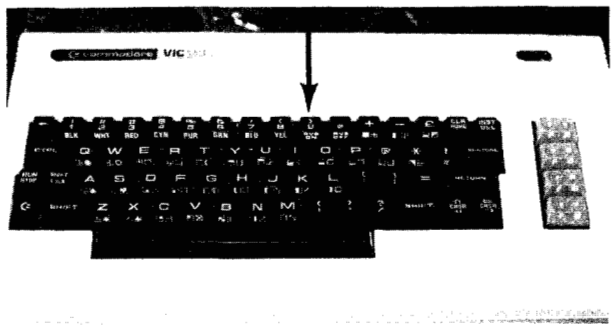


Figure 1.14 The numeric keys and their colors with CTRL.

Numeric Key	Color
1	black
2	white
3	red
4	cyan
5	purple
6	green
7	blue
8	yellow

If you have a Commodore 64, you can get eight additional colors by pressing the LOGO key together with one of the numeric keys 1–8. The colors you get are shown in Figure 1.15. To see them on the screen, repeat the previous sequence of keyboard entries but substitute LOGO color in place of CTRL color. To get back to normal color on the Commodore 64, you can enter LOGO and blue together (to get light blue characters) as well as STOP and RESTORE.

The color keys can be used with the graphics keys and reverse video keys to produce color graphics. For example, clear the screen, position the cursor about 1/3 down from the top, and type in the following keys:

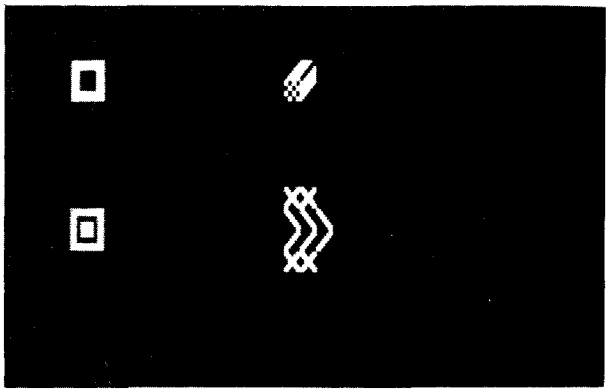
Figure 1.15 The number keys and their colors with the LOGO key (Commodore 64 only).

Numeric Key	Color
1	orange
2	brown
3	light red
4	gray 1
5	gray 2
6	light green
7	light blue
8	gray 3

CTRL CYAN
SHIFT M, SPACE (5 times), SHIFT N
CURSOR DOWN, CURSOR LEFT (6 times)
SHIFT M, SPACE (3 times), SHIFT N
CURSOR DOWN, CURSOR LEFT (4 times)
CTRL RVS ON, SHIFT M, SPACE, SHIFT N
CURSOR DOWN, CURSOR LEFT (4 times)
SHIFT U, SHIFT I, SPACE, SHIFT U, SHIFT I
CURSOR DOWN, CURSOR LEFT (5 times)
SHIFT J, SHIFT K, SPACE, SHIFT J, SHIFT K
CURSOR DOWN, CURSOR LEFT (3 times)
SPACE, CURSOR DOWN (2 times)

What does this graphics picture remind you of? A bug? A spacecraft? Something else?

Figure 1.16 Graphic figures for Exercise 1-1.



EXERCISE 1-1
Try to generate the graphic figures shown in Figure 1.16.

EXERCISE 1-2
Repeat Exercise 1-1 but make each graphic figure a different color.

SIMPLE GRAPHICS: LEARNING TO USE THE PRINT STATEMENT

In Chapter 1 you learned how to make simple graphic figures by using the graphic keys and the cursor keys. Although this method is fairly straightforward, it is too time-consuming to use when drawing complicated figures. In this chapter you will learn:

1. how to pre-store graphic and cursor key moves so that the computer can draw graphic figures very quickly
2. how to use the PRINT statement
3. what strings and string variables are
4. the difference between the immediate and deferred modes of execution
5. to use the LIST and RUN commands
6. to edit a statement using the cursor keys and the INSERT/DELETE key.

STRINGS AND THE PRINT STATEMENT

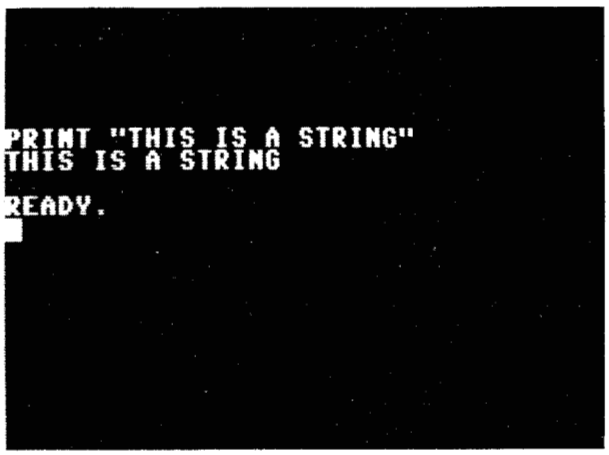
Clear the screen and type **PRINT "THIS IS A STRING"** and then press RETURN. The result should look like Figure 2.1. Note that the computer immediately printed the words **THIS IS A STRING.**

Any sequence of characters enclosed between quotation marks (" ") is called a *string*. If you type the word **PRINT** followed by a string, the computer will immediately print the string (without the quotation marks) on the screen. This is called the *immediate mode* of execution.

A string may include the *graphic* characters. For example, try printing the playing card symbols as shown in Figure 2.2.

A string may also include the *cursor* moves. This may seem a little strange at first, but this is what allows you to store an entire graphic figure as a string. When you press one of the cursor keys while you are typing a string, the cursor does not move. Instead, a special character is shown in the string. Later, when this string

Figure 2.1 Using the PRINT statement in the immediate mode of execution.



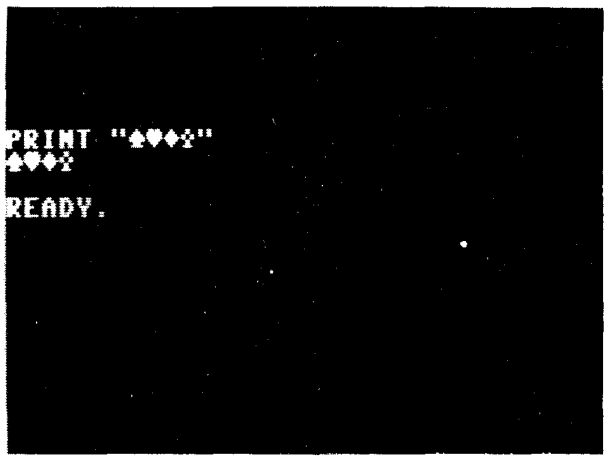


Figure 2.2 Printing graphic symbols using the PRINT statement.

is PRINTed, the cursor movements will occur in the order in which they were specified in the string.

For example, if you want to PRINT the circular figure shown in Figure 1.8 in the previous chapter, include these key strokes as a string in a PRINT statement:

SHIFT U
SHIFT I
CURSOR DOWN
CURSOR LEFT
CURSOR LEFT
SHIFT J
SHIFT K

The result will look like Figure 2.3. Try it.

Figure 2.3 PRINT statement containing cursor moves.

If you make a mistake while typing a string, you must type a second quotation mark before you can position the cursor to make the correction. Typing a second quotation mark followed by the DEL key will delete the quotation mark and still allow you to move the cursor back, using the CURSOR LEFT key.

The special characters that appear in a string when one of the cursor keys is pressed are used to tell the Commodore 64/VIC 20 what to do with the cursor when the PRINT statement is executed. The reverse video keys, RVS ON and RVS OFF, when depressed with the CTRL key also have special characters associated with them when used in a string. The numerical keys 1 8 when depressed with the CTRL key on either computer or with the LOGO key on the Commodore 64 allow the character color to be changed within a PRINT statement. All of the special characters that can occur in a string except the color keys are shown in Figure 2.4(a). The color key special characters for the Commodore 64 are shown in Figure 2.4(b). Only the first eight colors shown are available on the VIC 20.

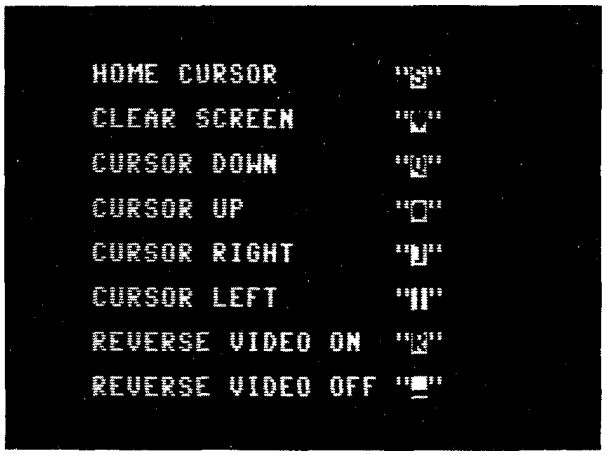
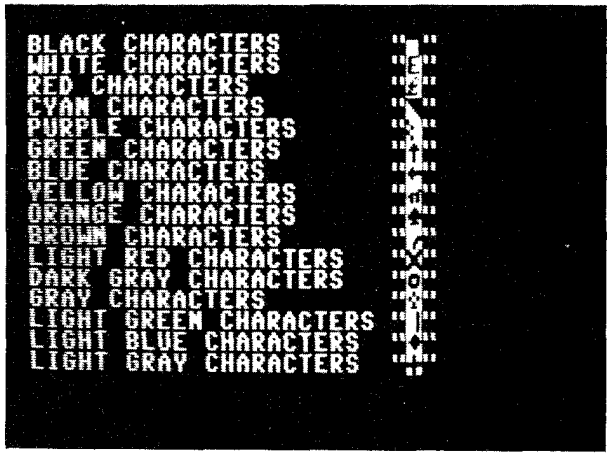


Figure 2.4(a) Noncolor special characters for string symbols.

(b) Color special characters for string symbols.



EXERCISE 2-1
Use the PRINT statement to generate the graphic figure shown in Figure 1.12 in the previous chapter. Your result should look like Figure 2.5.

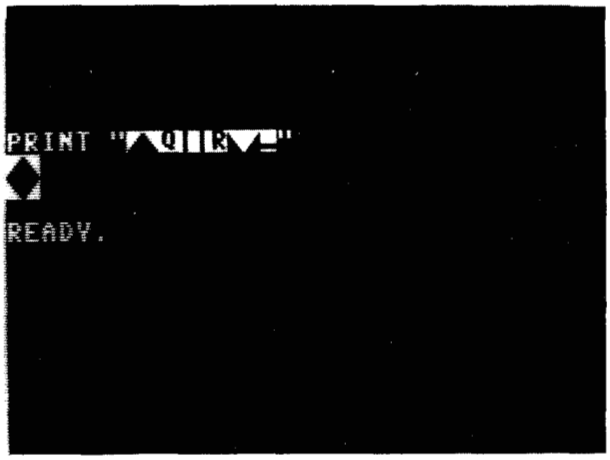


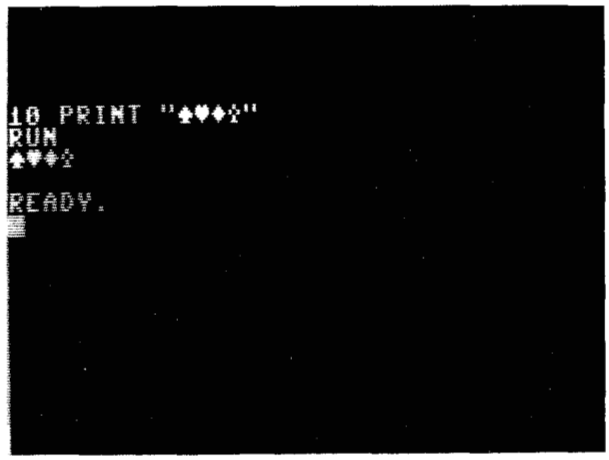
Figure 2.5 Answer to Exercise 2-1.

EXERCISE 2-2
Repeat Exercise 2-1 but make the figure cyan in color.

THE RUN AND LIST COMMANDS

If a BASIC statement such as PRINT is preceded by a line number, the statement is *not* executed immediately, but rather its execution is deferred until the command RUN is typed. Figure 2.6 shows how to print the playing card symbols using this *deferred* mode of execution.*

Figure 2.6 PRINT statement using the deferred mode of execution.



BASIC statements with line numbers are "stored" in the computer. They can be RUN at any time. If you type **RUN** again, the computer will again display the playing card symbols. Try it.

*Before typing this or a new set of statements with line numbers, type the command **NEW** to clear the memory. NEW is discussed in Chapter 3.

Note that you must press RETURN at the end of each statement (such as PRINT) or command (such as RUN). The Commodore 64/VIC 20 does not "look at" what you have typed on a line until you press RETURN.

You can find out at any time which BASIC statements you have stored in the computer by typing **LIST.** Try it. If you typed the statement shown in Figure 2.6 before, your list should look like Figure 2.7.

Figure 2.7 The LIST command will list all BASIC statements stored in memory.

You can now edit this PRINT statement by using the cursor keys. Suppose you want to print the word **HEAR** instead of the playing card symbols. Using the cursor keys, move the cursor over the spade in the PRINT statement. Then type **HEAR** and press RETURN. Now move the cursor down below READY and type **RUN.** The result should be as shown in Figure 2.8.

You can use the question mark (?) as an abbreviation for the word PRINT. Try typing **?"THIS WILL STILL PRINT".** If you want to print HELLO in the deferred mode, you can type:

10 ?"HELLO"
RUN

Note that, if you now type **LIST,** the word **PRINT** is substituted for the question mark (see Figure 2.9).

INSERT/DELETE KEY

The INSERT/DELETE key (INST/DEL, see Figure 2.10) can be used to edit any BASIC statement. If you make a mistake while typing a statement, or if you want to change a previously written statement, you will find the INSERT/DELETE key very useful.

For example, suppose you want to change the word HEAR in your PRINT statement to HEARING. First

```
LIST

10 PRINT "HE▒▒"
READY.
```

Figure 2.8 (a) Editing a PRINT statement.

```
LIST

10 PRINT "HEAR"
READY.
RUN
HEAR

READY.
```

(b) Running the edited statement.

list the statement by typing **LIST.** Then move the cursor over the last (right) quotation mark. While holding the shift key down, press the INSERT/DELETE key three times. This will move the

Figure 2.9 The question mark (?) can be used instead of the word PRINT.

```
? "THIS WILL STILL PRINT"
THIS WILL STILL PRINT

READY.

10 ? "HELLO"
RUN
HELLO

READY.

LIST

10 PRINT "HELLO"
READY.
```

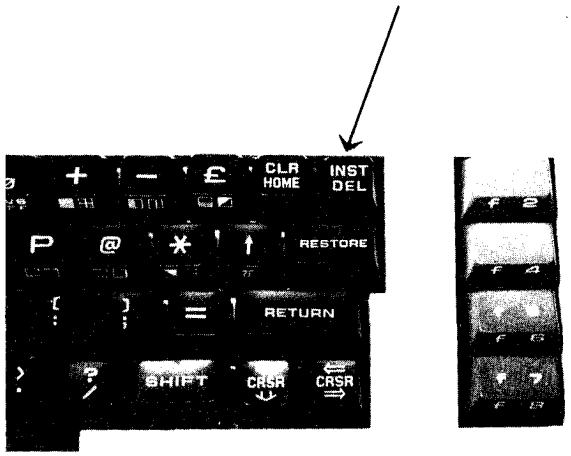


Figure 2.10 INSERT/DELETE key.

quotation mark over three places. Now type **ING** and press RETURN. Move the cursor to the beginning of the next line, and type **RUN.** The computer should print the word **HEARING** as shown in Figure 2.11. (The two lines marked "**10**" in Figure 2.11 will actually occur on the same line on the screen at different times).

```
10 PRINT "HEAR▒  "

10 PRINT "HEARING"
RUN
HEARING

READY.
```

Figure 2.11 Inserting: changing HEAR to HEARING.

Suppose you now want to change HEARING to RING. Type **LIST** (this is not necessary, but it reduces the distance on the screen that you have to move the cursor to reach the word **HEARING**) and then move the cursor over the letter **R** in **HEARING.** Press the INSERT/DELETE key three times to delete the letters **HEA.** Note that pressing the DELETE key will delete the character *immediately to the left* of the cursor. Now press RETURN, move the cursor down, and type **RUN.** The word **RING** should be printed on the screen as shown in Figure 2.12. (Again, the four lines marked "**10**" will actually occur on the same line on the screen at different times).

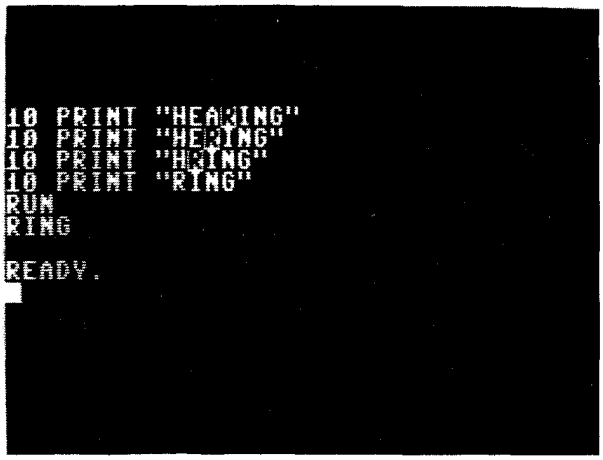


Figure 2.12 Deleting to change HEARING to RING.

STRING VARIABLES

As we have seen, any sequence of characters enclosed between quotation marks is called a string. Thus, for example, the following are strings:

"HEARING"
"$$$$"
"THIS IS A STRING"

Any character, graphic symbol, color key, or cursor move can be included in a string. A blank space is treated as a character when it is included in a string.

A string can be given a special name and can then be referred to by its name. These string names are sometimes called *string variables*. The *name* of a string must end with a dollar sign ($). It must also start with a letter, and it can contain one or two characters, the second of which can be either a letter or a number. The following are examples of valid string names:

A$
B3$
AX$
MA$
Z7$

You may use longer string names, such as HOUSE$ and BOAT$. However, the Commodore 64/VIC 20 uses only the *first two* characters to identify the string. It would therefore consider the names BALL$ and BAT$ to be the same, since they both start with the letters BA.

Longer names are more meaningful to the programmer, but not to the Commodore 64/VIC 20. In addition to the need for the first two characters of each name to be unique, there is another problem with the use of long names. The Commodore 64/VIC 20 has a number of *reserved words,* such as statement and command names. For example, RUN and LIST are reserved words. (A complete list of reserved words is given in Appendix A). If *any* of these reserved words occurs *anywhere* within a name that you make up, your program will not run properly. Thus, it is probably a good idea to keep your names short. Shorter names will also use less memory.

The equal sign (=) can be used in BASIC to *assign* a particular string to a particular string variable or name. Thus, for example, you could type **10 A$="THIS IS A STRING".** From now on the name A$ is considered to be the same thing as the string "THIS IS A STRING". You can, for example, print it with the PRINT statement **20 PRINT A$.** Try this. You should get the result shown in Figure 2.13.

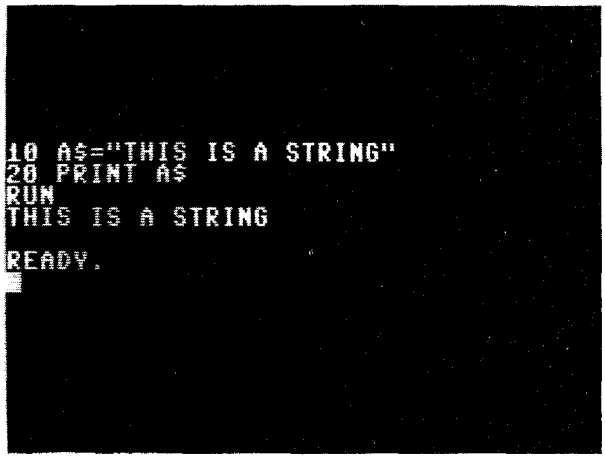
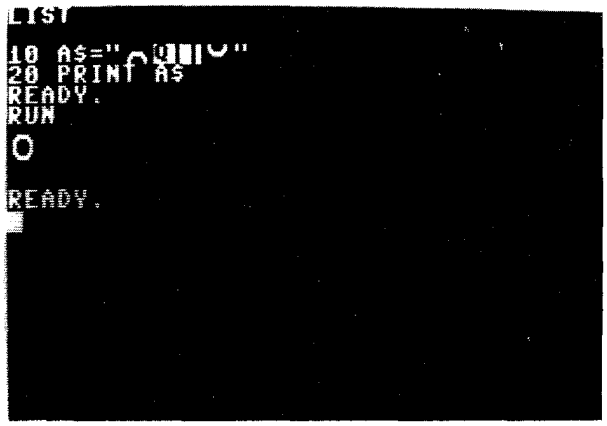


Figure 2.13 Using string variables in a PRINT statement.

Of course, you can include graphic characters and cursor moves in your definition of a string variable. Thus, for example, to draw the circular figure in Figure 2.3, you could define a string variable A$ as shown in Figure 2.14.

Figure 2.14 Defining a graphic figure as a string variable.



You may also define several graphic figures as separate strings and then print them all. For example, the three graphic figures shown in Figure 1.9 in Chapter 1 can be defined as the following three string variables:

A$: string for plotting large square
B$: string for plotting small square
C$: string for plotting diamond

Figure 2.15 shows a program that defines each of these string variables and then prints each figure. Type it in and run it. Note that each PRINT statement plots a separate figure starting on a new line. A detailed description of the PRINT statement is given in Chapter 4.

Some additional graphic figures are shown in Figure 2.16. Type in each of these programs and display the figures. Then try to draw the graphic figures shown in Exercise 2-3 at the end of this chapter. The answers to these exercises are given in the back of the book.

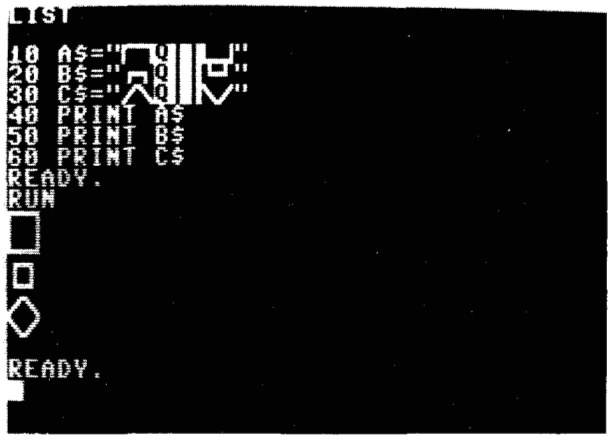


Figure 2.15 String variable definitions of the figures in Figure 1.9.

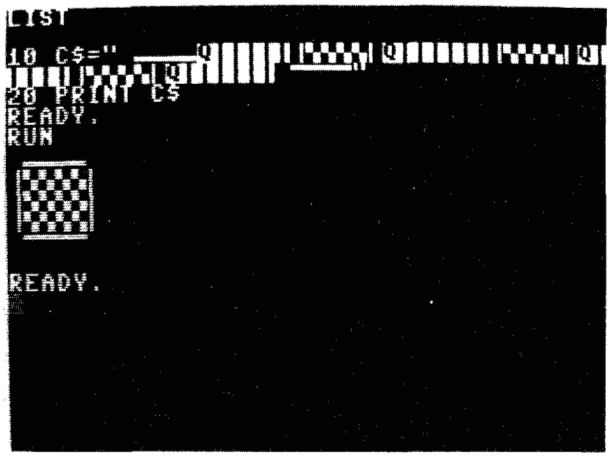
LINE LENGTH

Each line on the VIC 20 screen contains twenty-two character positions and each line on the Commodore 64 screen contains forty character positions. However,

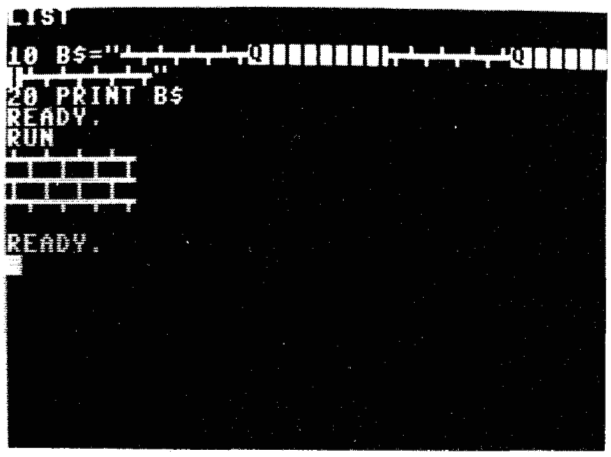
Figure 2.16 Examples of graphic figures defined as string variables.

(a)

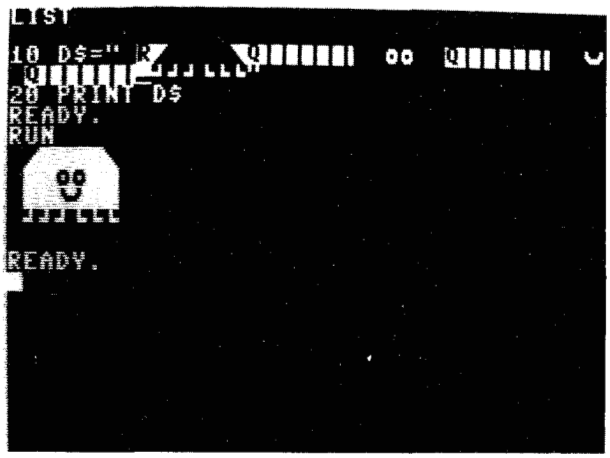
(c)



(b)



(d)

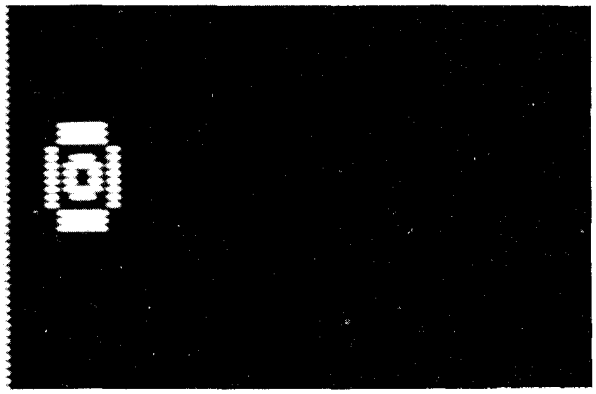


the VIC 20 is able to process up to eighty-eight characters per line and the Commodore 64 can process up to eighty characters per line. This means that you can use a maximum of four screen lines for any BASIC statement on a VIC 20 and a maximum of two screen lines for any BASIC statement on a Commodore 64. For example, if you are defining a string using a statement such as **10 A$="**------- and you get to the end of the line on the screen, you can keep on typing. Do not press RETURN. The Commodore 64/VIC 20 will automatically continue the statement on the next line. When you finish the statement, you must press RETURN. Remember that no statement can be more than four screen lines (88 characters) long on the VIC 20 and no statement can be more than two screen lines (80 characters) long on the Commodore 64.

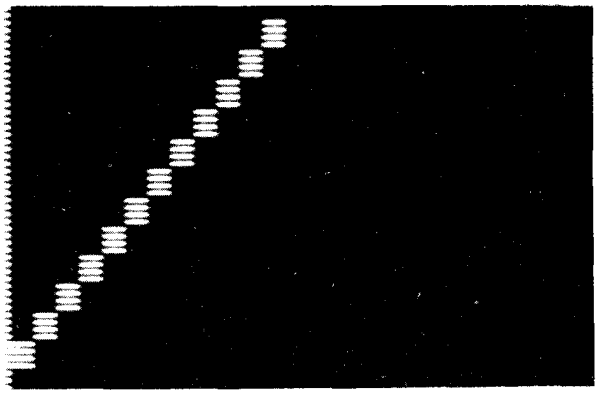
EXERCISE 2-3
Write a BASIC program to draw each of the following graphic figures.

Exercise 2-3

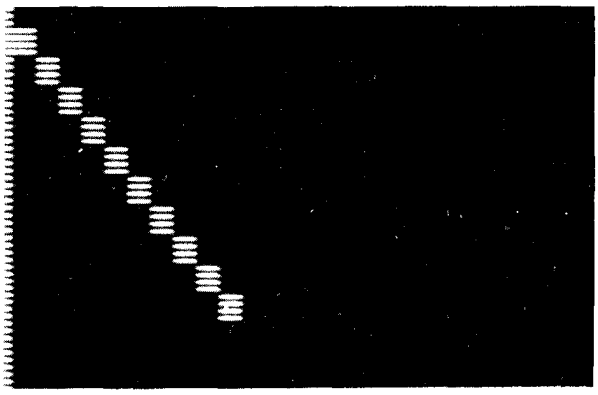
(a)



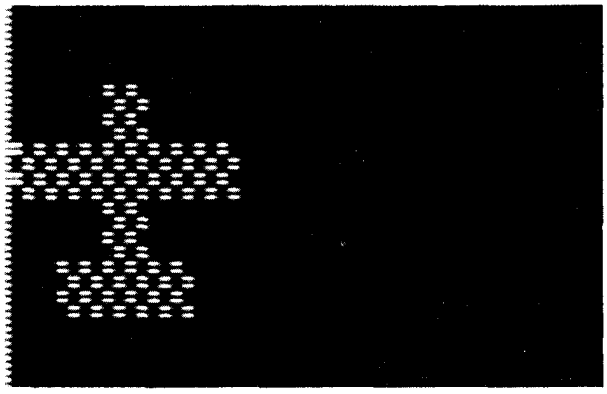
(b)



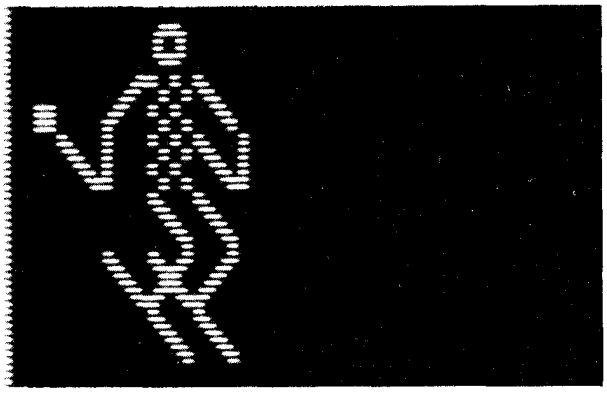
(c)



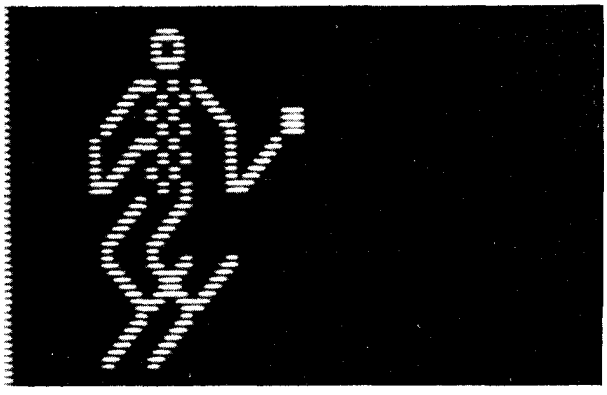
(d)



(e)



(f)



3 LEARNING TO PROGRAM IN BASIC

In the first two chapters you learned how to draw a variety of graphic figures on the Commodore 64/VIC 20. Graphics can be used to advantage in many types of computer programs, to make them more interesting and appealing. You will learn more about incorporating graphics into your programs in later chapters of this book. In this chapter we will begin to look at some of the ideas associated with writing BASIC programs.

In this chapter you will learn:

1. how to use the cassette tape recorder and floppy disk drive to save your programs
2. to use the commands NEW, SAVE, VERIFY, LOAD, and CONT
3. to use the RUN/STOP key
4. the general structure of a BASIC program
5. to use the statements GOTO, STOP, END, and REM.

THE BASIC PROGRAMMING LANGUAGE

The BASIC programming language was developed at Dartmouth College in 1963. BASIC stands for Beginners All-purpose Symbolic Instruction Code, and the language was designed to be easy to learn and easy to use. Since its original development the BASIC language has been extended and modified by various manufacturers. CBM* BASIC is used in the Commodore 64 and VIC 20 and is similar to the BASIC that is used with most other microcomputers.

The main advantages of using BASIC are that it is simple to use and that it is built into your Commodore 64/VIC 20. Although it is simple, CBM BASIC is quite powerful and will allow you to write high-performance programs.

There are, however, certain drawbacks to CBM BASIC. First of all, it is slow. You might not notice this until you try to draw a large picture quickly. The BASIC interpreter located in the Commodore 64/VIC 20 ROM must decode and execute each of your BASIC statements each time you run your program. This takes time.

Assembly Language

If you want to speed up the execution time of a program, you must write the program in *assembly language* rather than in BASIC. Assembly language is a lower-level language that the Commodore 64/VIC 20 can execute directly. The "brain" of the Commodore 64/VIC 20 is a 40-pin chip called a 6502 microprocessor (see Figure 3.1). It is this chip that can

*CBM is short for Commodore Business Machines, Inc.

decode and execute a 6502 assembly language program. Any microcomputer that uses the 6502 microprocessor can execute programs written in 6502 assembly language. The Apple II microcomputer also uses the 6502 microprocessor. On the other hand the Radio Shack TRS-80 uses the Z80 microprocessor, which executes a completely different assembly language. The Radio Shack TRS-80 Color Computer uses the 6809 microprocessor, which executes yet another assembly language.

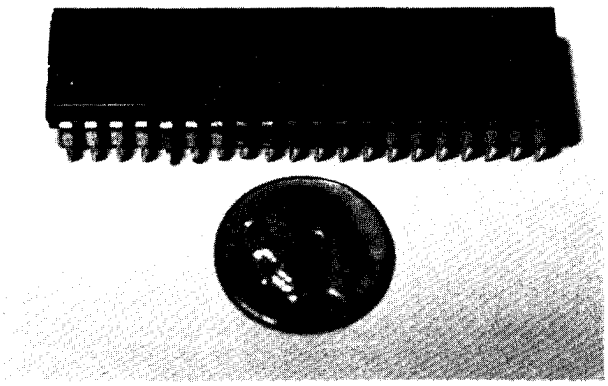


Figure 3.1 The 6502 microprocessor is the "brain" of the Commodore 64*/VIC 20 computer.

Structured Programming

You may hear that BASIC is not a very "well-structured" language and that other languages such as PASCAL are "better" in some sense. While it is true that PASCAL almost forces you to write well-structured programs, it is also true that well-structured programs can be written in any language, including BASIC. In this book we will try to counteract any bad programming habits that BASIC might encourage and show you how to write good programs in BASIC.

Learning the Language

There are two aspects to learning computer programming. The first is to learn a programming language. This is the easy part. The second is to learn how to write programs to accomplish particular tasks. This is the hard part.

Learning a computer language consists of learning the *syntax* and *semantics* of the various statements that make up the language. *Syntax* refers to the rules for forming the various statements. For example, the PRINT statement must be spelled PRINT, and a string must be enclosed between quotation marks. *Semantics* refers to what a particular statement does. For example, the statement PRINT followed by a string will print the string on the screen.

Learning How To Write Programs

Learning how to write a program to accomplish a particular task takes practice, insight, and a knack for solving problems. Programming is like solving a puzzle. You must determine what you have to tell the computer so that the computer will do what you want. You will find that the computer *always* does exactly what you tell it to do. However, often what you tell it to do is not what you think you are telling it to do. This can lead to errors that are sometimes hard to find. The best way to avoid many of these errors is to think through the problem carefully before you start to write the program. Understanding exactly what you want to do is a major step in solving a problem.

There are only a few basic rules for telling a computer what to do. Computers can do the same thing over and over again. This is accomplished in a computer program by means of a *loop*. We will look at a simple loop later in this chapter. More detailed discussions of loops are given in Chapters 7 and 8. Another thing that computers can do is make a simple choice between two alternatives. This process of making choices will be described in Chapter 6. By combining loops with the process of making simple choices, any computer program can be constructed.

Saving Your Programs

If you have a Commodore 64/VIC 20 you probably have a cassette tape recorder, as in Figure 3.2a. You may also have a floppy disk drive as shown in Figure 3.2b. Both devices are used to save programs and data when the Commodore 64/VIC 20 is shut off.

CASSETTE TAPE RECORDER

Press the STOP/EJECT key on the recorder all the way down to open the cover. Insert a new tape. (The best tapes to use are C-10 data tapes that contain fifty feet of tape. These tapes can store one long program or several short programs on each side.) Insert the cassette with the tape opening facing you. The left side of the cassette should contain the most tape. Otherwise, you will need to rewind the cassette. After inserting the cassette completely, close the cover.

*The Commodore 64 actually uses the 6510 microprocessor which differs from the 6502 in only one, relatively minor, respect.

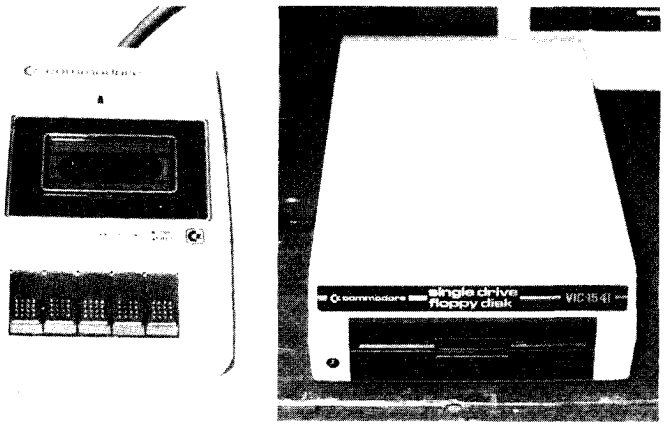


Figure 3.2 Commodore 64/VIC 20 cassette tape recorder (a) and floppy disk drive (b).

NEW

Type **NEW** and press RETURN. This will clear any BASIC program that you have stored in the computer. You should type NEW before you begin typing in a new program, so that parts of old programs are not combined with your new program. Now type in the program shown in Figure 3.3.

Figure 3.3 Program to draw three figures.

```
10 A$="
20 B$="
30 C$="
40 PRINT A$
50 PRINT B$
60 PRINT C$

READY.
```

SAVE

You can now save this program on the cassette tape by typing **SAVE "THREE FIGURES".** The Commodore 64/VIC 20 will respond with the message **PRESS RECORD & PLAY ON TAPE.** Now press both the REC key (the one on the far left) and the PLAY key. The Commodore 64/VIC 20 will first respond with **OK** and then **SAVING THREE FIGURES.** On all tape commands, the Commodore 64 (not the VIC 20) blanks the screen while it carries out the instruction. These steps are shown in Figure 3.4.

Your program is now stored on the cassette tape under the *file name* "THREE FIGURES." You can make up any name you want for a file name, but you should limit your file names to sixteen characters (including blanks), since this is the maximum number that the Commodore 64/VIC 20 will display on the screen when you load the program in from the tape.

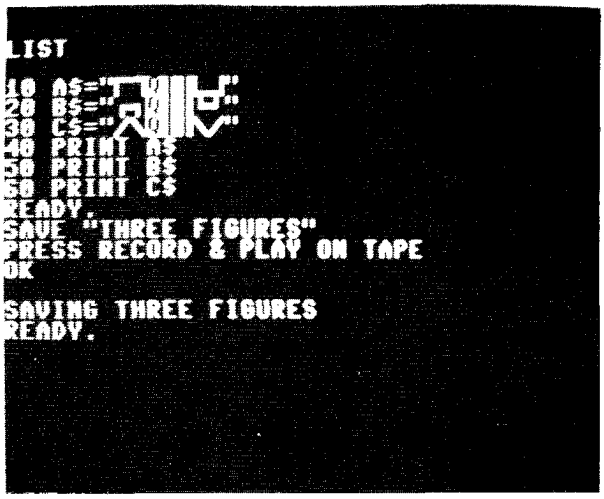


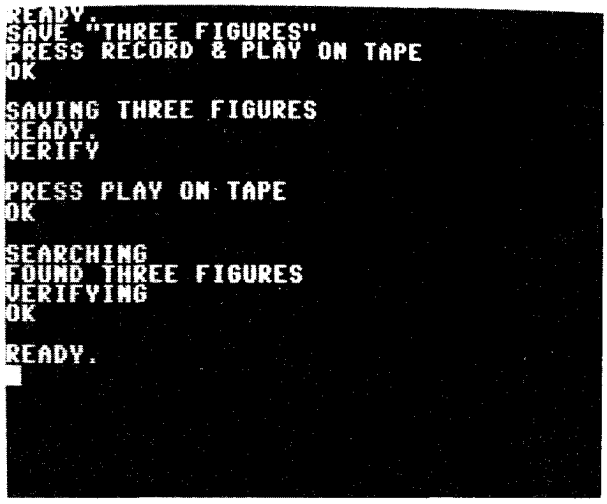
Figure 3.4 Saving a program on cassette tape.

VERIFY

You can verify that the program in the Commodore 64/VIC 20 was properly stored on the tape by rewinding the tape (press the REW key on the recorder until the tape is completely rewound, and then press the recorder STOP key) and then typing **VERIFY.** The Commodore 64/VIC 20 will then respond with the message **PRESS PLAY ON TAPE.**

Press the PLAY key on the recorder. The Commodore 64/VIC 20 will read your tape and compare it with the program that is stored in its memory. If no errors occurred when you saved your program, the Commodore 64/VIC 20 will display **OK.** If an error did occur, the Commodore 64/VIC 20 will display **?VERIFY ERROR** and you should try to SAVE your program again. The use of the VERIFY command is illustrated in Figure 3.5.

Figure 3.5 Use of the VERIFY command.



LOAD

Type **NEW.** This will clear your program from the memory. You cannot run this program again until you load it back in from the cassette tape. To do this, rewind the tape, and then type **LOAD** or **LOAD "THREE FIGURES."**

If you type **LOAD** then the Commodore 64/VIC 20 will respond with **PRESS PLAY ON TAPE.** Press the recorder PLAY key, and your program will be loaded into the Commodore 64/VIC 20 (see Figure 3.6). When you type only the word **LOAD,** the first program on the tape will be loaded into memory.

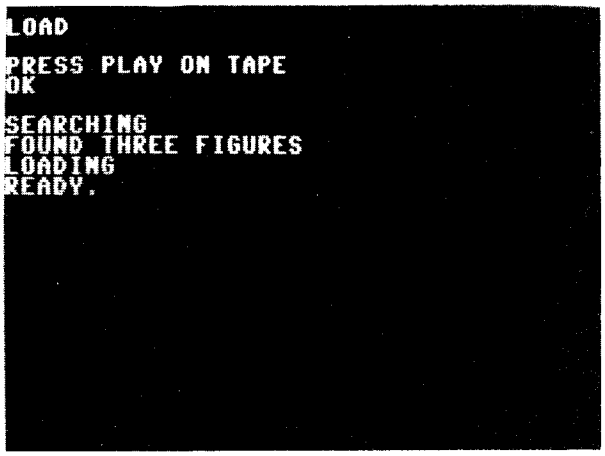


Figure 3.6 The statement LOAD will load the first program on the tape into the Commodore 64/VIC 20 memory.

If you type **LOAD "THREE FIGURES"** the Commodore 64/VIC 20 will search through the tape until it finds the file name "THREE FIGURES" and then load this file into memory (see Figure 3.7). The Commodore 64 (not the VIC 20) pauses after finding the correct file before loading it. You can eliminate the pause by pressing the LOGO key. Since your program was the first program on the tape, it will be found right away. However, if it had been the third program on the tape, the Commodore 64/VIC 20 would have searched until it found the correct name, without loading in the first two programs.

Figure 3.7 The statement LOAD "THREE FIGURES" will search for the file name THREE FIGURES and load this file into memory.

```
LOAD ''THREE FIGURES''

PRESS PLAY ON TAPE
OK

SEARCHING FOR THREE FI
GURES
FOUND THREE FIGURES

LOADING
READY.
```

THE FLOPPY DISK DRIVE

SAVE

If your Commodore 64/VIC 20 has a floppy disk connected to it, then you can ready the disk drive for saving and loading programs by inserting a formatted* floppy diskette (label up, notch at the left) into the slot on the front of the disk drive and then closing and latching the drive door.

To save your program on the floppy diskette, type

SAVE "THREE FIGURES",8

where THREE FIGURES is the name of the program. (The number 8 is the device number of the first disk drive in the system.) You can make up any program name containing up to 16 characters including blankspaces. The red light on the disk drive will light up and the disk drive motor will make a whirring sound for a few seconds. After it stops, everything should be all right as long as the red light is not flashing. A flashing red light means an error occurred in carrying out the instruction. For example, an error will occur if you try to save a program when there is no diskette in the disk drive. At this point, assuming no error occurred, the dialog on the screen should look like Figure 3.8.

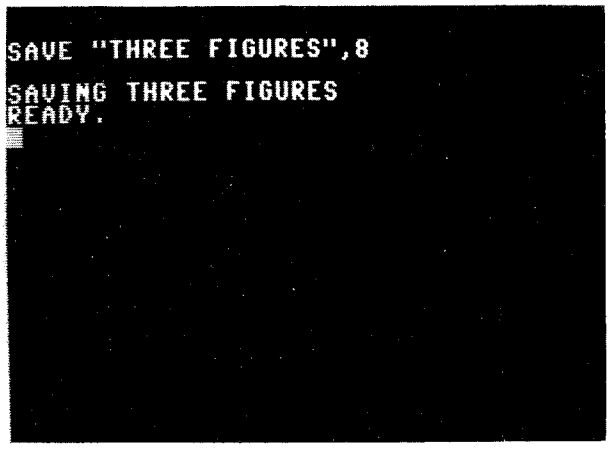
VERIFY

To verify that your program is really on the disk, type

VERIFY "THREE FIGURES",8

*To format a new diskette, consult your disk drive manual.

Figure 3.8 Saving a program named THREE FIGURES on the floppy diskette.



The red light on the disk drive will come on and you will hear the whirring sound again. The Commodore 64/VIC 20 will respond first with **SEARCHING FOR THREE FIGURES** and then **VERIFYING.** If the program was saved correctly on the diskette then the Commodore 64/VIC 20 will display **OK** as shown in Figure 3.9. If not, the display will read **?VERIFY ERROR.** The latter means the program was not SAVEd properly.

```
SAVE "THREE FIGURES",8

SAVING THREE FIGURES
READY.
VERIFY "THREE FIGURES",8

SEARCHING FOR THREE FIGURES
VERIFYING
OK

READY.
```

Figure 3.9 Verifying a program named THREE FIGURES on the diskette.

LOAD

Another way to verify that your program is on the disk is to type **NEW,** which will clear your BASIC program in the Commodore 64/VIC 20. If you now type **LIST** you will find that nothing gets listed. In order to retrieve your program from the diskette, type

LOAD "THREE FIGURES",8

The Commodore 64/VIC 20 will respond first with **SEARCHING FOR THREE FIGURES** and then **LOADING.** When the red light goes out, your program will be completely loaded. See Figure 3.10. You can now see the program listing by typing **LIST** and you can execute the program by typing **RUN.**

It is possible to get a list of the programs stored on a diskette. Just type

LOAD "$",8

Here, "$" denotes the file kept by the system containing the names of all programs on the diskette. If you now type **LIST,** you should see the name **"THREE FIGURES"** on the list. Loading "$" has the side effect of causing THREE FIGURES or any other program in memory to be erased.

```
NEW

READY.
LOAD "THREE FIGURES",8

SEARCHING FOR THREE FIGURES
LOADING
READY.
LIST

10 A$="
20 B$="
30 C$="
40 PRINT A$
50 PRINT B$
60 PRINT C$
READY.
```

Figure 3.10 Loading the program THREE FIGURES from the diskette.

ReSAVEing Programs

Once you have SAVEd a program and then modified it, you may want to save it on the disk again, erasing the copy already there. The command for this is

SAVE "@0:THREE FIGURES",8

(The digit 0 can ordinarily be omitted.) To see how this works, carry out the sequence of commands in Figure 3.11. In executing this new SAVE command, if THREE FIGURES by chance is not already on the diskette, then the SAVE is carried out in the usual manner described earlier.

If you try to SAVE a program without the @0:(or@:) prefixes on the program name and there is a program on the diskette with the same name, then the program will not be SAVEd. Unfortunately, the only indication you will get of this problem is the flashing red light on the disk drive, and this is easy to overlook.

Figure 3.11 ReSAVEing the program THREE FIGURES on the diskette.

```
LOAD "THREE FIGURES",8

SEARCHING FOR THREE FIGURES
LOADING
READY.
70 GOTO 40
SAVE "@0:THREE FIGURES",8

SAVING @0:THREE FIGURES
READY.
```

STOPPING PROGRAM EXECUTION

RUN/STOP Key

The RUN/STOP key is shown in Figure 3.12. If you hold the SHIFT key down and press the RUN/STOP key, the Commodore 64/VIC 20 will respond with the message **PRESS PLAY ON TAPE.** If you then press the recorder PLAY key, the Commodore 64/VIC 20 will LOAD in your program and then RUN it. In other words, pressing the RUN/STOP key while holding down the shift key is equivalent to typing the two commands **LOAD** and **RUN.** Rewind your tape and try it.

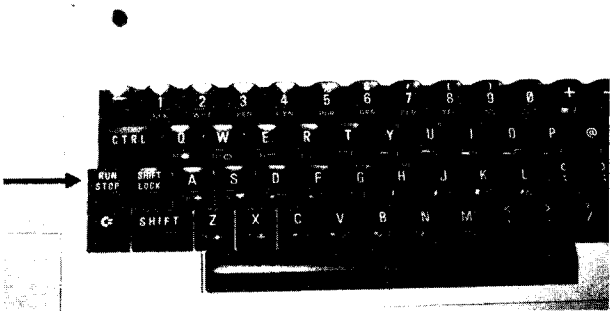


Figure 3.12 The RUN/STOP key.

The unshifted RUN/STOP key is used to stop the execution of a program. To see how this works, add the statement **70 GOTO 40** to the program you have in memory. You can do this by just typing this statement as shown. Then type **LIST** in order to see the entire program. It should look like Figure 3.13. The statement **70 GOTO 40** means exactly what it says. Statement 70 simply branches back to statement 40. This is a loop that continues indefinitely, as shown in Figure 3.14.

Figure 3.13 Program to display a continuous sequence of figures.

```
10 A$="
20 B$="
30 C$="
40 PRINT A$
50 PRINT B$
60 PRINT C$
70 GOTO 40

READY.
```

Figure 3.14 An indefinite loop that prints figures until you press the STOP key.

```
40 PRINT A$   (prints the large square)
50 PRINT B$   (prints the small square)
60 PRINT C$   (prints the diamond)
70 GOTO 40
```

Now RUN this program. As you can see, new figures are being printed endlessly. In order to stop this program, press the STOP key (the unshifted RUN/STOP key). Note that you get a **BREAK** message as shown in Figure 3.15.

CONT

The program shown in Figure 3.13 displays a large square, a small square, and a diamond over and over again. No matter where it is stopped, if you restart the program by typing **RUN,** it will start with the large square. This can be seen in Figure 3.16, where the program was stopped with the STOP key just after it displayed the small square.

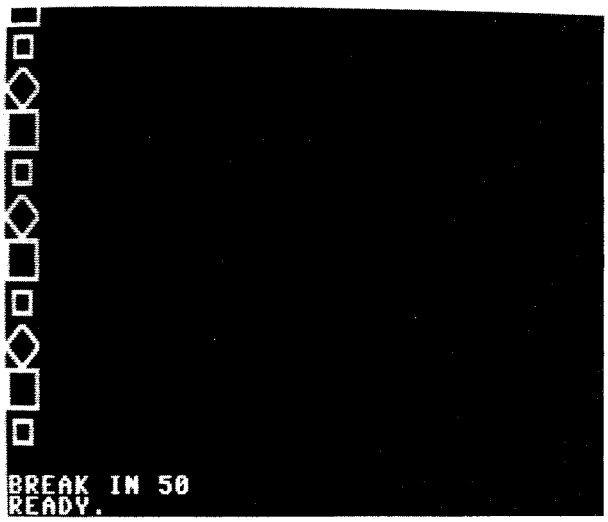


Figure 3.15 Stopping a program with the STOP key.

If a program has been stopped, the CONT statement can be used to continue the program where it left off. This is illustrated in Figure 3.17, where the program was again stopped just after displaying the small square. After **CONT** is typed, the program restarts by displaying the diamond.

STOP

The statement **STOP** can be included in a BASIC program. This will have the same effect as pressing the STOP key. This can be very useful in *debugging* (finding the errors in) a program that does not work properly. You can insert a STOP statement and then check what the program has done up to that point. You can then resume execution of the program by typing **CONT.**

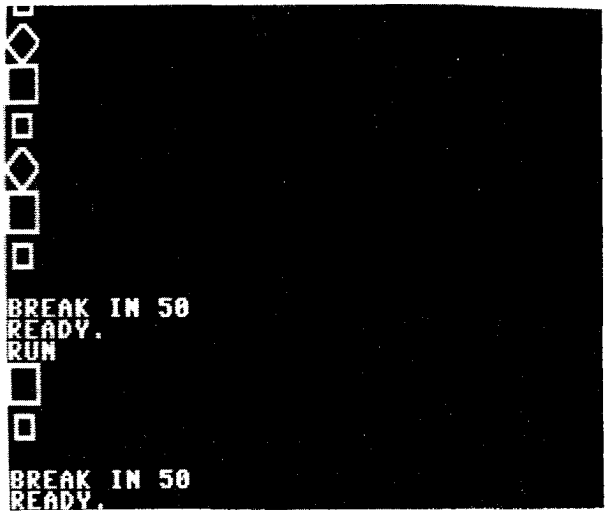


Figure 3.16 RUN causes the program to start at the beginning.

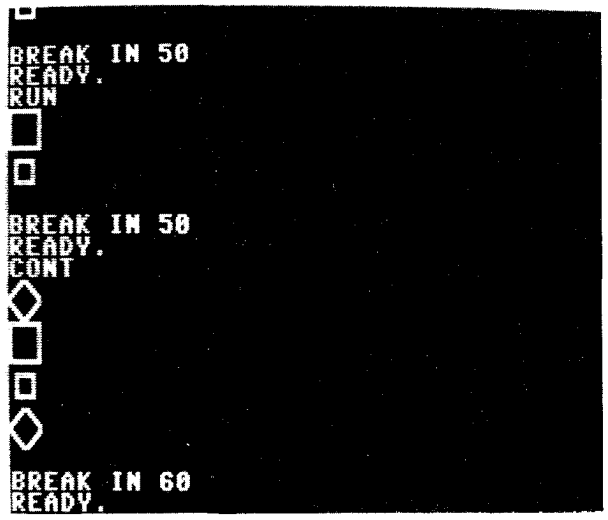


Figure 3.17 CONT causes the program to start from where it left off.

END

The END statement can be used to stop a BASIC program in the middle or at the end. It does not cause a BREAK message as the STOP statement does. In CBM BASIC the END statement is optional, because the program will automatically end if there are no more statements to execute.

THE STRUCTURE OF A BASIC PROGRAM

Sequence Numbers

A BASIC program consists of a sequence of BASIC statements. Each line of a BASIC program must begin with a *sequence number*. When the program is executed (by typing **RUN**), the statement with the lowest sequence number is executed first. Additional statements are then executed in numerical order.

When you write a BASIC program you should increment your sequence numbers by 10. That is, your program should look like this:

10 first statement
20 second statement
30 third statement
.
.
.

If you later want to insert a new statement between the second and third statements you can type **25** NEW STATEMENT and this new statement will be inserted between statement 20 and statement 30. Leaving unused numbers between statements will save you the chore of renumbering all of your statements when inserts are made.

If you think you may want to add some new statements at the beginning of your program, it would be a good idea to start your program with a sequence number of 100 and then continue with 110, 120, 130, and so on. You can later insert statements before line 100 by using sequence numbers less than 100. A sequence number can be any integer from 0 to 63999.

REM

A good statement to include at the beginning of your program is a REMark statement. This statement consists of the three letters REM. The remainder of the line can be used for any kind of *remark,* or descriptive comment. These remarks are ignored by the Commodore 64/VIC 20 when the program is executed. Their only purpose is to make the program easier to understand. For example, in the program shown in Figure 3.13 you may want to add the statement:

5 REM PROGRAM TO PRINT 2 SQUARES AND A DIAMOND CONTINUOUSLY

Note that, unlike the listing of Figure 3.18, the remark takes more than one line on the screen. As mentioned in Chapter 2, any BASIC statement can use up to eighty-eight character positions on the VIC 20 or eighty character positions on the Commodore 64 (four/two screen lines, respectively). When you type the remark shown in Figure 3.18, *do not press* RETURN at the end of the first line, or you will terminate the statement at that point. The REMark will just continue on the second line. If you press

Figure 3.18 Use of the REM statement to make remarks in a program on the Commodore 64/VIC 20.

```
5 REM  PROGRAM TO PRINT 2 SQUARES AND A DIAMOND CONTINUOUSLY
10 A$="┌──┐"
20 B$=" │  │ "
30 C$="└──┘"
40 PRINT A$
50 PRINT B$
60 PRINT C$
70 GOTO 40
```

RETURN at the end of the first line then you must start the second line with a new sequence number and another REM statement.

Multiple Statements per Line

CBM BASIC allows you to write more than one BASIC statement per line by separating the statements with a colon (:). In this context a line is the eighty-eight character line consisting of four screen lines on the VIC 20 or the eighty-character line consisting of two screen lines on the Commodore 64. This can be an advantage for a number of reasons: (1) it allows you to group a number of short, related statements together; (2) it allows you to include remarks on the same line as a BASIC statement; and (3) it saves some memory by reducing the number of sequence numbers in the program. Only the first BASIC statement on a line has a sequence number.

There are, however, some disadvantages to writing more than one statement per line. If used indiscriminately, this style can result in a program that is difficult to read and understand. You will not be able to branch to a statement (for example, with a GOTO statement) that starts in the middle of a line, since it will not have a sequence number. Finally, you will not be able to insert a new statement between existing multiple statements unless there is enough room left on the line to use the INSERT key. You should, therefore, be careful when writing multiple statements on a single line.

One good use of the multiple statement capability is to include remarks that help to describe what is going on in the program. For example, in Figure 3.19 we have added three remarks that tell what is being printed by each of the PRINT statements. You can see how the remarks in Figure 3.19 have made the program easier to understand. Note that a colon (:) is used to separate multiple statements on a single line. But remember that the REM statement will cause the rest of the line to be ignored, even if you add another BASIC statement following another colon.

More about LIST

We have seen that the LIST command will list the entire BASIC program that is stored in memory. You can slow down the speed at which the program is listed by pressing the CTRL key while the program is being listed. If you release the key the program will continue to be listed at its normal speed. This is particularly useful when listing long programs. You can stop the listing process at any time by pressing the STOP key.

It is also possible to list only selected parts of a program. For example, if you type **LIST 30** then only the line with the sequence number 30 will be printed on the screen. This is useful if you want to edit line 30 using the cursor keys.

You can list lines 20 through 40 by typing **LIST 20-40.** If you type **LIST-30** then you will list all of the lines from the beginning of the program through line

Figure 3.19 Multiple statements on a single line are separated by a colon (:).

```
5 REM  PROGRAM TO PRINT 2 SQUARES AND A DIAMOND CONTINUOUSLY
10 A$="┌──┐"
20 B$=" │  │ "
30 C$="└──┘"
40 PRINT A$:REM LARGE SQUARE
50 PRINT B$:REM SMALL SQUARE
60 PRINT C$:REM DIAMOND
70 GOTO 40
```

30. If you type **LIST 30-** you will list all of the lines from line 30 to the end of the program. These examples are shown in Figure 3.20.

MEMORY LOCATIONS AND COMPUTER PROGRAMS

A computer program is like a train going on a trip. The memory locations or memory cells in the computer are like the seats in the train. Each seat has an "address" or name that identifies it. These names correspond to the *variable* names in a BASIC program. For example, three different seat names could be A$, B$, and C$.

Whoever or whatever is in a particular seat corresponds to the *contents* of a particular memory location in the computer. For example, if JOHN is sitting in seat A$, then the BASIC statement **A$="JOHN"** can be interpreted as meaning "put JOHN in seat A$." It is very important to distinguish clearly between the name of the memory location, or seat on the train (A$), and the contents of that memory location or seat (JOHN). (See Figure 3.21.)

Up to now all of our memory locations have contained strings and have had names that end with a dollar sign. If a memory cell names does not end with a dollar sign, the Commodore 64/VIC 20 computer will assume that the memory cell contains a number. The use of memory cells containing numbers will be discussed in the next chapter.

Figure 3.20 Examples of the use of the LIST statement.

```
LIST 30
30 C$="
READY.
```

```
LIST 20-40
20 B$="
30 C$="
40 PRINT A$:REM  LARGE SQUARE
READY.
```

```
LIST -30
5 REM  PROGRAM TO PRINT 2 SQUARES AND A
DIAMOND CONTINUOUSLY
10 A$="
20 B$="
30 C$="
READY.
```

```
LIST 30-
30 C$="
40 PRINT A$:REM  LARGE SQUARE
50 PRINT B$:REM  SMALL SQUARE
60 PRINT C$:REM  DIAMOND
70 GOTO 40
READY.
```

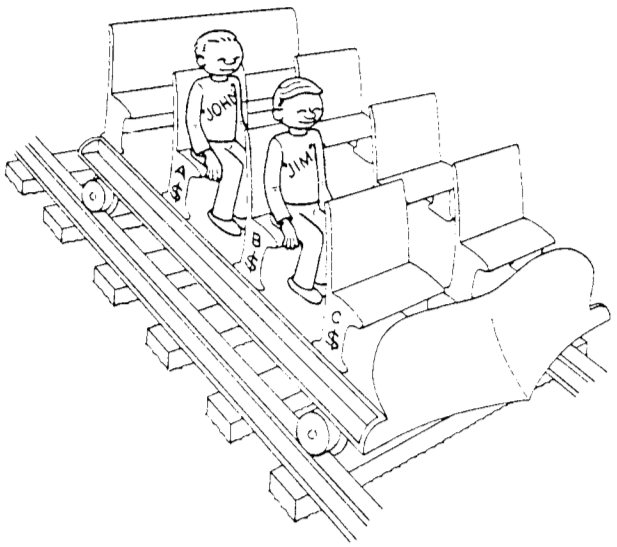


Figure 3.21 Memory locations are like seats on a train.

EXERCISE 3-1

Store the programs for Exercise 2-2 on a cassette tape. After storing the first program, do not move the position of the tape before storing the second program.

EXERCISE 3-2

Write a program that will continuously display the two skiers in Exercise 2-3 (e)-(f) over and over again. Stop the program by pressing the STOP key, and restart it by typing **CONT.**

4 LEARNING MORE ABOUT PRINT

In the first three chapters of this book you have written short programs that draw various graphic figures. In this chapter you will see how the Commodore 64/VIC 20 can work with *numbers* as well as strings. You will find that the Commodore 64/VIC 20 can serve as a very good calculator. In addition you will learn how to use the PRINT statement to make larger graphic figures.

In this chapter you will learn:

1. how to use the Commodore 64/VIC 20 as a calculator
2. to write arithmetic expressions involving addition, subtraction, multiplication, division, and exponentiation
3. how to use the comma and semicolon in a PRINT statement
4. to use the BASIC functions SPC and TAB
5. to draw larger graphic figures
6. about some of the built-in functions in the Commodore 64/VIC 20.

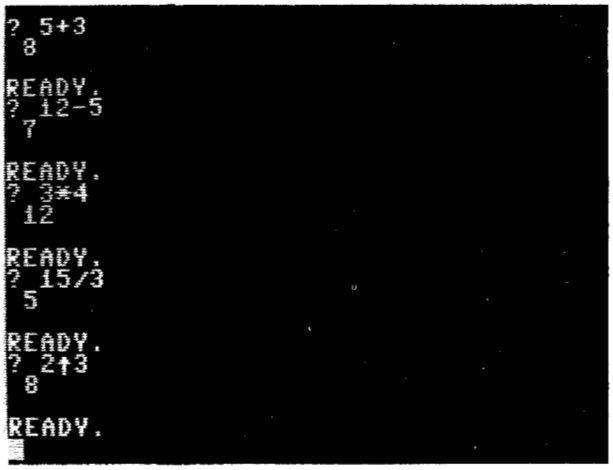
THE COMMODORE 64/VIC 20 AS A CALCULATOR

By using the PRINT statement in the immediate mode of execution, you can use your Commodore 64/VIC 20 as a calculator. You can add, subtract, multiply, divide, and raise a number to a power.

Addition

If you type **PRINT 5+3** the Commodore 64/VIC 20 will respond with **8.** You can use the question mark as an abbreviation for PRINT. Thus, if you type **? 5+3** the Commodore 64/VIC 20 will also respond with **8,** as shown in Figure 4.1. Try it.

Figure 4.1 Using the Commodore 64/VIC 20 as a calculator.



Subtraction

If you type **? 12−5** the Commodore 64/VIC 20 will respond with **7** as shown in Figure 4.1. Try it.

Multiplication

The symbol for multiplication in BASIC is the asterisk (*). Thus, if you type **? 3*4** the Commodore 64/VIC 20 will respond with **12** as shown in Figure 4.1. Try it.

Division

The symbol for division in BASIC is the slash (/). Thus, if you type **?15/3** the Commodore 64/VIC 20 will respond with **5** as shown in Figure 4.1. Try it.

Exponentiation

The symbol for exponentiation in BASIC is the upward arrow (↑). Thus, if you want to raise 2 to the power of 3 (2 cubed) you would type **? 2↑3** and the Commodore 64/VIC 20 would respond with **8** as shown in Figure 4.1. Try it.

Arithmetic Expressions

The arithmetic operators +, −, *, /, and ↑ can be combined in various ways in a single arithmetic expression. For example, if you type **? 5+3−2** the Commodore 64/VIC 20 will respond with **6.** Now type the expression **? 6+ 12/2 + 4.** Did the Commodore 64/VIC 20 display what you thought it would?

The Commodore 64/VIC 20 does division before addition, so the answer is 16. That is, the expression is evaluated as 6+(12/2)+4 = 16.

(Not all computer languages work this way. For example, the language APL evaluates all expressions from right to left. Thus, in APL the expression you typed would have a value of 8.)

In BASIC, arithmetic expressions are evaluated according to the following *order of precedence:*

1. all exponentiations, ↑, are evaluated first,
2. all multiplications, *, and divisions, /, are evaluated next,
3. all additions, +, and subtractions, −, are evaluated last.

Within each level of precedence, expressions are evaluated from *left to right.* Parentheses can be used to change this order of precedence. In this case expressions within the innermost parentheses are evaluated first.

Try to evaluate each of the following arithmetic expressions and then type them on the Commodore 64/VIC 20 to check your results. The answers are shown in Figure 4.2.

? 8−3+4/2
? 3*2−5+8/4
? 8+1/3
? 5+3*4/6
? (3+4)*(6−3)
? 20/2/5

```
? 8-3+4/2
 7

READY.
? 3*2-5+8/4
 3

READY.
? 8+1/3
 8.33333333

READY.
? 5+3*4/6
 7

READY.
? (3+4)*(6-3)
 21

READY.
? 20/2/5
 2

READY.
```

Figure 4.2 Evaluation of arithmetic expressions on the Commodore 64/VIC 20.

Did you guess the correct answer for the last one? Remember that the two divisions are evaluated from left to right so that the correct result is

$$\frac{20/2}{5} = \frac{10}{5} = 2$$

and not

$$\frac{20}{2/5} = \frac{20*5}{2} = 50$$

If you want the second result you can type **? 20/(2/5).** Try it.

Note that in the next to last example in Figure 4.2 it is necessary to use the multiplication symbol, *. Although (3+4)(6−3) is used to imply multiplication in ordinary algebra, it does *not* imply multiplication to the Commodore 64/VIC 20. Any time you want to multiply anything on the Commodore 64/VIC 20 you *must* use the multiplication symbol, *.

Pi, π

The exponentiation key on the Commodore 64/VIC 20 has the Greek letter pi, π, below the symbol ↑. The value of π is 3.14159265 . . . and is equal to the ratio of the circumference of a circle to its diameter. The Commodore 64/VIC 20 has this useful value stored in its memory, and you can use it in any expression. For example, type **?π** and the Commodore 64/VIC 20 will print the value of π (see Figure 4.3).

A circle of radius 5 has a circumference equal to 2*5*π. To find the value, type **? 2*5*π** as shown in Figure 4.3.

The area of a circle of radius 5 can be found by typing **?π*5↑2** as shown in Figure 4.3.

```
?π
 3.14159265

READY.
? 2*5*π
 31.4159265

READY.
? π*5↑2
 78.5398164

READY.
```

Figure 4.3 Examples of using π.

NUMERICAL VARIABLES

We have seen that strings such as "JOHN" can be stored in memory cells with names such as A3$. If a memory cell name does *not* end with a dollar sign, the Commodore 64/VIC 20 will assume that the memory cell contains a numerical value. For example, if you type

A=3

?A

the Commodore 64/VIC 20 will respond with 3 as shown in Figure 4.4a. Similarly, if you type

A=5

B=3

?A*B

the Commodore 64/VIC 20 will respond with **15** as shown in Figure 4.4b.

Note that these examples used the immediate mode of execution. The deferred mode of execution can also be used, as shown in Figure 4.5.

(a)

```
A=3

READY.
? A
 3

READY.
```

(b)

```
A=5

READY.
B=3

READY.
? A*B
 15

READY.
```

Figure 4.4 Numerical variables can be used in (a) the immediate mode of execution and (b) in arithmetic expressions.

Figure 4.5 Use of numerical variables in the deferred mode of execution.

```
10 A=5
20 B=3
30 PRINT A*B
READY.
RUN
 15

READY.
```

The rules for naming numerical variables are the same as the rules for naming string variables, except that there is no dollar sign at the end of the name. That is, each name can contain one or two characters; the

first must be a letter, while the second can be a letter or a numeral. The following are examples of valid names for numerical variables: Q, A3, XX, AT, C2.

As mentioned in discussions of string variables, the Commodore 64/VIC 20 will allow you to type more than two characters for the name of a numerical variable. However, it looks at only the first two characters. To verify this, try typing these statements, as shown in Figure 4.6:

BOX=7
?BOB

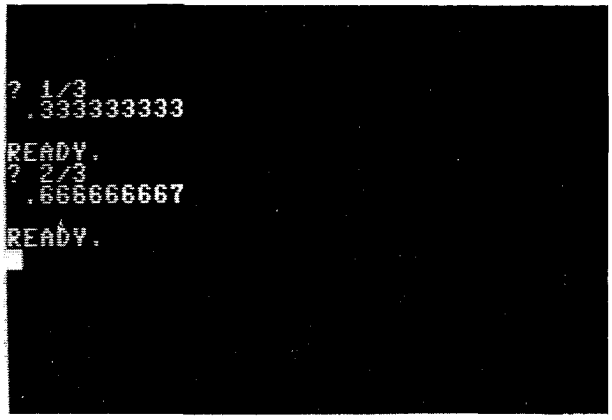
You must be careful if you use memory cell names containing more than two characters.

Figure 4.6 The Commodore 64/VIC 20 uses only the first two characters of a memory cell name.

SCIENTIFIC NOTATION

How many digits of a number does the Commodore 64/VIC 20 display? Try typing **?1/3** and **?2/3** as shown in Figure 4.7. Note that nine digits are displayed, and the last **6** in **2/3** is rounded to **7.**

Figure 4.7 The Commodore 64/VIC 20 displays nine digits and rounds the last digit.



What happens if you type in a number containing ten or more digits? Try typing **?1122334455** as shown in Figure 4.8. Note that the Commodore 64/VIC 20 has rounded the number to **112233446** and then rewritten the number in a form that contains an E. This is called *scientific notation.* The number after the E is the number of places to move the decimal point in order to obtain the correct number (1.12233446E+09 = 1122334460). If the number after the E is *positive,* move the decimal point to the *right.* If the number after the E is *negative,* move the decimal point to the *left.* Try typing **?.00123** as shown in Figure 4.8.

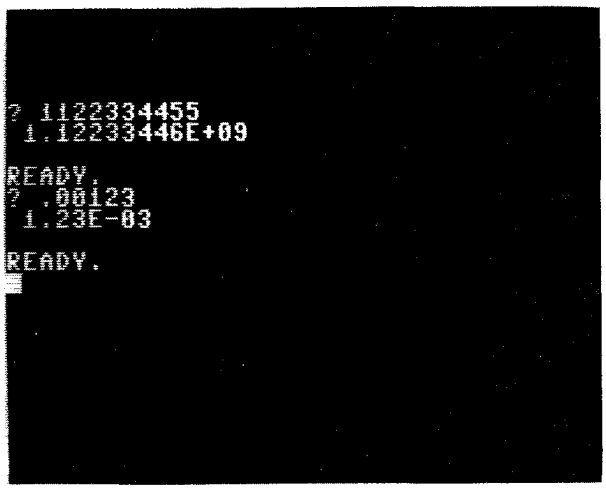


Figure 4.8 Scientific notation is used by the Commodore 64/VIC 20 for numbers greater than 999999999 and less than 0.01.

The Commodore 64/VIC 20 uses scientific notation for numbers greater than 999999999 and less than 0.01. You can use scientific notation if you want to, and the Commodore 64/VIC 20 will convert back to standard notation if your number is between 0.01 and 999999999. Some examples are shown in Figure 4.9. Note that the Commodore 64/VIC 20 printed **?OVERFLOW ERROR** when we tried to print **1.8E38.** It turns out that the *largest number* (magnitude) that the Commodore 64/VIC 20 can store is **±1.70141183E+38.** If you try to store a larger number you will get an overflow error. Also, any number between ±2.93875388E-39 will be stored in the Commodore 64/VIC 20 as zero.

CONTROLLING PRINTED OUTPUT

When you use the PRINT statement you are able to control where on the screen the output is to be printed by using commas, semicolons, and the functions SPC and TAB.

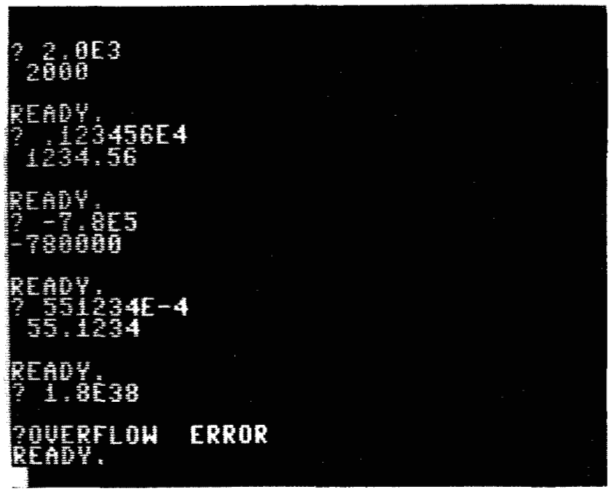


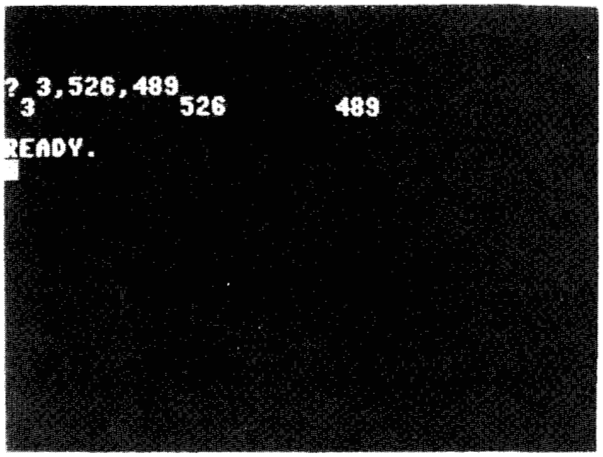
Figure 4.9 You can use scientific notations in your programs.

Comma

The comma has a special meaning in BASIC. It *cannot* be used in the customary way to separate groups of three digits in a large number. For example, in BASIC the number 3,526,489 must be written without commas as **3526489.**

Try printing the number 3,526,489 with the commas by typing **?3,526,489** as shown in Figure 4.10. Note that instead of printing one number the Commodore 64/VIC 20 printed the *three* numbers **3, 526,** and **489.** In a PRINT statement the comma is used to move to the next fixed tab position. These fixed tab positions are located in columns 0 and 11 in the VIC 20 (the twenty-two screen columns are numbered 0 through 21). If you try to PRINT more than two numbers on a line, the extra numbers will be printed on the next line. On the Commodore 64, the fixed tab positions are located in columns 0, 10, 20, and 30 (the forty screen columns are numbered 0 through 39). In this case, as shown in Figure 4.11 (example 1), if you try to PRINT more than four numbers on a line, the extra numbers will print on the next line. In Figure 4.11, note that a blank space has been left in front of each number for the sign. This can be seen more clearly in the second example of Figure 4.11, where some of the numbers have negative signs. If the number contains more than seven digits on the Commodore 64 (eight digits on the VIC 20) the next tab position is skipped, as shown by the third and fourth examples of Figure 4.11. One or more commas can precede a number in order to skip tab positions, as shown in the last example of Figure 4.11.

Figure 4.10 The comma acts like a tab in a PRINT statement.



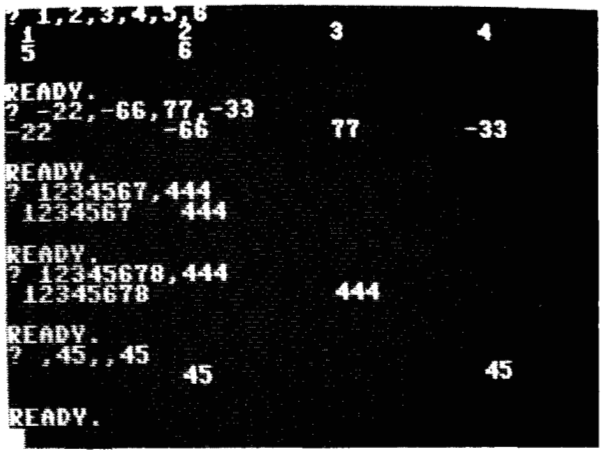
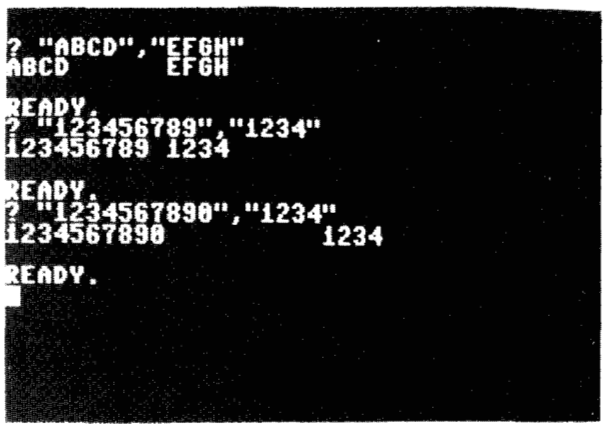


Figure 4.11 Examples of using the comma as a tab.

The comma can also be used with strings, as shown in Figure 4.12. Up to ten characters on the VIC 20 or nine characters on the Commodore 64 can be included in a string before a tab position is skipped prior to printing a second string. Strings begin printing in the beginning tab positions (columns 0 and 11 on the VIC 20; columns 0, 10, 20, 30 on the Commodore 64).

Figure 4.12 Using the comma tab with strings.



The comma can be used in PRINT statements to separate strings from numerical variables, as shown in Figure 4.13. Note that after the string "**A=**" is printed, the comma causes a tab to column 10 (11 on the VIC 20) before the value of A, **3**, is printed. This gap can be eliminated by using a semicolon instead of a comma.

```
LIST

10 A=3
20 PRINT "A=",A
30 PRINT "THE VALUE OF A IS",A
READY.
RUN
A=           3
THE VALUE OF A IS     3

READY.
```

Figure 4.13 Using the comma to separate strings and numerical variables.

Semicolon

If numerical values are separated by semicolons instead of commas, then only a single space (plus a space for the sign) is inserted after each value, as shown in Figure 4.14. Commas and semicolons can be mixed in a single PRINT statement.

When used with *strings,* the semicolon leaves no blank spaces between two strings, as shown in Figure 4.15. When combining strings and numerical values, the semicolon can be used to eliminate unsightly gaps as shown in Figure 4.16a. Note that if the value of A is negative, you may want to include a blank space at the end of the string, as in line 30 of Figure 4.16b. The use of the semicolon with strings containing graphic characters simplifies the process of creating larger graphic figures. This process will be described in detail later in this chapter.

Figure 4.14 Using the semicolon to separate numerical values.

```
? 1;2;3;4;5;6
 1  2  3  4  5  6

READY.
?-22;-66;77;-33
-22 -66  77 -33

READY.
?11;22;33;44;55
 11  22  33  44  55

READY.
```

```
? "ABCD";"EFGH"
ABCDEFGH

READY.
```

Figure 4.15 The semicolon leaves no blank spaces between strings.

Figure 4.16 *Using the semicolon to separate strings* and numerical variables.

(a)

```
LIST

10 A=3
20 PRINT "A=";A
30 PRINT "THE VALUE OF A IS";A
READY.
RUN
A= 3
THE VALUE OF A IS 3

READY.
```

(b)

```
LIST

10 A=-3
20 PRINT "A=";A
30 PRINT "THE VALUE OF A IS ";A
READY.
RUN
A=-3
THE VALUE OF A IS -3

READY.
```

SPC

The function SPC(X) can be used in a PRINT statement to move the cursor X spaces to the right. The value of X must be between 0 and 255. If the cursor reaches the end of a line, it will continue at the beginning of the next line.

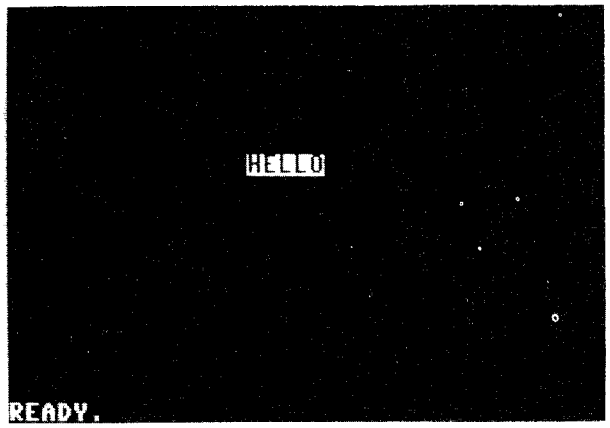
As an example, suppose you want to print the word HELLO in reverse video near the center of the screen. The program shown in Figure 4.17 will do this. The string D$ contains ten "cursor down" characters. The first string in the PRINT statement contains the "clear screen" character. Thus, the PRINT statement first clears the screen, then moves the cursor ten lines down the screen, then skips fifteen spaces using the function SPC(15), then prints **HELLO** in reverse video, and finally moves the cursor ten more lines down the screen. This causes the **READY** message to appear near the bottom of the screen. Note that the semicolon must be used between items in the PRINT statement in order to keep the cursor at its ending location after each item is processed.

Figure 4.17 The function SPC(15) will skip 15 spaces.

(a)

```
10 D$="■■■■■■■■■■"
20 PRINT"■";D$;SPC(15);"■HELLO■";D$

READY.
```

(b)



TAB

While the comma can be used to tab to the next *fixed* tab position on a line (0,10,20,30/0,11), the TAB function can be used to tab to *any* position on a line. For example, TAB(15) will move the cursor to column 15 (the sixteenth position) on the line. If SPC(15) in Figure 4.17 is replaced with TAB(15), the program will produce the same result.

However, SPC and TAB do not always produce the same result. You can see the difference in Figure 4.18. The function SPC(10) skips ten spaces after the string **"1979"**. On the other hand the function TAB(10) moves the cursor to column 10.

When using the function TAB(X) the value of X must be between 0 and 255. If the value of X is less than the position of the cursor on the line at the time of execution, the TAB function is ignored.

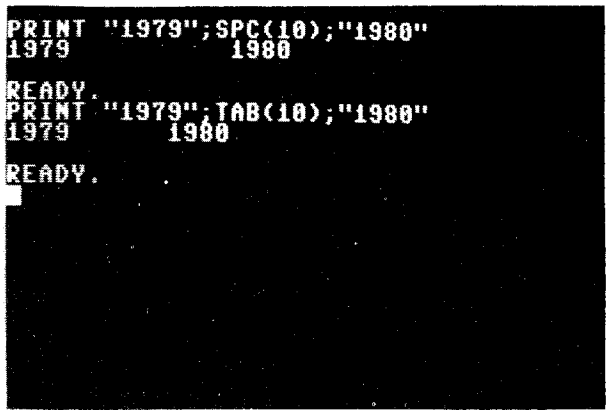
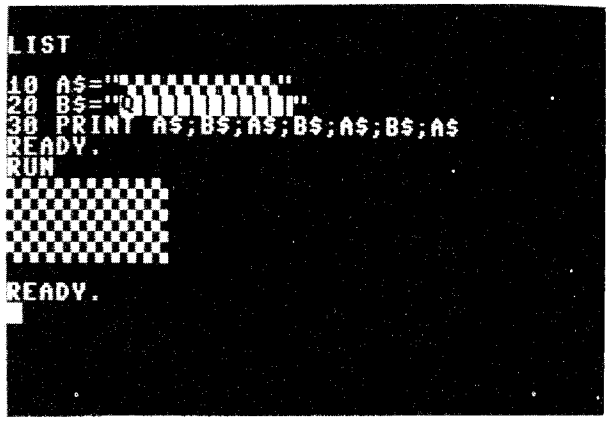


Figure 4.18 SPC and TAB will, in general, produce different results.

MORE GRAPHICS

The use of string variables and the semicolon in a PRINT statement make it easy to draw larger graphic figures. For example, suppose that A$ is a string consisting of ten repetitions of the left graphic symbol on the B key. Also let B$ be a string consisting of one "cursor down" and ten "cursor left" moves. Then a checkerboard pattern can be generated by the statement **PRINT A$;B$;A$;B$;A$;B$;A$** as shown in Figure 4.19. Note that string variables, including those defined to represent strings of graphics characters and cursor moves, can be used more than once in a PRINT statement. Note also that the string variables in the

Figure 4.19 Forming graphic figures using multiple string variables.



PRINT statement are separated by semicolons. This is required so that each successive string will pick up where the previous one left off.

A second example of drawing a graphic figure is shown in Figure 4.20, where a three-dimensional T is displayed. The strings B3$, B6$, and B9$ are cursor move strings containing one "cursor down" and three, six, and nine "cursor left" moves, respectively. Line 60 draws the top of the T and line 70 draws the vertical post of the T.

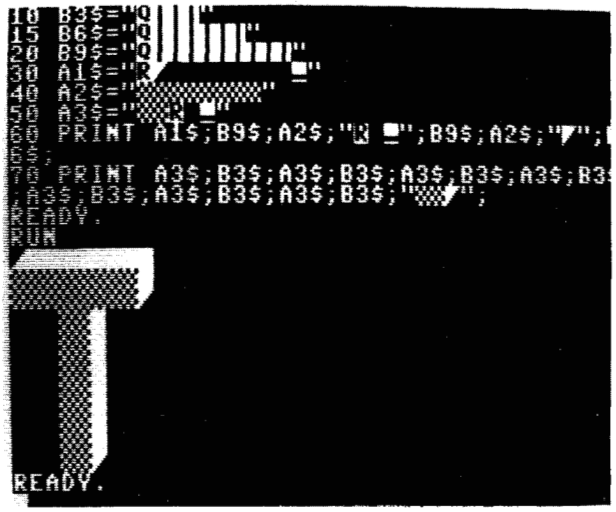
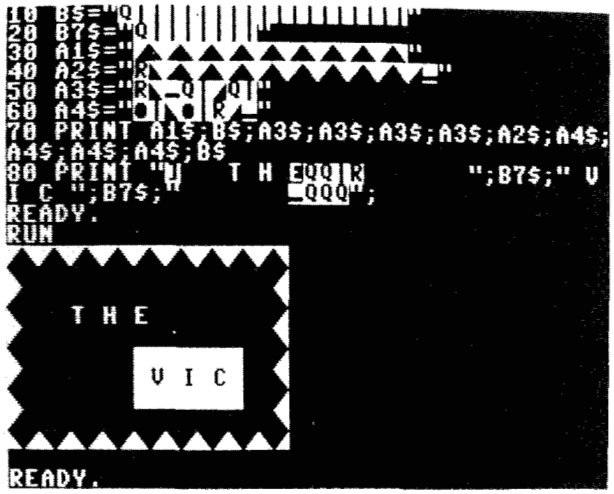


Figure 4.20 Program for drawing a three-dimensional T.

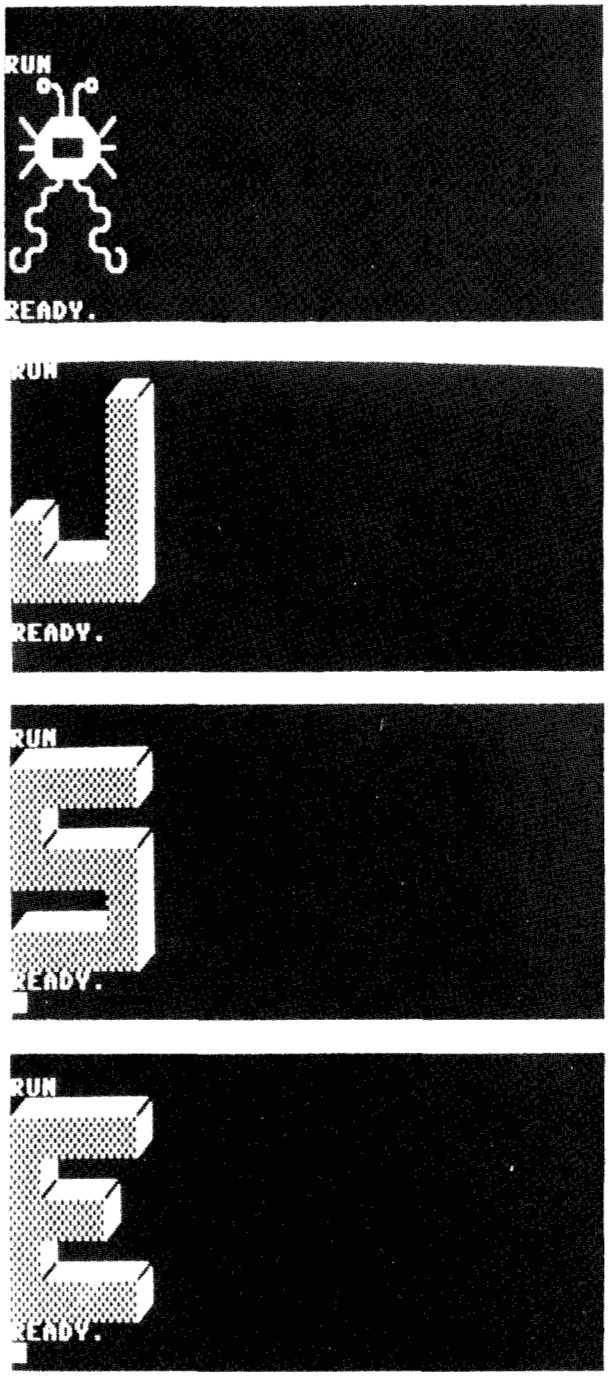
A third example of drawing a graphic figure is shown in Figure 4.21. The strings B$ and B7$ are cursor move strings containing one "cursor down" and eighteen and seven "cursor left" moves, respectively. The strings A1$ and A2$ contain the graphic characters for the top and bottom of the frame. The

Figure 4.21 Example of character graphics.



string A3$ contains the graphic characters required to draw a single triangular element on the left side of the frame. This string is called four times in the PRINT statement in line 70 in order to draw the *left side* of the frame from *top to bottom*. The string A4$ contains the graphic characters required to draw a single triangular element on the right side of the frame. This string is called four times in the PRINT statement in line 70 in order to draw the *right side* of the frame from *bottom to top*. The PRINT statement in line 70 draws the

Figure 4.22 Graphic figures for Exercise 4-1.

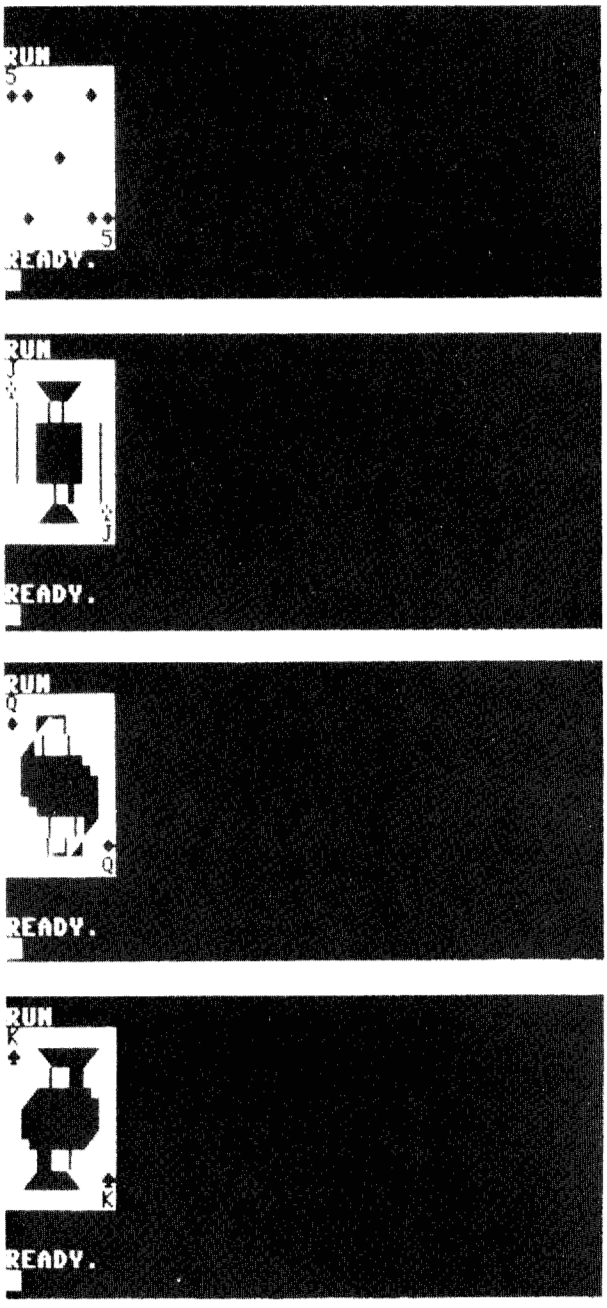


entire frame in the following order: top, left side, bottom, right side. The words **THE VIC** are printed by line 80.

EXERCISE 4-1
Write programs to draw the graphic figures shown in Figure 4.22. Answers are in the back of the book.

EXERCISE 4-2
Write programs to draw the four playing cards shown in Figure 4.23. Answers are given in the back of the book.

Figure 4.23 Playing card graphic figures for Exercise 4-2.



SOME BUILT-IN FUNCTIONS

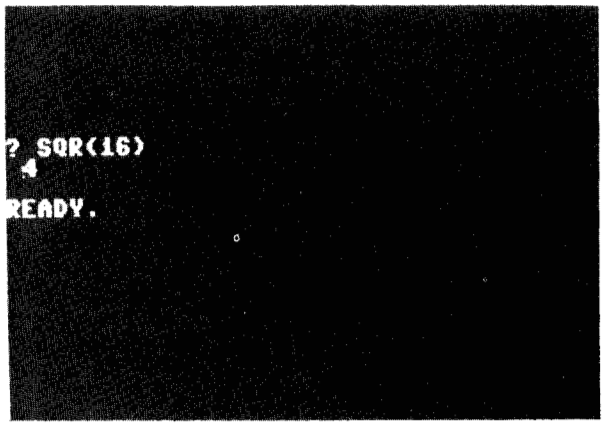
The Commodore 64/ VIC 20 has a number of built-in arithmetic functions that simplify many calculations. You may wish to ignore the descriptions of these functions until you actually need to use them.

Square Root

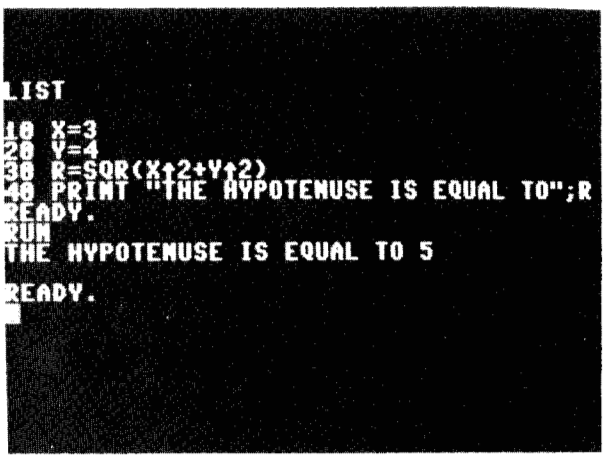
The square root of a number can be found by using the BASIC function SQR(X) where X is a non-negative number. For example, to find the square root of 16, type **?SQR(16)** as shown in Figure 4.24a. To find the hypotenuse, R, of the right triangle shown in Figure 4.25, you could use the program in Figure 4.24b.

Figure 4.24 Use of the square root function, SQR.

(a)



(b)



The Functions ABS, INT, and SGN

The *absolute value* of a number is the magnitude of a number without regard to its sign. The absolute value of a number X can be found by using the built-in function ABS(X). Thus, for example, if X=−7, then the value of ABS(X) will be 7.

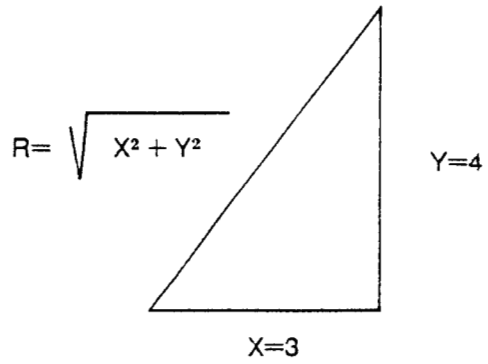


Figure 4.25 Finding the hypotenuse of a right triangle.

The value of the function INT(X) is equal to the *integer part* of X. Thus, if X=3.25 then INT(X) is equal to 3. When computing INT(X) the Commodore 64/VIC 20 will round to the next *lower whole signed number*. Thus, if X=−3.25 then the value of INT(X) will be −4.

The function SGN(X) can be used to determine the *sign of a number*. It can have the following three values:

$$\text{SGN(X)} = \begin{cases} +1 & \text{if } X>0 \\ 0 & \text{if } X=0 \\ -1 & \text{if } X<0 \end{cases}$$

Examples using ABS, INT, and SGN are shown in Figure 4.26.

Jiffies and the Time of Day

Try typing **?TI** and then type it again. The Commodore 64/VIC 20 will display two different numbers, such as those shown in Figure 4.27. (Your numbers will be different from those shown in the figure.) The Commodore 64/VIC 20 keeps track of the time that has elapsed since you turned on its power. The variable TI will always contain the number of *jiffies* (1 jiffy = 1/60 second) that have passed since power to the Commodore 64/VIC 20 was turned on.

Figure 4.26 Examples of finding the absolute value, ABS, the integer part, INT, and the sign, SGN, of a number.

```
? ABS(-3.2)
 3.2

READY.
? INT(3.2)
 3

READY.
? INT(-3.2)
-4

READY.
? SGN(3.2)
 1

READY.
? SGN(-3.2)
-1

READY.
? SGN(0)
 0

READY.
```

```
? TI
 194033

READY.
? TI
 194527

READY.
```

Figure 4.27 The variable TI contains the number of jiffies (1/60 second) that have passed since power was turned on.

You can use the variable TI if you want to see how long a particular program (or part of a program) takes to run. For example, to see how long it takes to print the T in Figure 4.20, add the following statements to the program in Figure 4.20:

```
 5 A=TI
80 B=TI
90 J=B-A
110 PRINT "THE NUMBER OF JIFFIES IS"; J
120 PRINT "THE NUMBER OF SECONDS IS":
J/60
```

The result of running this new program is shown in Figure 4.28.

Figure 4.28 Counting the number of jiffies during the drawing of the T.

```
RUN
 THE NUMBER OF JIFFIES IS  5
THE NUMBER OF SECONDS IS  .0833333333

READY.
```

Statement 5 will store in memory cell A the value of TI when the program begins. Statement 80 will store in memory cell B the value of TI after the T has been drawn. The difference B−A is then equal to the number of jiffies that have occurred between lines 5 and 80. This value is stored in memory cell J by line 90 and printed on the screen by line 110. The equivalent number of seconds is printed on the scren by line 120.

The Commodore 64/VIC 20 uses these jiffies to keep track of the number of hours, minutes, and seconds that have elapsed since power was turned on. These numbers are contained in the string TI$. Try typing **?TI$** (typing **?TIME$** will produce the same result). You will get a result similar to that shown in Figure 4.29. The first two digits are the *hours,* the second two digits are the *minutes,* and the last two digits are the *seconds.*

You can change the value stored in TI$ to correspond to the correct time, after which the Commodore 64/VIC 20 will keep time for you. For example, if it is 11:26:13, then you would type **TI$="112613".** When you press RETURN, the Commodore 64/VIC 20 will start keeping time from this value. Whenever you want to know what time it is, just type **?TI$** (see Figure 4.29).

```
? TI$
005509

READY.
TI$="112613"

READY.
? TI$
112616

READY.
```

Figure 4.29 Setting Commodore 64/VIC 20's clock.

Random Numbers

In many programs, particularly game programs, it is useful to be able to generate random numbers. These can then be used to simulate dealing cards, rolling dice, or creating other unpredictable results. BASIC has a built-in function called RND that makes generating random numbers easy.

Type the following program and run it twice, as shown in Figure 4.30:

```
10 ?RND(1)
20 ?RND(1)
30 ?RND(1)
```

```
LIST

10 PRINT RND(1)
20 PRINT RND(1)
30 PRINT RND(1)
READY.
RUN
 .185564016
 .0468986348
 .827743801

READY.
RUN
 .554749226
 .897233831
 .572916248

READY.
```

Figure 4.30 The function RND(1) produces a random number between 0 and 1.

The function RND(X) will return a pseudorandom number between 0 and 1 if the argument X is a positive number. (It does not matter what the positive value is. We will use 1.) This seems to be happening in Figure 4.30. Each time RND(1) is called, it produces a different number between 0 and 1. However, if you turn your Commodore 64/VIC 20 off and then back on and rerun the program in Figure 4.30, you will obtain the same set of "random" numbers. This is because the function RND uses a "seed" that determines the initial random number to be generated. This "seed" is set to the same value each time the Commodore 64/VIC 20 is turned on. Therefore, the first time you call RND(1) after the Commodore 64/VIC 20 is turned on, you will always get the same "random" number.

In order to change the "seed," you can call the RND function with a negative argument. For example, if you add the statement **5 X=RND(−1)** to this program, you will generate a different sequence of random numbers. However, this statement will always generate the same sequence of numbers, as shown in Figure 4.31.

A different negative argument will produce a different seed and therefore a different random sequence, as shown in Figure 4.32.

Thus, if you know which negative number is used in RND(−X) you will, in principle, know what sequence of "random" numbers will follow. One good way to make the selection of a seed more truly random is to use the "jiffy" variable as the value for the negative number. Thus, if you change line 5 to **5 X=RND(−TI)** then each time you run the program you will generate a completely different sequence of random numbers, as shown in Figure 4.33.

```
LIST

5 X=RND(-1)
10 PRINT RND(1)
20 PRINT RND(1)
30 PRINT RND(1)
READY.
RUN
 .328788872
 .978964086
 .895758909

READY.
RUN
 .328788872
 .978964086
 .895758909

READY.
```

Figure 4.31 The function RND(−1) produces a particular seed.

```
LIST

5 X=RND(-2)
10 PRINT RND(1)
20 PRINT RND(1)
30 PRINT RND(1)
READY.
RUN
 .865554613
 .846128798
 .981100118

READY.
RUN
 .865554613
 .846128798
 .981100118

READY.
```

Figure 4.32 The function RND(−2) produces a different seed.

Figure 4.33 Using RND(−TI) to produce a seed will generate different sequences of random numbers.

```
LIST

5 X=RND(-TI)
10 PRINT RND(1)
20 PRINT RND(1)
30 PRINT RND(1)
READY.
RUN
 .446082864
 .806575336
 .807437216

READY.
RUN
 .175837498
 .172576785
 .822971522

READY.
```

Trigonometric Functions

The Commodore 64/VIC 20 contains the following built-in trigonometric functions:

Function	*Value of Function*
SIN(X)	*sine of X*
COS(X)	cosine of X
TAN(X)	tangent of X
ATN(Y)	arc tangent of Y

In the above expressions, X is a numeric constant, variable, or expression that represents the value of an angle in *radians*. The value of ATN(Y) is expressed in radians in the range ± 1.57, and Y is a numeric constant, variable, or expression.

The definition of a radian is shown in Figure 4.34. To convert degrees to radians, multiply the number of degrees by $\pi/180$. Examples of using the trigonometric functions are shown in Figure 4.35.

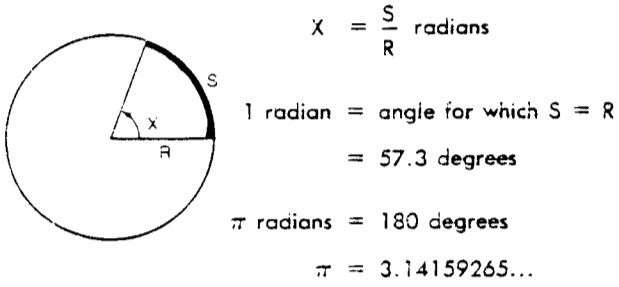


Figure 4.34 Definition of a radian.

Figure 4.35 Examples of using the trigonometric functions SIN, COS, TAN, and ATN.

```
? SIN(45*π/180)
 .707106781

READY.
? COS(60*π/180)
 .5

READY.
? TAN(60*π/180)
 1.73205081

READY.
? ATN(2)
 1.10714872

READY.
```

Natural Logarithms and the Exponential Function*

Consider the equation:

$y = b^x$

*This section may be skipped without loss of continuity.

In this expression x is called the logarithm of y to the base b and is written

$x = \log_b y$

If the base b is equal to e=2.718281 . . . , we say that y is the exponential function $y = e^x$ and x is the natural logarithm of y:

$x = \ln y$

In BASIC e^x can be computed using the function EXP(X), and ln x can be computed using the function LOG(X).

The following properties of logarithms are illustrated in the examples shown in Figure 4.36:

LOG(A*B)=LOG(A)+LOG(B)
LOG(A/B)=LOG(A)−LOG(B)
LOG(A↑K)=K*LOG(A)

```
 2.48490665

READY.
? LOG(3)+LOG(4)
 2.48490665

READY.
? LOG(9/2)
 1.5040774

READY.
? LOG(9)-LOG(2)
 1.5040774

READY.
? LOG(2.5↑3)
 2.7488722

READY.
? 3*LOG(2.5)
 2.7488722

READY.
```

Figure 4.36 Illustrating properties of logarithms.

When the *rate* at which a quantity grows is proportional to the *amount* of the quantity then we have *exponential growth.* The amount of money in a savings account on which interest is compounded *continually* grows exponentially. Thus D dollars invested at P percent annual interest, compounded continually, will yield X dollars after T years where:

$X = De^{PT/100}$

For example, to find the amount of money you would earn in seven years by investing $3000 at 9.5% interest, compounded continually, type **? 3000*EXP(9.5*7/100)** as shown in Figure 4.37.

Note that the answer is more than $5833, or almost double your original investment. A characteristic of exponential growth is a constant doubling time T_d. From the above equation for X we see that X will be equal to 2D in the time T_d where:

```
? 3000*EXP(9.5*7/100)
 5833.47156

READY.
? 100*LOG(2)
 69.3147181

READY.
```

Figure 4.37 Examples related to the exponential function.

$2D = De^{PT_d/100}$

or

$2 = e^{PT_d/100}$

Taking the natural logarithm of both sides of this equation and using the third property of logarithms illustrated in Figure 4.36, we obtain

$\ln(2) = PT_d/100 \ln(e)$
$= PT_d/100$

or

$$T_d = \frac{100 \ln(2)}{P}$$

Note that ln(e) = 1. (Try typing **? LOG(2.718281).**)

In order to see how long this doubling time is type **?100*LOG(2)** as shown in Figure 4.37. We see that the doubling time is approximately 70 divided by the percentage growth rate, or

$T_d \approx 70/P$

Thus, for example, a 10% inflation rate will double prices every seven years.

USER-DEFINED FUNCTIONS

Sometimes it is convenient to be able to define your own BASIC function. For example, suppose you want to calculate the area of a circle for different values of the circle radius. You can define a function FNA(R) that is equal to the area of a circle of radius R using the DEF FN (define function) statement as follows: **10 DEF FNA(R)=π*R↑2.** Later in the program any reference to the function FNA(R) will cause the expression π*R↑2 to be calculated. For example, the statement **20 PRINT FNA(3), FNA(5)** will print the

```
LIST

10 DEF FNA(R)=π*R↑2
20 PRINT FNA(3),FNA(5)
READY.
RUN
 28.2743339          78.5398164

READY.
```

Figure 4.38 The defined function FNA(R) computes the area of a circle of radius R.

areas of circles of radius 3 and 5. This example is shown in Figure 4.38.

The general form of the define function statement is:

DEF FNname(arg)=expression.

The *name* can be any valid variable name (one or two characters) and the *expression* can be any arithmetic expression containing numeric constants and variables. Only one argument (arg) can be passed to the user-defined function. The defining expression may, however, contain previously defined functions. For example, the two statements:

```
10 DEF FNY2(Y)=Y↑2
20 DEF FNH(X)=SQR(X↑2+FNY2(Y))
```

will define the function FNH(X) to be equal to the hypotenuse of a right triangle with sides X and Y. (See Figures 4.24 and 4.25.) If you add the statements:

```
30 X=3:Y=4
40 PRINT FNH(X)
```

and run the program, the Commodore 64/VIC 20 will print the number 5 ($5 = \sqrt{3\uparrow 2 + 4\uparrow 2}$).

EXERCISE 4-3

Let the variables A,B,C, and D have the following values: A=2, B=3, C=4, D=5. Use the Commodore 64/VIC 20 to evaluate the following expressions. The answers are given in the answer section at the end of the book.

a. $X = \left(A - \frac{C}{D}\right)^{0.5}$

b. $Z = \frac{A(B-C)}{D(B^{\wedge}-1)}$

c. $Y = \frac{(A+B)}{C(D-A)}$

d. $R = \sqrt{(A+B)/(D-A)}$

e. $S = \frac{e^{\wedge} - e^{-\wedge}}{2}$

5

ENTERING DATA FROM THE KEYBOARD: LEARNING ABOUT INPUT

In earlier chapters of this book you learned how to use the PRINT statement to make the Commodore 64/VIC 20 produce various forms of data on the screen. In this chapter you will learn how to make the Commodore 64/VIC 20 accept various forms of data that you type on the keyboard. You do this by using the INPUT statement in a BASIC program. You will learn how to use this INPUT statement by studying the following sample programs:

1. add two numbers
2. compute the area of a rectangle
3. compute the area of a circle
4. calculate gas mileage
5. draw custom checkerboard patterns.

THE INPUT STATEMENT

The INPUT statement can be used only in the deferred mode of execution. The following are some valid forms of the INPUT statement:

```
10 INPUT R
10 INPUT A,B
10 INPUT "ENTER 3 VALUES";X,Y,Z
10 INPUT A$
```

When the first of these INPUT statements is executed, the Commodore 64/VIC 20 will print a question mark and then wait for you to enter some numerical value from the keyboard. When you press the RETURN key, the value that you typed on the screen will be stored in the memory cell R. The next statement in the BASIC program will then be executed.

When the second INPUT statement is executed, the Commodore 64/VIC 20 will expect you to enter *two* numerical values, separated by a comma. If you press RETURN after entering only one value, the Commodore 64/VIC 20 will print a double question mark and wait for you to enter the second value. These two values will then be stored in the two memory cells A and B.

The third form of the INPUT statement shown will print the message **ENTER 3 VALUES** followed by a question mark. The Commodore 64/VIC 20 will then wait for you to enter three values separated by commas. These three values will then be stored in the three memory cells X,Y, and Z. If someone else (the user) will be entering data, the program should always *prompt* the user so that he or she will know when to enter data. This can either be done as shown in this example or by using a PRINT statement just before the INPUT statement.

The fourth form of the INPUT statement shown above will store whatever you type on the screen in the string variable A$. You do not need to type the quotation marks when entering a string with the INPUT statement unless you are entering some cursor move, color, or reverse video characters, or a string containing either blankspaces at the ends or a comma.

The use of the INPUT statement will be illustrated in the following five programs.

SUM OF TWO NUMBERS

Figure 5.1 shows a listing and sample run of a program that will add two numbers entered from the keyboard and display the sum. Type in this program and run it.

Line 20 prints the message **ENTER 2 NUMBERS SEPARATED BY A COMMA.** Line 30 prints a question mark on the next line and then waits for you to enter two numbers. In the first example (after **RUN**) the two numbers **5** and **9** were entered from the keyboard. Line 40 then prints the value stored in A (**5**) followed by a plus sign, followed by the value stored in B (**9**), followed by an equal sign, followed by the sum A+B (**14**). Line 50 is a PRINT statement with nothing following the word PRINT. The only purpose of this statement is to skip a line on the screen. Line 60 causes the program to branch back to line 20, which asks for another two numbers to be entered.

In the second example the value **8** is entered for the first number. But then the RETURN key was pressed. Note that the response is a double question mark asking you to enter the second number. In this example **−3** was then entered.

This program will continue to ask you for two more numbers. How can you stop the program? You will find the STOP key *does not respond* while the INPUT statement is displaying the question mark and waiting for you to enter data. The only way to end a program during an INPUT statement is to hold down the RUN/STOP key and press RESTORE. In general, this will have the side effect of clearing the screen.

Try experimenting with this program to see how it behaves. Study the program carefully and make sure you understand what every statement does.

Figure 5.1 Sample program to add two numbers.

```
LIST

10 REM PROGRAM TO ADD TWO NUMBERS
20 PRINT "ENTER 2 NUMBERS SEPARATED BY A
 COMMA"
30 INPUT A,B
40 PRINT A;" + ";B;" = ";A+B
50 PRINT
60 GOTO 20
READY.
RUN
ENTER 2 NUMBERS SEPARATED BY A COMMA
? 5,9
 5  +  9  =  14

ENTER 2 NUMBERS SEPARATED BY A COMMA
? 8
?? -3
 8  + -3  =  5

ENTER 2 NUMBERS SEPARATED BY A COMMA
? 
```

AREA OF A RECTANGLE

Figure 5.2 shows the listing and a sample run of a program that computes the area of a rectangle, where the length and width are entered from the keyboard. Type in this program and run it.

The main difference between this program and the previous one is that the *prompt* message is included in the INPUT statement in line 30. When this is done, the question mark and the blinking cursor remain on the *same line* as the message. Thus, you normally enter the data on the same line as the prompting message.

However, there is a problem with the Commodore 64/VIC 20 in this regard. When the prompt message is over one linewidth in length (22 characters on the VIC 20; 40 characters on the Commodore 64), the computer does not properly read the data you type in if the data is displayed on the same line as the last part of the prompt message. Instead, it gives the error message: **?REDO FROM START.** There is apparently a "bug" in the CBM software that causes this. One way to deal with the problem, whenever the question mark and prompt message are displayed, is to enter a "cursor down" key just before you enter the data called for. Then, the data will be displayed *on the line following* the prompt message. There, the Commodore 64/VIC 20 seems to be able to read it okay. This practice was followed throughout the examples in this book.

Figure 5.2 Sample program to calculate the area of a rectangle.

```
LIST

10 REM PROGRAM TO COMPUTE THE AREA
20 REM  OF A RECTANGLE
30 INPUT "ENTER THE 2 SIDES OF A RECTAN
LE";X,Y
40 PRINT "THE AREA OF A RECTANGLE WITH
IDES"
50 PRINT X;"AND ";Y;" IS EQUAL TO";X*Y
60 PRINT
70 GOTO 30
READY.
RUN
ENTER THE 2 SIDES OF A RECTANGLE? 4,5
THE AREA OF A RECTANGLE WITH SIDES
 4 AND  5  IS EQUAL TO 20

ENTER THE 2 SIDES OF A RECTANGLE? 6,
THE AREA OF A RECTANGLE WITH SIDES
 6 AND  0  IS EQUAL TO 0

ENTER THE 2 SIDES OF A RECTANGLE? 
```

Note in the second example of Figure 5.2 that the RETURN key was pressed after the comma was typed. When this is done, the Commodore 64/VIC 20 assigns a value of zero to the numerical variable Y.

AREA OF A CIRCLE

The area of a circle of radius r is given by

$$\text{area} = \pi r^2$$

Figure 5.3 shows the listing and a sample run of a program that computes the area of a circle whose radius is entered from the keyboard. Type in this program and run it.

This program shows that the PRINT statement in line 20 and the INPUT statement in line 30 behave the same way as the single statement

30 INPUT "ENTER THE RADIUS OF A CIRCLE";R

That is, the question mark and blinking cursor are displayed on the same line as the message. This is because the PRINT statement in line 20 ends with a *semicolon,* which always leaves the cursor at its current position. When a PRINT statement does not end with any punctuation, the equivalent of a RETURN is inserted at the end of the print statement.

Line 35 calculates the area of the circle. Recall that π is equal to 3.14159265. . . .

Note that in the second example two values, **6** and **3,** were entered. But the Commodore 64/VIC 20 was expecting only one value. It therefore used only the first value (**6**) and printed the warning message **?EXTRA IGNORED.**

In the third example a value of **2.5E19** was entered. But this results in a value of the area A that is larger than 1.7E38 and therefore the message **?OVERFLOW ERROR IN 35** is printed, indicating that the overflow occurred in the calculation of the area A. (See Figure 4.9 for an example of overflow error.)

Figure 5.3 Sample program to calculate the area of a circle.

```
LIST
10 REM PROGRAM TO COMPUTE THE AREA OF A
CIRCLE
20 PRINT "ENTER THE RADIUS OF A CIRCLE "
;
30 INPUT R
35 A=π*R↑2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
READY.
RUN
ENTER THE RADIUS OF A CIRCLE ? 35.2
THE AREA OF THE CIRCLE IS 3892.55897

ENTER THE RADIUS OF A CIRCLE ? 6,3
?EXTRA IGNORED
THE AREA OF THE CIRCLE IS 113.097336

ENTER THE RADIUS OF A CIRCLE ? 2.5E19

?OVERFLOW  ERROR IN 35
READY.
```

GAS MILEAGE

The program shown in Figure 5.4 computes gas mileage in miles per gallon. The reading of the dashboard odometer (the device that displays the mileage) at the last fill-up is stored in memory cell M1 by line 25. The odometer reading at the present fill-up is stored in memory cell M2 by line 35. The number of gallons it takes to fill the tank is stored in memory cell G by line 45. The total number of miles traveled since the last fill-up is equal to M2-M1. Therefore the number of miles per gallon is given by (M2-M1)/G. This is calculated by line 50 and stored in the memory cell MPG. It is printed on the screen by line 60. Line 70 skips a line, and line 80 branches back to line 20 to run the program again.

Figure 5.4 Program for computing gas mileage.

```
10 REM GAS MILEAGE PROGRAM
20 PRINT "ENTER PREVIOUS ODOMETER READING"
25 INPUT M1
30 PRINT "ENTER NEW ODOMETER READING"
35 INPUT M2
40 PRINT "ENTER GALLONS SINCE LAST FILLUP"
45 INPUT G
50 MPG=(M2-M1)/G
60 PRINT "GAS MILAGE:";MPG;"MILES/GAL."
70 PRINT
80 GOTO 20

READY.
```

A sample run is shown in Figure 5.5a. The answer is printed as **19.556962 MILES/GAL.** This answer contains many more digits after the decimal point than are meaningful. It probably makes sense to compute the miles per gallon only to the nearest tenth. How can we have the Commodore 64/VIC 20 display the miles per gallon to the nearest tenth? The following steps will do it:

1. Multiply the present value by 10
   19.556962×10 = 195.56962
2. Add 0.5
   195.56962+0.5 = 196.06962
3. Take the *integer part* of the result
   INT(196.06962) = 196
4. Divide by 10
   196/10 = 19.6

Although this may look complicated, it can all be done with this *single* BASIC statement:

55 MPG=INT(MPG*10+0.5)/10.

Note that the result is stored in the memory cell **MPG.** Therefore, if you add this statement to the program shown in Figure 5.4 and run the program with the same values used in Figure 5.5a, the result will be as shown in Figure 5.5b.

(a)

```
RUN
ENTER PREVIOUS ODOMETER READING
? 12345
ENTER NEW ODOMETER READING
? 12654
ENTER GALLONS SINCE LAST FILLUP
? 15.8
GAS MILAGE: 19.556962 MILES/GAL.

ENTER PREVIOUS ODOMETER READING
?
```

(b)

```
RUN
ENTER PREVIOUS ODOMETER READING
? 12345
ENTER NEW ODOMETER READING
? 12654
ENTER GALLONS SINCE LAST FILLUP
? 15.8
GAS MILAGE: 19.6 MILES/GAL.

ENTER PREVIOUS ODOMETER READING
?
```

Figure 5.5 Sample runs of gas mileage program.

CUSTOM CHECKERBOARD PATTERNS

The previous examples in this chapter have used the INPUT statement to enter numerical data into the Commodore 64/VIC 20. The program described in this section will use the INPUT statement to enter *strings* into the Commodore 64/VIC 20. In particular, the program will display an 8 × 8 checkerboard pattern made up of any two characters, including graphic characters. The program is shown in Figure 5.6. Type in this program and run it. A sample run of the program is shown in Figure 5.7.

Line 20 is an INPUT statement that asks you to enter two graphic characters that it will store in memory cells A$ and B$. (You must separate the characters by a comma as usual.) Every other line in

Figure 5.6 Program to display an 8 × 8 custom checkerboard pattern.

```
10 REM CUSTOM CHECKERBOARD PATTERNS
20 INPUT "ENTER 2 GRAPHIC CHARACTERS ";A$,B$
30 C$=A$+B$
40 D$=B$+A$
50 R1$=C$+C$+C$+C$
60 R2$=D$+D$+D$+D$
65 PRINT
70 PRINT R1$:PRINT R2$:PRINT R1$:PRINT R2$
80 PRINT R1$:PRINT R2$:PRINT R1$:PRINT R2$
90 PRINT
100 GOTO 20

READY.
```

```
RUN
ENTER 2 GRAPHIC CHARACTERS ? +,V

ENTER 2 GRAPHIC CHARACTERS ?
```

Figure 5.7 Sample run of the program in Figure 5.6.

the checkerboard pattern will begin with the A$ character, and the remaining lines will begin with the B$ character. For this reason lines 30 and 40 define two new string variables C$ and D$ that each contain the two characters A$ and B$. The string C$ starts with A$ and the string D$ starts with B$. The meanings of these strings are shown in Figure 5.8, where the strings C$ and D$ are printed after running the program shown in Figure 5.6. The string A$ contains the left graphic character on the + sign key, and the string B$ contains the right graphic character on the V key.*

Lines 50 and 60 in Figure 5.6 define new strings R1$ and R2$ that contain four copies of C$ and D$, respectively. The meanings of these two strings are also shown in Figure 5.8. You can see from the appearance of R1$ and R2$ in Figure 5.8 that every other line in the checkerboard pattern should be the string R1$, and the remaining lines should be the string R2$. Lines 70 and 80 print eight lines,

*On the VIC 20, the former character has twice as many dots and the latter has thinner legs. Thus, they "look" somewhat different than Figures 5.7 and 5.8.

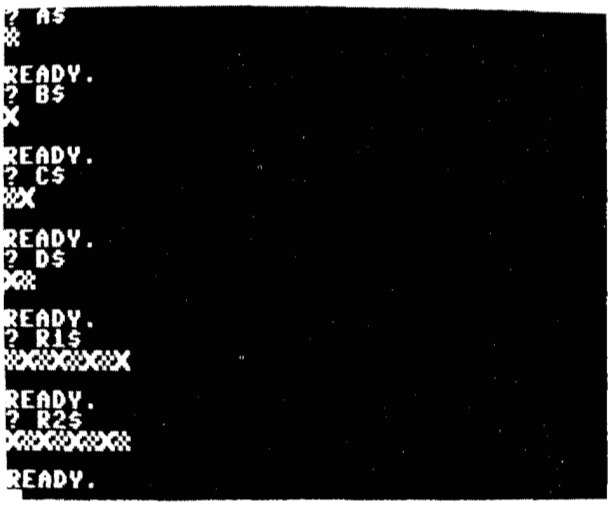


Figure 5.8 Illustrating the meaning of A$, B$, C$, D$, R1$, and R2$.

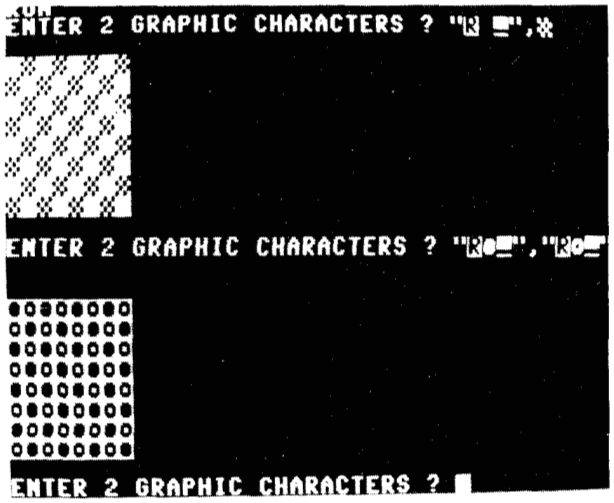


Figure 5.9 Using characters with reverse video in the program shown in Figure 5.6.

alternating R1$ and R2$. Note that we have included four PRINT statements on each of two lines only for convenience. We could have written the PRINT statements on eight different lines.

If you want to use reverse video with any of your graphic characters, then you must enter the data using quotation marks, as illustrated in the sample runs shown in Figure 5.9.

As you can see, you can make many different checkerboard patterns with this program. Some choices of graphic characters will produce unexpected results. You should try lots of different combinations. Try the two right characters on the O and P keys. Also use the left character on the B key twice. You should be able to find dozens of other interesting combinations.

EXERCISE 5-1

Run the checkerboard program shown in Figure 5.6 using:

a. the two right graphic characters on the O and P keys

b. the left graphic character on the B key twice.

EXERCISE 5-2

Write a program that will ask the user to enter his or her name, street address, and city, state, and zip code. The program should then display the name and address on the screen.

EXERCISE 5-3

The temperature in degrees Celsius (°C) is related to the temperature in degrees Fahrenheit (°F) by the formula:

$$°C = \frac{5}{9}(°F - 32)$$

Write a program that will accept a temperature in °F entered from the keyboard and print on the screen the temperature in both °F and °C.

6 MAKING CHOICES: LEARNING ABOUT IF...THEN

All of the programs that we have written so far have consisted of a simple sequence of instructions. The Commodore 64/VIC 20 simply does what it is told and executes one statement after another. However, the thing that makes computers appear to be smart is their ability to make decisions. The primary decision-making statement in CBM BASIC is the IF...THEN statement. This statement allows the Commodore 64/VIC 20 to branch to one of two possible statements, depending upon the truth or falsity of a particular logical expression. A *logical expression* is an expression that can be either *true* or *false*.

In this chapter you will learn:

1. to use the IF...THEN statement to make simple choices
2. the meaning of the Commodore 64/VIC 20's relational operators
3. the meaning of the Commodore 64/VIC 20's logical operators
4. about the *if...then...else* statement, flowcharts, and structured flowcharts.

THE IF...THEN STATEMENT

The IF...THEN statement in CBM BASIC allows your program to conditionally execute some statements or to conditionally branch to some statement. Here are three different forms of the IF...THEN statement:

50 IF *logical expression* **THEN** *statement*
50 IF *logical expression* **THEN** *statement 1: statement 2: . . .*
50 IF *logical expression* **THEN** *line number*

In each of these forms the *logical expression* is some BASIC expression that is either *true* or *false*. These expressions will normally contain relational operators (such as <) and/or logical operators (such as OR). These operators will be defined and discussed in detail in this chapter.

In the first form of the IF...THEN statement shown above, if the logical expression is *true*, the *statement* following the word THEN is executed. This can be any BASIC statement that can be executed conditionally. If the *logical expression* is *false*, the statement with the *next* line number is executed.

The second form of the IF...THEN statement shown above behaves somewhat like the first form. However, if the *logical expression* is *true*, all of the *statements* following the word THEN are executed. The only limitation is that all of these statements must fit on the four VIC 20 or two Commodore 64 screen lines associated with the line number of the IF...THEN statement. Remember that if the logical expression is

false, the statement with the *next* line number is executed.

In the third form of the IF...THEN statement shown above, if the *logical expression* is *true,* the program will branch to the specified line number. This form is equivalent to the first form if the statement is a GOTO statement. Thus, for example, these two statements are equivalent:

50 IF A<0 THEN 90
50 IF A<0 THEN GOTO 90

In fact, the word THEN can be omitted in the second form, and you can write:

50 IF A<0 GOTO 90

We will illustrate the use of the IF...THEN statement by adding some conditional statements to the programs we wrote in Chapter 5.

Gas Mileage Program

In the gas mileage program shown in Figure 5.4 of Chapter 5, M1 is the old odometer reading and M2 is the new odometer reading. To make sense M2 must be greater than M1 (M2>M1). It is always a good idea to have the program check the data entered through the keyboard to try to detect any typing errors. For example, if after entering the value of M2 in line 35, M1 is greater than M2, a typing error has probably been made. We could add these statements to the program in Figure 5.4, as shown in Figure 6.1:

37 IF M1>M2 THEN PRINT "READING TOO SMALL": GOTO 20.

A sample run of this new program is shown in Figure 6.2. Note that during the first execution, the last digit of the new odometer reading was omitted. This made M2<M1. Statement 37 caught it and printed the message **READING TOO SMALL** and then branched back to statement 20, where the program started over again.

In statement 37 you might have branched back to statement 30 and asked only for the new odometer reading. However, the error may have occurred when entering M1 (you may have typed an extra digit) and therefore it is better to re-enter both odometer readings.

```
RUN
ENTER PREVIOUS ODOMETER READING
? 12345
ENTER NEW ODOMETER READING
? 1265
READING TOO SMALL
ENTER PREVIOUS ODOMETER READING
? 12345
ENTER NEW ODOMETER READING
? 12654
ENTER GALLONS SINCE LAST FILLUP
? 15.8
GAS MILAGE: 19.556962 MILES/GAL.

ENTER PREVIOUS ODOMETER READING
?
```

Figure 6.2 *Program will check to make sure that M2 is greater than M1.*

Circle Program

In the circle program shown in Figure 5.3 the radius should obviously be positive. Actually, if you want to calculate only the area of the circle given by πr^2, then a negative radius will give the same answer as the same positive radius. On the other hand if we also calculate the circumference of the circle given by $2\pi r$, the radius must be positive. We can calculate the circumference by adding these two statements to the program in Figure 5.3:

45 C=2*π*R
47 PRINT "CIRCUMFERENCE="; C

Figure 6.1 Gas mileage program containing an IF...THEN statement.

```
10 REM GAS MILEAGE PROGRAM
20 PRINT "ENTER PREVIOUS ODOMETER READING"
25 INPUT M1
30 PRINT "ENTER NEW ODOMETER READING"
35 INPUT M2
37 IF M1>M2 THEN PRINT "READING TOO SMALL": GOTO 20
40 PRINT "ENTER GALLONS SINCE LAST FILLUP"
45 INPUT G
50 MPG=(M2-M1)/G
60 PRINT "GAS MILAGE:";MPG;"MILES/GAL."
70 PRINT
80 GOTO 20

READY.
```

We can then test to see if the radius is negative by adding the statement:

32 IF R<0 THEN PRINT "RADIUS MUST BE POSITIVE": GOTO 20

If the value of R entered in the INPUT statement on line 30 is less than 0, then the message **RADIUS MUST BE POSITIVE** will be printed, and the program will branch back to line 20 and ask for another radius to be entered.

We saw in Figure 5.3 that if the radius is too large, an overflow error will occur when the area is computed by line 35. Since the value of the area A cannot be greater than 1.7E38, the largest radius R that will not result in an overflow can be found as follows:

$$A = \pi r^2 \leq 1.7E38$$
$$r^2 \leq 1.7E38/\pi$$
$$r \leq \sqrt{1.7E38/\pi}$$

Thus, if R>SQR(1.7E38/π) the area will be greater than 1.7E38 and cause an overflow. We can test this by adding the following statement to the program:

33 IF R>SQR(1.7E38/π) THEN PRINT "RADIUS TOO LARGE": GOTO 20

The complete revised program is shown in Figure 6.3, and a sample run is shown in Figure 6.4. Note the use of the two IF...THEN statements in lines 32 and 33. The first IF...THEN statement checks to see if R is less than 0. If this is *false* (if R is positive), then the next IF...THEN statement on line 33 is executed. If R is not greater than SQR(1.7E38/π), the program will continue with line 35.

```
RUN
ENTER THE RADIUS OF A CIRCLE ? -3
RADIUS MUST BE POSITIVE
ENTER THE RADIUS OF A CIRCLE ? 2.5E20
RADIUS TOO LARGE
ENTER THE RADIUS OF A CIRCLE ? 5.6
THE AREA OF THE CIRCLE IS 98.5203457
CIRCUMFERENCE= 35.1858377

ENTER THE RADIUS OF A CIRCLE ?
```

Figure 6.4 Sample run of program in Figure 6.3.

Rectangle Program

As another example of using the IF...THEN statement to check data entered with the INPUT statement, consider the program shown in Figure 5.2 that computes the area of a rectangle. It is clear that *both*

Figure 6.3 Modified circle program that checks the value of the radius R.

```
10 REM PROGRAM TO COMPUTE THE AREA OF A CIRCLE
20 PRINT "ENTER THE RADIUS OF A CIRCLE ";
30 INPUT R
32 IF R<0 THEN PRINT "RADIUS MUST BE POSITIVE":GOTO 20
33 IF R>SQR(1.7E38/π) THEN PRINT "RADIUS TOO LARGE":GOTO 20
35 A=π*R↑2
40 PRINT "THE AREA OF THE CIRCLE IS";A
45 C=2*π*R
47 PRINT "CIRCUMFERENCE=";C
50 PRINT
60 GOTO 20

READY.
```

Figure 6.5 The IF...THEN statement in line 35 contains a compound logical expression.

```
10 REM PROGRAM TO COMPUTE THE AREA
20 REM  OF A RECTANGLE
30 INPUT "ENTER THE 2 SIDES OF A RECTANGLE";X,Y
35 IF X<0 OR Y<0 THEN PRINT "VALUES MUST BE POSITIVE":GOTO 30
40 PRINT "THE AREA OF A RECTANGLE WITH SIDES"
50 PRINT X;"AND ";Y;" IS EQUAL TO";X*Y
60 PRINT
70 GOTO 30

READY.
```

dimensions of a rectangle must be positive. Thus, if *either* of the two values entered via the INPUT statement on line 30 is negative, the program should print an error message and ask for new input. We can do this by adding the following IF...THEN statement:

35 IF X<0 OR Y<0 THEN PRINT "VALUES MUST BE POSITIVE": GOTO 30

The resulting program is shown in Figure 6.5, and a sample run is shown in Figure 6.6. Note from this sample run that the Commodore 64/VIC 20 will not allow the program to continue if either value entered is negative or if both are negative. Thus, the meaning of the logical expression X<0 OR Y<0 is that it is *true* if either X<0 or Y<0 is true, or if both are true.

In this logical expression the symbol < (less than) is one of the *relational operators*. The word OR is one of the *logical operators*. Relational operators and logical operators will be discussed in more detail in the next two sections.

```
RUN
ENTER THE 2 SIDES OF A RECTANGLE? -2,6
VALUES MUST BE POSITIVE
ENTER THE 2 SIDES OF A RECTANGLE? 3,-6
VALUES MUST BE POSITIVE
ENTER THE 2 SIDES OF A RECTANGLE? -5,-9
VALUES MUST BE POSITIVE
ENTER THE 2 SIDES OF A RECTANGLE? 6,8
THE AREA OF A RECTANGLE WITH SIDES
 6 AND  8  IS EQUAL TO 48

ENTER THE 2 SIDES OF A RECTANGLE?
```

Figure 6.6 Sample run of program in Figure 6.5.

RELATIONAL OPERATORS

A *relational operator* is used to form a logical expression by comparing two arithmetic expressions. (An arithmetic expression can be a numerical constant, variable, or expression.) For example, A<0 is a logical expression (it is either true or false) which uses the relational operator <. If the content of memory cell A is less than zero, this logical expression is true; otherwise, it is false.

The Commodore 64/VIC 20 stores the logical value "false" as a zero (0). It stores the logical value "true" as −1. You can see this by typing

```
A=3
?A<3
```

and

```
A=-3
?A<0
```

as shown in Figure 6.7. Note that you can print the value of logical expressions such as A<0.

```
A=3

READY.
? A<3
 0

READY.
A=-3

READY.
? A<0
-1

READY.
```

Figure 6.7 The Commodore 64/VIC 20 stores "true" as −1 and "false" as 0.

The relational expressions used in the Commodore 64/VIC 20 are given in Table 6.1. Figure 6.8 shows some examples using these relational operators. You should try some examples of your own.

TABLE 6.1 Relational operators

Operator	*Meaning*
=	equal to
<> or ><	not equal to
<	less than
>	greater than
<= or =<	less than or equal to
>= or =>	greater than or equal to

Figure 6.8 Examples of logical expressions formed using the relational operators.

```
? 6-2=4
-1

READY.
? 5*2<>10
 0

READY.
? 8>SQR(16)
-1

READY.
? 3<=SQR(9)
-1

READY.
? 20>=5↑2
 0

READY.
```

LOGICAL OPERATORS

In addition to the relational operators the Commodore 64/VIC 20 uses the three logical operators NOT, AND, and OR. The meanings of these operators are shown in Table 6.2

TABLE 6.2 Logical Operators

A and B are logical expressions

A	*NOT A*
true	false
false	true

A	*B*	*A AND B*	*A OR B*
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

NOT

The logical operator NOT is a unary operator; that is, it operates on a single logical expression, A. If A is *true*, then NOT A is *false*. If A is *false*, then NOT A is *true*. Examples using the logical operator NOT are shown in Figure 6.9.

AND

The logical operator AND is a *binary* operator that operates on *two* logical expressions. As shown in Table 6.2, A AND B is *true* only if *both* A and B are true. It is *false* if either A or B is false, or if both are false. Examples using the logical operator AND are shown in Figure 6.10.

Figure 6.9 Examples using the logical operator NOT.

```
? NOT 3=3
 0

READY.
? NOT 5<2
-1

READY.
? NOT 10<>5*2
-1

READY.
? NOT 6>=SQR(25)
 0

READY.
```

```
? 2=2 AND 3=3
-1

READY.
? 2=2 AND 3>4
 0

READY.
? 7<=5 AND 8<10
 0

READY.
? 4<5 AND NOT 7<5
-1

READY.
```

Figure 6.10 Examples using the logical operator AND.

OR

The logical operator OR is also a binary operator. As shown in Table 6.2, A OR B is *false* only if *both* A and B are false. It is *true* if either A or B is true, or if both are true. Examples using the logical operator OR are shown in Figure 6.11.

Note that the third example in Figure 6.11 is *false* while the fourth example is *true*. The only difference between the two is the inclusion of the parentheses in the third example. The fourth example is true because the AND operation is performed *before* the OR operation.

Figure 6.11 Examples using the logical operator OR.

```
? 5< 3 OR 6>=5
-1

READY.
? NOT 6=6 OR 7<>7
 0

READY.
? (2=2 OR 3=3) AND 1=2
 0

READY.
? 2=2 OR 3=3 AND 1=2
-1

READY.
```

There is an order of precedence for logical and relational operators as well as for arithmetic operators. When the Commodore 64/VIC 20 evaluates an expression, it uses the order of precedence shown in Table 6.3.

Within each level of precedence the expression is evaluated from left to right.

TABLE 6.3 Order of Precedence for Evaluating Expressions

Operator	*Meaning*
( )	Parentheses
↑	Exponentiation
−	Unary Minus
*,/	Multiplication and Division
+,−	Addition and Subtraction
=, <>, <, >, <=, >=	Relational Operators
NOT	Logical Complement
AND	Logical AND
OR	Logical OR

WEEKLY PAY PROGRAM

As another example of using the IF...THEN statement consider the problem of calculating the weekly pay of an employee whose hourly rate is $4.00 and who receives time-and-a-half for overtime. Suppose that the total hours worked per week cannot exceed sixty hours. We want to write a program that will:

1. ask for the number of hours worked to be entered from the keyboard
2. check to make sure that the number of hours entered is not greater than sixty
3. check to make sure that the number of hours entered is not negative
4. compute the pay at $4.00 per hour for the first forty hours and at $6.00 per hour for any hours over forty
5. print the total amount of pay.

The program to do this is shown in Figure 6.12. Lines 20 and 30 ask for the number of hours to be entered (INPUT statement) and show that the value is stored in H. Line 40 checks to make sure that the number of hours is not greater than 60, and line 50 checks to make sure that it is not negative.

Line 60 will compute the total pay to be **M=H*4** if H is less than or equal to 40. Note that this line ends with the statement **GOTO 90,** which branches to statement 90. Line 90 rounds the value of M to two places after the decimal point (see discussion of gas mileage program in Chapter 5). Line 100 prints the amount of pay.

If H is greater than 40, the logical expression **H<=40** in line 60 will be false, and line 70 will be executed next. Line 70 computes the number of overtime hours (OV). Line 80 computes the total pay (M), consisting of the first forty hours at $4.00 per hour plus the remaining overtime hours at $6.00 per hour (**M=40*4+OV*6**). Lines 90 and 100 round and print the total pay.

Sample runs of this program are shown in Figure 6.13. Note that trailing zeros are not printed on the screen. Thus, for example, $233.50 is printed as **$233.5.** (In Chapter 13 we will see how to make both positions of the cents field appear on the screen, even when the right-most position contains a zero.)

Figure 6.12 Listing of weekly pay program.

```
10 REM PROGRAM TO COMPUTE WEEKLY WAGES
20 PRINT "ENTER NUMBER OF HOURS WORKED"
30 INPUT H
40 IF H>60 THEN PRINT "TOO MANY HOURS":GOTO 20
50 IF H<0 THEN PRINT "INVALID DATA":GOTO 20
60 IF H<=40 THEN M=H*4:GOTO 90
70 OV=H-40
80 M=40*4+OV*6
90 M=INT(M*100+0.5)/100
100 PRINT "WEEKLY PAY= $";M

READY.
```

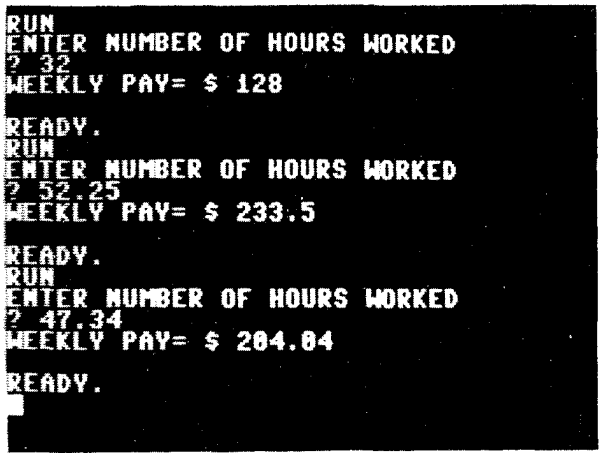


Figure 6.13 Sample runs of program in Figure 6.12.

AREA OF A TRIANGLE

The area of the triangle shown in Figure 6.14 can be calculated from the formula

$$\text{AREA} = [S(S-A)(S-B)(S-C)]^{0.5}$$
$$= \sqrt{S(S-A)(S-B)(S-C)}$$

where A, B, and C are the sides of the triangle and

$$S = \frac{1}{2}(A+B+C)$$

is the semi-perimeter.

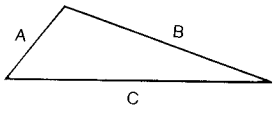
In BASIC the formula for the area can be written as

AREA=(S*(S−A)*(S−B)(S−C))↑0.5

or

AREA=SQR(S*(S−A)*(S−B)*(S−C))

Remember that the multiplication symbol * must *always* be explicitly typed, and every left parenthesis must have an accompanying right parenthesis.



Semi-perimeter, $S = \frac{1}{2}(A+B+C)$

$$\text{Area} = [S(S-A)(S-B)(S-C)]^{0.5}$$

Figure 6.14 Finding the area of a triangle.

We want to write a program that will ask the user to enter the lengths of the three sides of a triangle from the keyboard and will then display the area of the triangle on the screen. It should be clear that not all combinations of three numbers can represent the sides of a triangle. For example, a triangle cannot be formed with the three sides 10, 5, and 3, as shown in Figure 6.15. From this figure you can see that to form a triangle the sum of A + B must be greater than C, where C is the longest side. This is equivalent to requiring C to be less than the semi-perimeter S = (A+B+C)/2. If this were not true, the formula for the area would involve taking the square root of a negative number, which is not a real value.

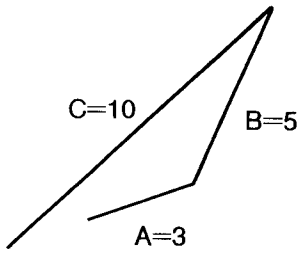


Figure 6.15 To form a triangle the following relations must be true: A + B > C or C < S=1/2(A+B+C).

Therefore, our program should check to make sure that the three numbers entered from the keyboard can really represent the sides of a triangle. We need to check to make sure that the longest side is less than S. The longest side may be the first, second, or third number to be entered from the keyboard. If the program uses the INPUT statement **INPUT A,B,C** then the longest side may actually be stored in memory cell A,B, or C. Therefore, the program must find the longest side, L, and then make sure that L is less than the semi-perimeter, S.

We can determine the largest number stored in memory cells A,B, and C by the following procedure:

1. Compare A and B
   if A>B
   then set L=A
   else set L=B
2. Compare C and L
   if C > L
   then set L=C

Convince yourself that this *algorithm,* or step-by-step procedure, will in fact put the largest value into memory cell L. This value of L can then be compared to the semi-perimeter S to see if a triangle is possible.

The BASIC program to do all of this is shown in Figure 6.16. Line 20 asks for the three sides of the triangle to be entered, and line 30 stores these three values in A,B, and C. Line 40 compares A and B and if

Figure 6.16 Program to find the area of a triangle.

```
10 REM PROGRAM TO FIND THE AREA OF A
15 REM  TRIANGLE
20 PRINT "ENTER THE THREE SIDES OF A TRIANGLE"
30 INPUT A,B,C
40 IF A>B THEN L=A: GOTO 60
50 L=B
60 IF C>L THEN L=C
70 S=(A+B+C)/2
80 IF L>S THEN PRINT "NO TRIANGLE POSSIBLE": GOTO 20
90 AREA=(S*(S-A)*(S-B)*(S-C))↑0.5
100 PRINT "THE AREA OF THE TRIANGLE IS";AREA
110 PRINT
120 GOTO 20

READY.
```

A>B, it stores the value of A in L and branches to line 60. If A is not greater than B, line 50 will store the value of B in L. Thus, when line 60 is executed, L will contain the larger of A and B. Line 60 compares C and L and, if C is greater than L, it stores the value of C in L. Therefore, by the time line 70 is executed, L will contain the largest of the numbers stored in A,B, and C.

Line 70 computes the semi-perimeter, S, and line 80 compares L and S to see if a triangle is possible. If L is

```
RUN
ENTER THE THREE SIDES OF A TRIANGLE
? 7,12,9
THE AREA OF THE TRIANGLE IS 31.3049517

ENTER THE THREE SIDES OF A TRIANGLE
? 7,12,4
NO TRIANGLE POSSIBLE
ENTER THE THREE SIDES OF A TRIANGLE
? 3,5,7
THE AREA OF THE TRIANGLE IS 6.49519054

ENTER THE THREE SIDES OF A TRIANGLE
? 3,5,9
NO TRIANGLE POSSIBLE
ENTER THE THREE SIDES OF A TRIANGLE
? 
```

Figure 6.17 Sample runs of the program in Figure 6.16

greater than or equal to S, the message **NO TRIANGLE POSSIBLE** is printed, and the program branches back to line 20 and asks for three new sides. But if L is less than S, line 90 is executed to compute the area of the triangle. Line 100 prints the result. Line 110 skips a line on the screen, and line 120 branches back to line 20 to run the program again. Sample runs of this program are shown in Figure 6.17.

RANDOM CHECKERBOARD PATTERNS

In Chapter 5 we wrote a program to generate 8 × 8 custom checkerboard patterns. In this section we will write a program that will continuously fill the screen with a "random" checkboard pattern. The pattern will consist of two graphic characters A$ and B$. However, instead of alternating the characters as we did in the checkerboard pattern (See Figure 5.6), we will print at each screen location either the character A$ or the character B$, depending upon whether the random number RND(1) is greater than or less than 0.5. This means that each location will have an equal chance of being either an A$ or a B$ character.

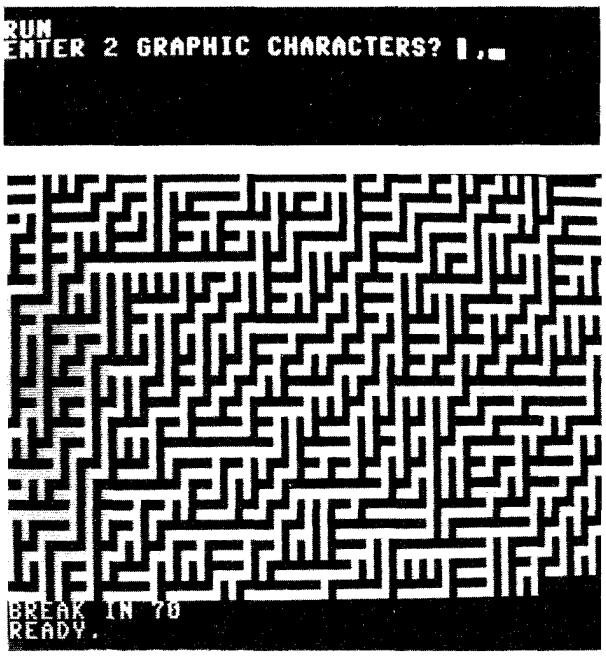
The program for displaying this random pattern is shown in Figure 6.18. Line 30 requests two graphic characters to be entered from the keyboard and stores them in the string variables A$ and B$. Line 40 stores a seed for the random number generator, using the number of jiffies stored in TI (see discussions of jiffies and random numbers in Chapter 4). Line 50 clears the screen. Line 60 stores a random number between 0 and 1 in memory cell X. Line 70 compares X to 0.5 and, if X<0.5, prints the character stored in A$ in the next screen location. Note that the PRINT statement ends with a semicolon so that the cursor will stay positioned at the next screen location. The program then branches back to line 60 to obtain another random number. If X is not less than 0.5 in line 70, then line 80 will be executed. This will print the character stored in B$ and then return to line 60 to obtain another random number. The program will continue until the STOP key is pressed.

A sample run of this program is shown in Figure 6.19. In this instance, the two graphic characters used are the left ones on the I and K keys. You should try running this program using graphic characters of your choice.

Figure 6.18 Program for displaying random checkerboard pattern.

```
10 REM PROGRAM FOR GENERATING A RANDOM
20 REM  CHECKERBOARD PATTERN
30 INPUT "ENTER 2 GRAPHIC CHARACTERS";A$,B$
40 X=RND(-TI)
50 PRINT "{CLR}"
60 X=RND(1)
70 IF X<0.5 THEN PRINT A$;:GOTO 60
80 PRINT B$;:GOTO 60

READY.
```

Figure 6.19 Sample run of program shown in Figure 6.18.



IF...THEN...ELSE

In this chapter we have used the BASIC IF...THEN statement in the form of an *if...then...else* statement. For example, in the program to find the area of a triangle we used the following algorithm to find the largest value in A,B, and C and store it in L:

if A>B
then L=A
else L=B
if C > L
then L=C

We coded this algorithm in BASIC as follows:

```
40 IF A>B THEN L=A: GOTO 60
50 L=B
60 IF C>L THEN L=C
```

The *if...then...else* statement is available in other programming languages such as PASCAL, but it is not directly available in BASIC. With the full form of the statement, the *else* statements are automatically skipped after the *then* statements are executed. However, in BASIC we must tell the computer to skip the *else* statements by ending the *then* statements with GOTO.

The *else* is generally optional in an *if...then...else* statement. Without *else,* it is like the BASIC IF...THEN statement. The restriction here is that the number of statements following THEN must fit on four VIC 20 or two Commodore 64 screen lines. If they do not, you can still simulate an *if...then...else* statement in BASIC as follows:

```
100 IF A>B THEN 120
110 GOTO 170
120 ----------
130 ----------     THEN statements
140 ----------
150 ----------
160 GOTO 210
170 REM ELSE
180 ----------
190 ----------     ELSE statements
200 ----------
210
```

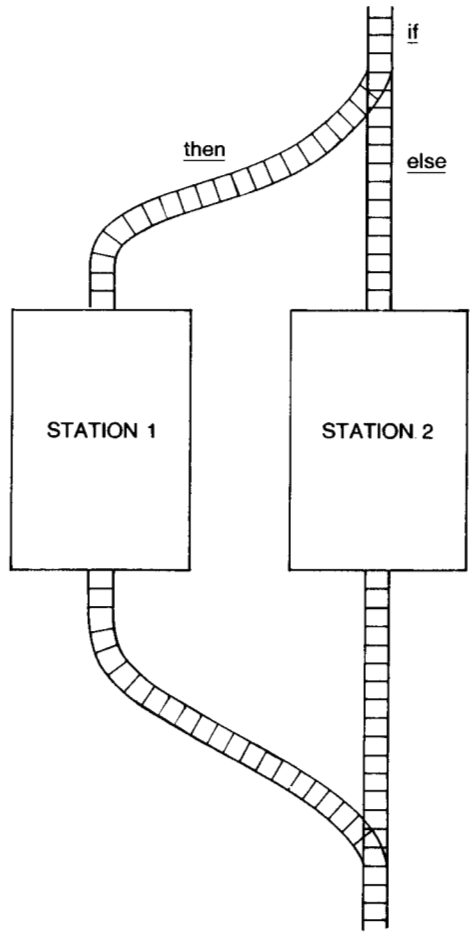
Note that line 110 is executed if A>B is *false*. This will branch to the ELSE statements. If A>B is *true* then line 120, the first of the THEN statements, is executed. You can use as many lines as you need for the THEN statements. However, at the end of the THEN statements you must include a GOTO statement that will skip over the ELSE statements.

In Chapter 3 we said that a computer program is like a train going on a trip. The seats in the train are like memory locations with unique names or addresses that distinguish one seat from another. The "seats" may contain strings (such as the name of the person sitting in the seat) or numerical values (such as the age of the person sitting in the seat).

As the train goes along the track it can come to a station where new people can get on, some people can get off, or others can exchange seats or add things to their seats. This is equivalent to executing BASIC statements such as PRINT, INPUT, and A=B+C.

The *if...then...else* statement is like a *switch* in the track that allows the train to go on one of two different paths, as shown in Figure 6.20. These two paths lead to two different stations and then merge on the other side of the stations. If the logical expression following *if* is true, the train follows the track to station 1, where the *then* statements are executed. If the logical expression following *if* is false, the train follows the track to station 2, where the *else* statements are executed. Note

Figure 6.20 The *if...then...else* statement "takes the train" to one of two possible stations.



that the train can only go either to station 1 or station 2. It *cannot* go to both stations, neither in sequence nor simultaneously.

Flowcharts and Pseudocode

Flowcharts have traditionally been used to express computer algorithms. The *if...then...else* statement illustrated in Figure 6.20 can be represented as a flowchart, as shown in Figure 6.21. The similarity to Figure 6.20 is obvious. If the logical expression in the diamond-shaped box is *true,* the path to statements A is followed. Otherwise, the path to statements B is followed.

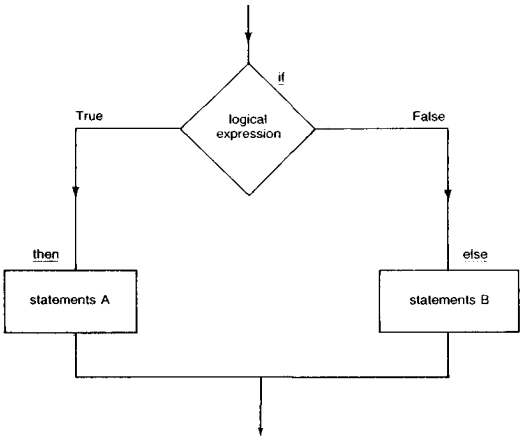
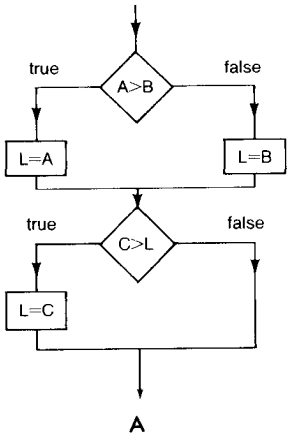


Figure 6.21 Flowchart representation of the *if...then...else* statement.

The algorithm given earlier for finding the largest value in A,B, and C is expressed as a flowchart and in *pseudocode* (using *if...then...else*) in Figure 6.22. Many people find pseudocode representations to be easier to write than flowcharts, and just as easy to understand. In addition, it is easy to generate flowcharts that end up looking like "bowls of spaghetti." For these reasons the use of flowcharts has declined in recent years. Remember that a pseudocode is just a way of expressing an algorithm and will not be understood by the Commodore 64/VIC 20.

For those who prefer a graphic representation of an algorithm, *structured flowcharts* can be used.

Figure 6.22 (a) Flowchart and (b) pseudocode for algorithm to find the largest value in A, B, and C.




```
if A>B
then L=A
else L=B
if C>L
then L=C
B
```

Structured Flowcharts

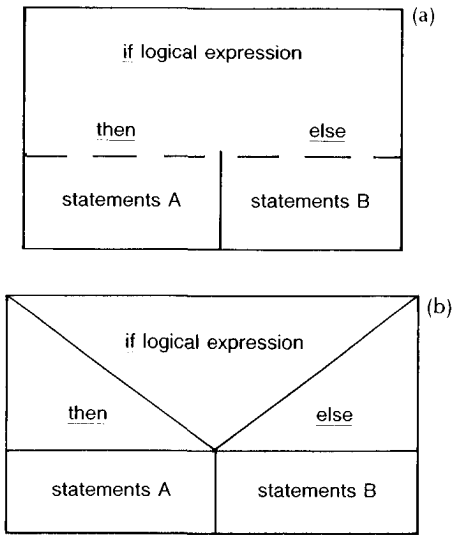
A structured flowchart (also called Nassi-Schneiderman charts after the people who introduced them) is an alternate representation of an algorithm. It consists of various nested "boxes" without the connecting lines shown in Figure 6.22. Two alternate representations of the *if...then...else* statement are shown in Figure 6.23. We will use the form shown in Figure 6.23a. Using a structured flowchart, the algorithm shown in Figure 6.22 can be represented as shown in Figure 6.24.

Flowcharts and pseudocode are just different ways of representing an algorithm to try to make it easier to understand. When developing a computer program, it is generally easier to express the program in the form of a flowchart, structured flow chart, or pseudocode before coding it in BASIC.

The structured flowchart and pseudocode for the weekly pay program discussed earlier in this chapter are shown in Figures 6.25a and 6.25b. The BASIC listing of this program is shown in Figure 6.25c. You should carefully compare these three representations of the same program.

The advantage of the structured flowchart representation is that it clearly displays the logic of the program

Figure 6.23 Two forms of a structured flowchart that represents the *if...then...else* statement.



in a graphic form. The advantage of the pseudocode is that it describes the algorithm in a simple and straightforward manner. Note the importance of the indentation in the pseudocode description. The advantage of the BASIC representation is that it can be executed on the Commodore 64/VIC 20.

Some people have devised a variety of indentation conventions that will make a BASIC program easier to understand. If you try to indent your programs, you will discover that the Commodore 64/VIC 20 automatically removes leading blanks when LISTing a program. You can overcome this problem by typing a colon (:) as the first character following the line number. Any blanks that you type following the colon will not be removed when the program is LISTed. However, these blanks will use up memory in the computer. You should always keep a written version of your program on paper. This version can include indentation, pseudocode, structured flowcharts, or anything else that will help you to understand the program.

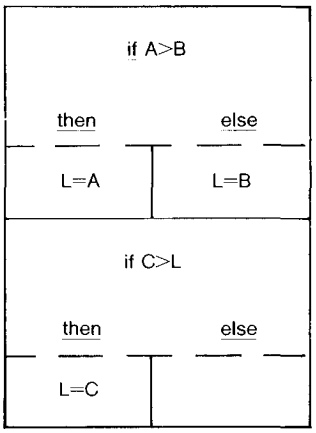


Figure 6.24 Structured flowchart representation of algorithm to find the largest value in A, B, and C.

The complete structured flowchart for the program to find the area of a triangle is shown in Figure 6.26a. The BASIC listing of this program is shown in Figure 6.26b. You should compare carefully the structured flowchart and the BASIC listing. Note that the GOTO statement in line 120 is represented in the structured flowchart as an "outer loop" that continues forever (or until the program is STOPped).

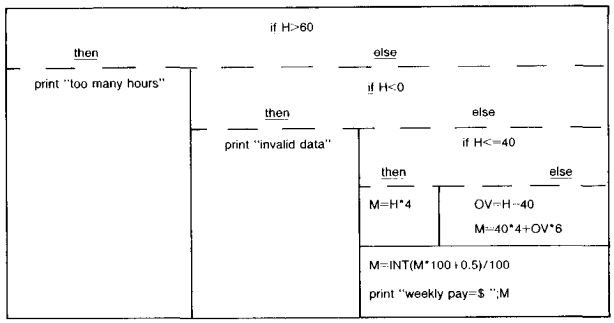
In the next chapter we will take a closer look at *loops*. In particular you will learn how to stop a loop any time you want.

EXERCISE 6-1

For married taxpayers filing joint returns, with a taxable income between $20,200 and $24,600, the Federal income tax is $3,273 plus 28% of the amount over $20,200. Write a program that will accept from the keyboard a taxable income, check that it is between $20,200 and $24,600, and then compute and print the income tax.

Figure 6.25 (a) Structured flowchart, (b) pseudocode, and (c) BASIC listing of weekly pay program.

(a)



(b)

```
if H>60
then print "too many hours"
else if H<0
    then print "invalid data"
    else if H<=40
        then M=H*4
        else OV=H-40
            M=40*4+OV*6
        M = INT(M*100+0.5)/100
        print "weekly pay = $" ;M
```

(c)

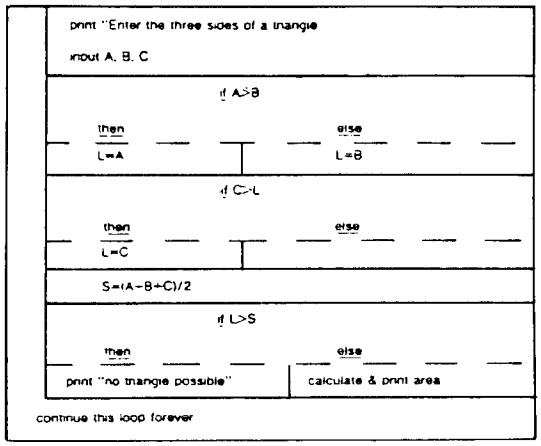
```
10 REM PROGRAM TO COMPUTE WEEKLY WAGES
20 PRINT "ENTER NUMBER OF HOURS WORKED"
30 INPUT H
40 IF H>60 THEN PRINT "TOO MANY HOURS":GOTO 20
50 IF H<0 THEN PRINT "INVALID DATA":GOTO 20
60 IF H<=40 THEN M=H*4:GOTO 90
70 OV=H-40
80 M=40*4+OV*6
90 M=INT(M*100+0.5)/100
100 PRINT "WEEKLY PAY= $";M

READY.
```

EXERCISE 6-2

Write a program to compute take-home pay. The program should accept from the keyboard an hourly wage and the number of hours worked. Assume that 6.65% of the gross pay is deducted for Social Security taxes, 14.8% of the gross pay is deducted for Federal income taxes, and 4% of the gross pay is deducted for state income taxes. The program should print out the wage rate, the number of hours worked, the amount

Figure 6.26 (a) Structured flowchart and (b) BASIC listing of program to find the area of a triangle.

(a)



(b)
```
10 REM PROGRAM TO FIND THE AREA OF A
15 REM  TRIANGLE
20 PRINT "ENTER THE THREE SIDES OF A TRIANGLE"
30 INPUT A,B,C
40 IF C>L THEN L=A: GOTO 60
50 L=B
60 IF C>L THEN L=C
70 S=(A+B+C)/2
80 IF L>S THEN PRINT "NO TRIANGLE POSSIBLE": GOTO 20
90 AREA=(S*(S-A)*(S-B)*(S-C))↑.5
100 PRINT "THE AREA OF THE TRIANGLE IS";AREA
110 PRINT
120 GOTO 20

READY.
```

deducted for Social Security, Federal and state income taxes, and the take-home pay.

EXERCISE 6-3

Write a program that will accept from the keyboard a continuous series of test scores. When a negative score is entered the program should print the number of scores entered, the largest score, the smallest score, and the average of the test scores.

LEARNING ABOUT LOOPS: ANOTHER LOOK AT IF...THEN

In Chapter 6 we used the IF...THEN statement to choose between two alternatives. We saw that this use of the IF...THEN statement was equivalent to using an *if...then...else* statement. In this chapter we will use the IF...THEN statement for a completely different purpose—that of forming loops. Because you are using the same IF...THEN statement, it may appear that there is no difference between the use of IF...THEN to form loops and its use to form an *if...then...else* construction. But there is a fundamental difference between loops and an *if...then...else* statement. An *if...then...else* statement merely makes a decision between two different paths. A loop, on the other hand, implies repetition in which the same statements are executed over and over again until (or while) some condition is met.

In this chapter you will learn:

1. to use the IF...THEN statement to form a *repeat while* loop
2. to repeat a loop while an affirmative answer is given to a question
3. to use nested loops
4. the difference between a *repeat while*, a *repeat until*, a *do while*, and a *do until* loop and how to implement these loops in BASIC
5. how to implement a *loop...exit if...endloop* and a *loop...continue if...endloop* construction in BASIC.

THE REPEAT WHILE LOOP

Very often you will have a sequence of BASIC statements that you will want to repeat as long as a particular logical expression is *true*. For example, you may wish to do the following

40 -------

50 -------

60 -------

repeat lines 40-60 *while* A>0

You can do this with the statement **70 IF A>0 THEN 40.**

Lines 40-70 form a *loop* that is exited only when A>0 becomes false, that is, when A<=0. In order to get out of the loop, there must be something in lines 40-60 that will eventually cause A to become less than or equal to zero.

Here are some examples.

RANDOM CHECKERBOARD PATTERNS

The program for generating a random checkerboard pattern was described in Chapter 6 and is shown in Figure 6.18. Review this program and key it in again (or load it in from tape or disk, if you saved it). Remember that the only way to stop this program is to press the STOP key. It would be much neater if the

program could stop itself after printing as many screen lines as we want.

We can modify the program to ask for the number of lines to be printed and to print only that number of lines. The pseudocode and structured flowchart for this modified program are shown in Figure 7.1. Each time through the loop only *one* character is printed, and N counts the number of characters that have been printed. If you have a VIC 20, let **WIDTH = 22** and, if you have a Commodore 64, let **WIDTH = 40.** Since each line contains **WIDTH** characters, **NLINE*WIDTH** characters will have been printed when **NLINE** lines have been printed. Therefore, in order to print **NLINE** lines we must repeat the loop *while* **N<NLINE*WIDTH.**

We can modify the BASIC program shown in Figure 6.18 to agree with the algorithm in Figure 7.1 by making the following changes. Add the following statements:

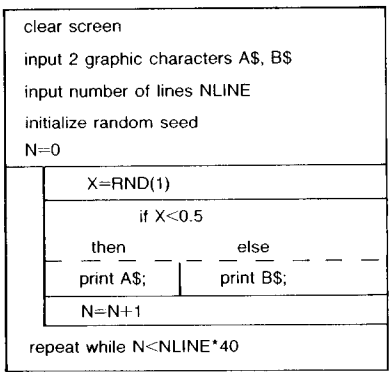
25 PRINT "clear screen"
35 INPUT "ENTER THE NUMBER OF LINES";NLINE
45 N=0
90 N=N+1
100 IF N<NLINE*WIDTH THEN 60

In lines 70 and 80 of the original program change **GOTO 60** to **GOTO 90** and delete line 50.

The resulting BASIC program is shown in Figure 7.2. Compare this listing carefully with the pseudocode and structured flowchart shown in Figure 7.1.

Figure 7.1 (a) Pseudocode and (b) structured flowchart for modified random checkerboard program.

```
clear screen
input 2 graphic characters A$,B$
input number of lines NLINE
initialize random seed
N=0
loop:  X=RND(1)
       if X<0.5
       then print A$
       else print B$
       N=N+1
repeat while N<NLINE*40
```



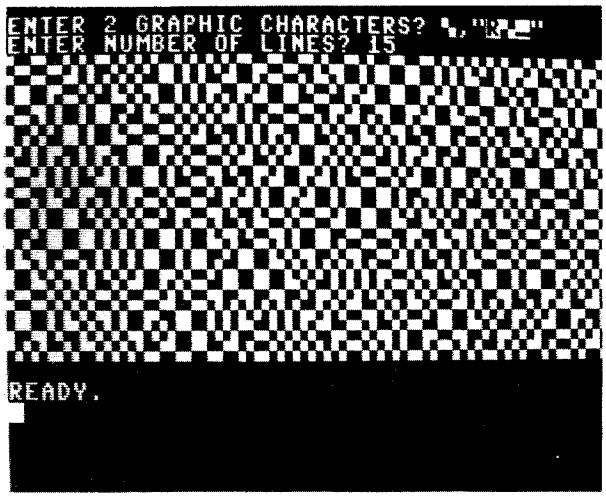
Note especially how the IF...THEN statement is used to implement both the *if...then...else* statement and the *repeat while* statement. Note also that we have included an END statement at the end of the program. Although this is optional it does show explicitly where the IF...THEN statement in line 100 branches if **N<NLINE*WIDTH** is false.

A sample run of this program is shown in Figure 7.3. Try generating other patterns with this program.

Figure 7.2 BASIC listing for modified random checkerboard pattern.

```
10 REM  PROGRAM FOR GENERATING A
20 REM  RANDOM CHECKERBOARD PATTERN
25 PRINT "J"
26 WIDTH=40
30 INPUT "ENTER 2 GRAPHIC CHARACTERS";A$,B$
35 INPUT "ENTER NUMBER OF LINES";NLINE
40 X=RND(-TI)
45 N=0
60 X=RND(1)
70 IF X<0.5 THEN PRINT A$;:GOTO 90
80 PRINT B$;:GOTO 90
90 N=N+1
100 IF N<NLINE*WIDTH THEN 60
110 END

READY.
```

Figure 7.3 Sample run of program shown in Figure 7.2.



TRIANGLE PROGRAM

A program to find the area of a triangle was discussed in Chapter 6, and the BASIC listing is given in Figure 6.16. Because of the GOTO statement in line 120, this program is executed over and over again until the STOP key is pressed. A better way to end the program would be to ask the user if he or she wants to continue. This can be done by replacing the **GOTO 20** statement on line 120 with the following statements:

120 INPUT "DO YOU WANT TO CONTINUE (Y,N)";A$
130 IF A$="Y" THEN 20
140 END

Line 120 displays the message **DO YOU WANT TO CONTINUE (Y,N)?** and then waits for a response to be entered from the keyboard. This response is stored in the string A$. Line 130 compares this string to "Y" and if **A$="Y"** the program branches back to line 20 and the area of another triangle is found. Any other response will terminate the program.

The BASIC listing of this modified program is shown in Figure 7.4, and a sample run is shown in Figure 7.5. Note that the INPUT statement in line 120 provides the question mark for the message, and you do not need to include it in the string. Also remember that if the response to an INPUT statement is expected to be a non-numeric value, a string variable rather than a numerical variable must be used in the INPUT statement. If the INPUT statement contains a numerical variable, and a non-numeric value is typed in, the Commodore 64/VIC 20 will respond with the message **?REDO FROM START.** It will then wait for a numeric value to be entered. An INPUT statement containing a string variable will accept any input, but will treat it as a string. Thus, in line 130 in Figure 7.4 the variable A$ must be compared to the *string* "Y".

```
RUN
ENTER THE THREE SIDES OF A TRIANGLE
? 6,8,9
THE AREA OF THE TRIANGLE IS 23.5252524

DO YOU WANT TO CONTINUE (Y,N)? Y
ENTER THE THREE SIDES OF A TRIANGLE
? 4,6,8
THE AREA OF THE TRIANGLE IS 11.61895

DO YOU WANT TO CONTINUE (Y,N)? N

READY.
```

Figure 7.5 Sample run of program shown in Figure 7.4.

NESTED LOOPS

The program shown in Figure 5.6 of Chapter 5 displays a custom 8 × 8 checkerboard pattern. In this section we will write a program that will display a checkerboard pattern of varying width and height.

Each "elementary square" of the checkerboard will occupy a 2 × 2 area of the screen, as shown in Figure 7.6. A$ and B$ are the strings containing the two graphic characters that will make up the checkerboard. We will assume that the square shown in Figure 7.6 has a width of 1 and a height of 1. We will limit the maximum width of the checkerboard to 10 on the VIC 20 (20 screen locations) and 19 on the Commodore 64 (38 screen locations) and the maximum height to 9 (18 screen locations).

Figure 7.6 Elementary square of the checkerboard.

A$	B$
B$	A$

The input section of the program will do the following:

clear the screen
input the two characters A$,B$
input the width W
verify that W is in the range 1-19 [1-10 on the VIC]

Figure 7.4 BASIC listing of modified triangle program.

```
10 REM  PROGRAM TO FIND THE AREA OF A
15 REM  TRIANGLE
20 PRINT "ENTER THE THREE SIDES OF A TRIANGLE"
30 INPUT A,B,C
40 IF C>L THEN L=A: GOTO 60
50 L=B
60 IF C>L THEN L=C
70 S=(A+B+C)/2
80 IF L>S THEN PRINT "NO TRIANGLE POSSIBLE": GOTO 20
90 AREA=(S*(S-A)*(S-B)*(S-C))↑.5
100 PRINT "THE AREA OF THE TRIANGLE IS";AREA
110 PRINT
120 INPUT "DO YOU WANT TO CONTINUE (Y,N)";A$
130 IF A$="Y" THEN 20
140 END

READY.
```

input the height H
verify that H is in the range 1-9

This input section corresponds to lines 10-45 of the program listing shown in Figure 7.7. (On the VIC 20, change 19 to 10 in both lines 30 and 35.) Note that line 35 means "*repeat line 30 while W<1 or W>19.*" Therefore, only values of the width W between 1 and 19 (10 on the VIC 20) will allow the program to proceed. Similarly, line 45 means "*repeat line 30 while H<1 or H>9.*"

Figure 7.7 BASIC listing of program to generate checkerboard pattern of varying width and height.

```
10 REM CHECKERBOARD PATTERN
15 PRINT "□"
20 INPUT "ENTER 2 GRAPHIC CHARACTERS";A$,B$
30 INPUT "ENTER WIDTH OF PATTERN,1-19";W
35 IF W<1 OR W>19 THEN 30
40 INPUT "ENTER HEIGHT OF PATTERN,1-9";H
45 IF H<1 OR H>9 THEN 40
50 NH=0
60 NW=0
70 PRINT A$;B$;:NW=NW+1
80 IF NW<W THEN 70
90 PRINT: NW=0
100 PRINT B$;A$;:NW=NW+1
110 IF NW<W THEN 100
120 PRINT: NH=NH+1
130 IF NH<H THEN 60
140 END

READY.
```

Lines 50-140 correspond to the algorithm shown in pseudocode and as a structured flowchart in Figure 7.8. The outer loop counts the number of rows, NH, of the checkerboard that have been printed. Note that each row uses two screen lines (see Figure 7.6). Inside this outer loop are two *nested* inner loops. The first one prints one screen line of length W starting with the character A$. The second loop prints one screen line of length W starting with the character B$. At the end of each line, it is necessary to insert a PRINT statement that will cause the cursor to return to the beginning of the next screen line.

Study this algorithm carefully and compare Figure 7.8 with lines 50-140 in the BASIC program shown in Figure 7.7. Key in and run this program. A sample run is shown in Figure 7.9. Try lots of different graphic characters.

DIFFERENT TYPES OF LOOPS

There are really four different elementary loop structures. You can test the logical expression at the beginning of the loop or at the end of the loop, and you can branch out of the loop when the logical expression

(a)

```
NH=0
    loop: NW=0
    loop: PRINT A$;B$;          prints one screen line start-
          NW=NW+1               ing with A$
    repeat while NW<W
    PRINT                       returns cursor to start of next
                                line
    NW=0                        prints one screen line start-
    loop: PRINT B$;A$;          ing with B$
          NW=NW+1
    repeat while NW<W
    PRINT                       returns cursor to start of next
                                line
    NH=NH+1
repeat while NH<H
```

(b)

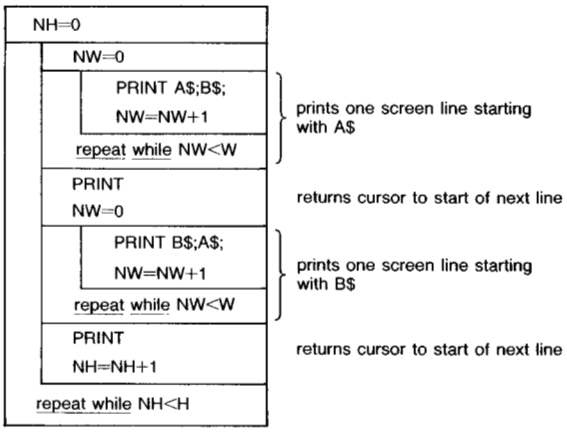
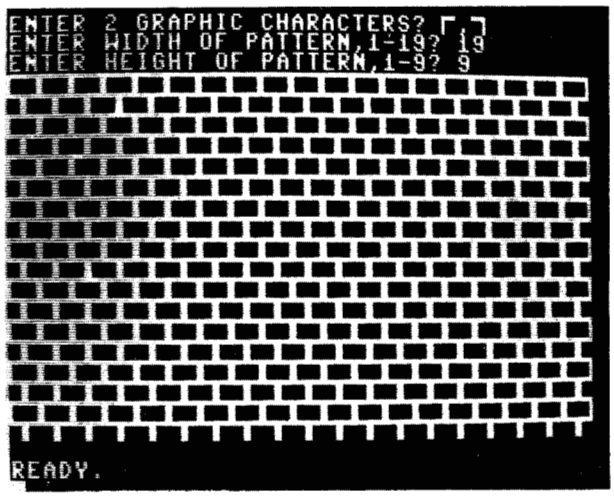


Figure 7.8 (a) Pseudocode and (b) structured flowchart representation of algorithm to produce checkerboard pattern.

is *true* or when it is *false*. We will call the two loops with the test at the end of the loop the *repeat while* and the *repeat until* loops. We will call the two loops with the test at the beginning of the loop the *do while* and the *do until* loops.

Figure 7.9 Brick wall pattern as example of running program shown in Figure 7.7.



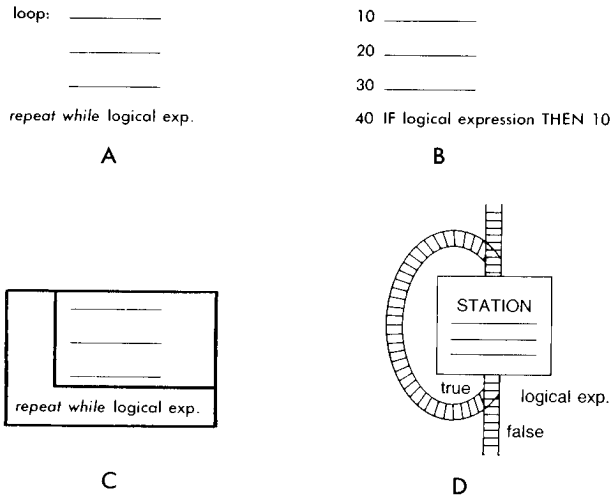
In addition to these elementary loops it is possible to use a more general loop structure in which the logical expression is tested other than at the beginning or end. Depending upon whether the loop exit is made when the logical expression is true or when it is false, we will call the general loop structures *loop...exit if...endloop* and *loop...continue if...endloop.*

All of these loop structures can be implemented in BASIC, although some are easier to implement than others. Most good programmers use only two or three of these loop structures. The choice depends on the programming language being used and, to some extent, on personal preference.

The Repeat While Loop

This is the loop that we have been using so far in this chapter. Its general form is shown in Figure 7.10. In this figure *logical exp.* is any logical expression that is either true or false. This loop is *repeated while* the logical expression is *true.* Figure 7.10d shows what this loop looks like in our train track model of a computer program. Note that the train continues to loop around and through the station as long as the logical expression is true.

Figure 7.10 The *repeat while* loop: (a) Pseudocode. (b) BASIC implementation. (c) Structured flowchart. (d) Train track equivalent.



The Repeat Until Loop

The general form of the *repeat until* loop is shown in Figure 7.11. In this case the loop is *exited* if the logical expression is true. That is, the loop is repeated *until* the logical expression is true. In general you should choose to use either the *repeat while* or the *repeat until* loop in your programs. This will help you avoid logical errors because you will always be thinking either "*while*" or "*until.*" Many people prefer the *repeat until* loop, and some languages implement this loop directly.

However, by comparing Figures 7.10b and 7.11b you can see that it is easier to implement a *repeat while* loop in BASIC because the *repeat until* implementation requires an additional GOTO statement. Therefore, when we form a loop with the test at the end of the loop, we will make it a *repeat while* loop.

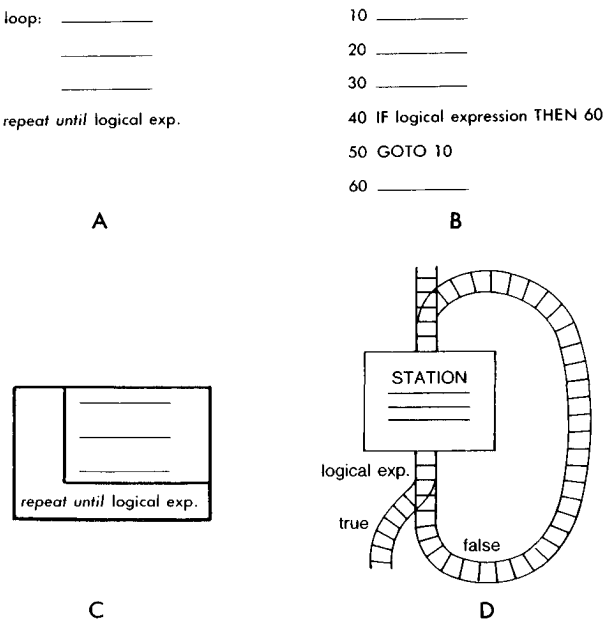


Figure 7.11 The *repeat until* loop: (a) Pseudocode. (b) BASIC implementation. (c) Structured flowchart. (d) Train track equivalent.

The Do While Loop

The *do while* loop is one of those useful programming statements that is found in newer languages such as PASCAL. Its general form is shown in Figure 7.12. In this loop the logical expression is tested at the *beginning* of the loop. This means that if the logical expression is initially *false,* the statements within the loop will never be executed. The BASIC implementation of the *do while* loop requires *two* GOTO statements, one following the IF...THEN statement to skip over the loop statements if the logical expression is false, and one at the end of the loop to branch back to the IF...THEN statement.

The Do Until Loop

The fourth elementary loop structure is the *do until* loop, whose general structure is shown in Figure 7.13. In this loop the logical expression is also tested at the

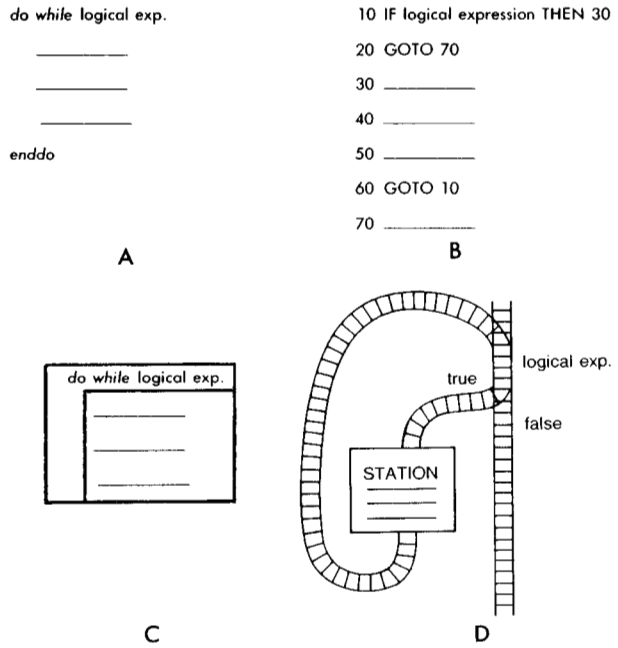


Figure 7.12 The *do while* loop: (a) Pseudocode. (b) BASIC implementation. (c) Structured flowchart. (d) Train track equivalent.

beginning of the loop. However, the statements within the loop are executed only if the logical expression is *false,* that is, *until* the logical expression is *true.* If the logical expression is initially *true,* the statements within the loop will never be executed.

The BASIC implementation of the *do until* loop requires only one GOTO statement. For this reason we will normally implement the *do until* loop rather than the *do while* loop when we need a test at the beginning of a loop.

People who write structured programs using a "good" structured programming language use the *do while* and the *repeat until* loops. In BASIC it will save us some code (and therefore some memory) if instead we use the *do until* and the *repeat while* loops.

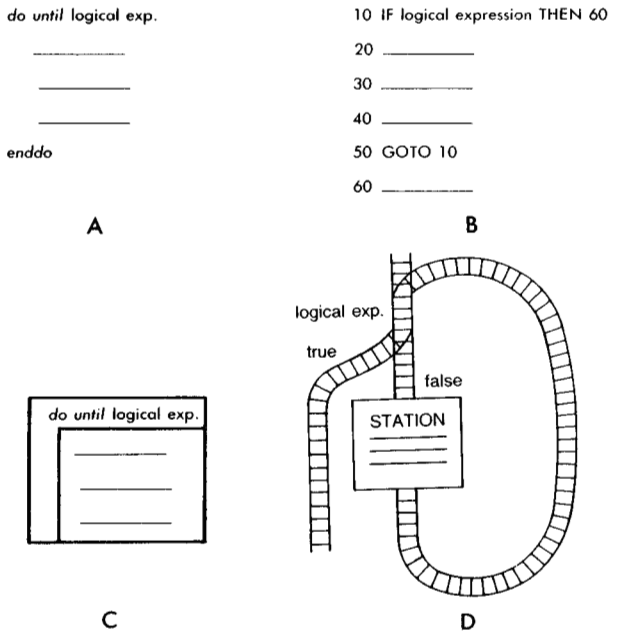


Figure 7.13 The *do until* loop: (a) Pseudocode. (b) BASIC implementation. (c) Structured flowchart. (d) Train track equivalent.

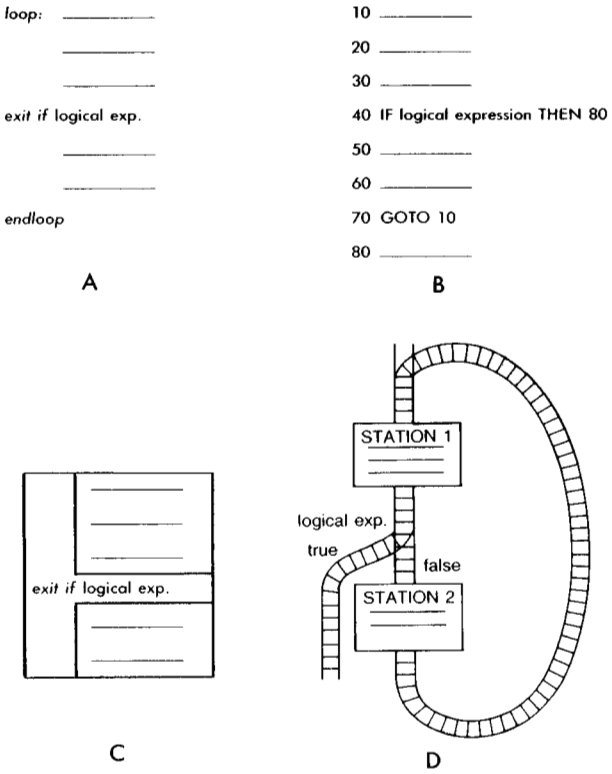


Figure 7.14 The *loop...exit if...endloop* loop: (a) Pseudocode. (b) BASIC implementation. (c) Structured flowchart. (d) Train track equivalent.

The loop...exit if...endloop Loop

Occasionally it is convenient to use a more general looping structure. Such a loop is the *loop...exit if...endloop* construction, whose general form is shown in Figure 7.14. This is really a generalized *until* loop. If the *exit if* statement is at the top of the loop, it becomes the *do until* loop. At the bottom of the loop, it becomes the *repeat until* loop.

The loop...continue if...endloop Loop

In order to complete the discussion of loops, the general form of the *loop...continue if...endloop* construction is shown in Figure 7.15. This is really a generalized *while* loop. If the *continue if* statement is at the top of the loop, it becomes the *do while* loop. If

the *continue if* statement is at the bottom of the loop, it becomes the *repeat while* loop.

Any of these loops can be used in your programs, but the easiest ones to implement in BASIC are the *repeat while,* the *do until,* and the *loop...exit if...endloop* structures.

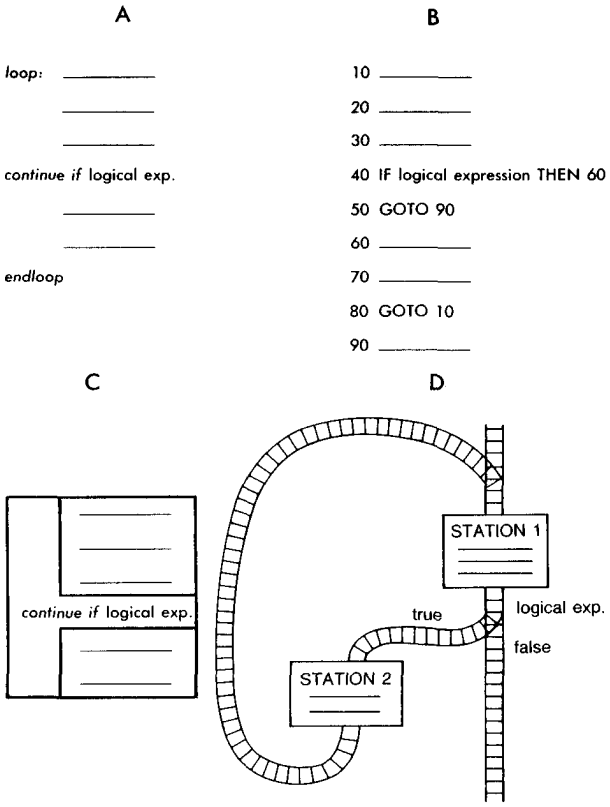
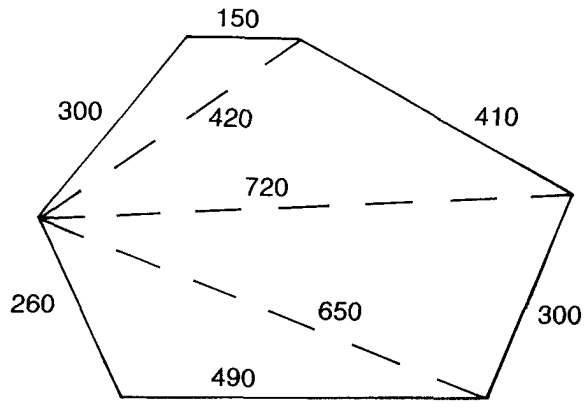


Figure 7.15 The *loop...continue if...endloop* loop: (a) Pseudocode. (b) BASIC implementation. (c) Structured flowchart. (d) Train track equivalent.

EXERCISE 7-1

The dimensions, in feet, of a tract of land are shown in this figure:

Exercise 7-1



Modify the program shown in Figure 7.4 to calculate the acreage of this tract of land. The total acreage can be found by computing the area of each of the four triangles and adding the results. One acre = 43,560 square feet.

EXERCISE 7-2

Suppose that the tract of land shown in Exercise 7-1 contains a circular pond 200 feet in diameter completely within its boundaries. Write a program that will compute the acreage of the land, excluding the water.

EXERCISE 7-3

The Fibonacci sequence

1 1 2 3 5 8 13 21 . . .

has the property that each number in the sequence (starting with the third) is the *sum* of the two numbers immediately preceding it. Write a program that will display on the screen all numbers in the Fibonacci sequence that are less than 1000.

EXERCISE 7-4

You decide to deposit an amount of money, D, in a savings account each month. The account pays P percent interest, compounded monthly. Write a program that will accept D and P as inputs from the keyboard and then determine the number of years (and months) that it will take for you to accumulate a million dollars.

The amount of interest added to the account each month is determined in the following way: If B is the balance in the account at the beginning of the month, then at the end of the month an amount of interest B*MR is added to the account. (MR is the monthly interest rate, equal to 0.01*P/12.) Thus, the total amount of money in the account at the end of the month will be equal to B+B*MR.

Run the program for the following cases:

a. deposit $500 per month at 8% interest

b. deposit $1,000 per month at 10% interest

c. deposit $1,000 per month at 12% interest.

EXERCISE 7-5

Manhattan Island was purchased from the Indians in 1626 for $24. If that $24 had been deposited in 1626 in a bank paying 4% interest, compounded annually, what would it be worth today?

EXERCISE 7-6

If you deposit $100 each year in a bank account paying 5% interest, compounded annually, how much money will you have after ten years?

EXERCISE 7-7

Population growth. In 1980 the U.S. birth rate was 16.2 births per 1000 population, the death rate was 8.9 deaths per 1000 population, and the net migration rate (including those entering and leaving the country) was 2.0 per 1000 population. Assume that these rates will remain constant in the future and that the population of the United States at the beginning of 1980 was 226,504,825. Also assume for the purpose of simulating this process on the computer that all births, deaths, and immigrations take place on the last day of each year. Write and run a program that will determine in which year the population of the U.S. will reach 300,000,000.

EXERCISE 7-8

A rocket is fired vertically into the air with an initial velocity of V feet per second. The height H of the rocket above the ground at any time T is given by

$$H = -16.1\ T^2 + VT$$

Write a program that will:

a. accept a value of V entered from the keyboard,

b. print the letters T and H for a table heading,

c. compute H for values of T starting at 0 and increasing at one-second intervals until the rocket hits the ground, and

d. print the values of T and H in the form of a table. Run the program using a value of V = 200 feet per second.

8 DISPLAYING THE FLAG: LEARNING ABOUT FOR...NEXT

In Chapter 7 you learned how to use the IF...THEN statement to form various looping structures in BASIC. There is another looping structure available in BASIC that is called a FOR...NEXT loop. This loop uses the BASIC statements FOR and NEXT and is particularly useful when you know how many times you want to go through a loop.

In this chapter you will learn:

1. how to form a FOR...NEXT loop
2. to draw lines and boxes using the FOR...NEXT loop
3. how to use nested FOR...NEXT loops
4. how to display the American flag on the Commodore 64/VIC 20 screen.

THE FOR...NEXT LOOP

The general form of a FOR...NEXT loop is shown in Figure 8.1. When statement 10 is executed, the value of I, called the index variable, is equated to M1 and statements 20, 30, and 40 are executed. If M3 > 0, then when statement 50 is executed the value of I will be incremented by M3, and if I is less than or equal to M2, statements 20, 30, and 40 will be executed again. This process continues until I becomes greater than M2, at which point the program branches to line 60. Every time around the loop I is incremented by M3. In line 10 the phrase STEP M3 is optional. If it is omitted, an increment of 1 is assumed.

If M3 < 0, then when statement 50 is executed, the value of I will be decremented by M3, and statements 20, 30, and 40 will continue to be executed until I becomes less than M2.

Figure 8.1 General form of the FOR...NEXT loop.

```
10 FOR I=M1 TO M2 STEP M3
20 ————————
30 ————————
40 ————————
50 NEXTI
60 ————————
```

The pseudocode and structured flowchart equivalent of the FOR...NEXT loop are shown in Figure 8.2 for the case M3>0. When writing pseudocode or drawing structured flowcharts, you can also use the representations of the FOR...NEXT loop shown in Figure 8.3. The FOR...NEXT loop is always executed at least once, even if M1 is greater than M2 and M3 is positive.

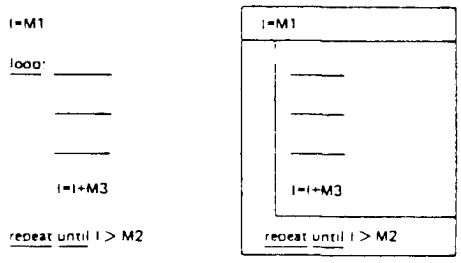


Figure 8.2 (a) Pseudocode and (b) structured flowchart equivalent of the FOR...NEXT loop.

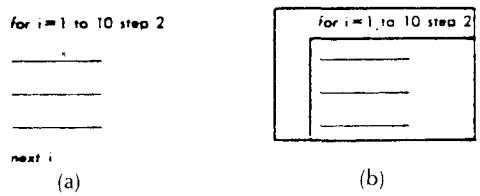


Figure 8.3 (a) Pseudocode and (b) structured flowchart representation of the FOR...NEXT loop.

Immediate Mode Execution of the FOR...NEXT Loop

If the entire FOR...NEXT loop can fit on four screen lines on the VIC 20 or two screen lines on the Commodore 64, you can execute the loop in the immediate mode. To see how the FOR...NEXT loop works, try typing in the following examples in the immediate mode.

FOR I=1 TO 10:? I;:NEXTI

FOR I=1 TO 10 STEP 2: ? I;: NEXTI

FOR J=10 TO 1 STEP −1: ? J;: NEXTJ

FOR I=1 TO 10: ?"SHIFT B CURSOR DOWN CURSOR LEFT";: NEXT

These examples are shown in Figure 8.4. Note from the fourth example that the index variable I on NEXTI is optional, and NEXTI can be written simply as NEXT. We will normally use only NEXT, except in nested FOR...NEXT loops, where it is helpful to know which NEXT goes with which loop.

Drawing Boxes

Figure 8.5 shows the listing of a program that draws a large box on the screen. Type in this program and run it. The result of running this program on both the VIC 20 and the Commodore 64 is shown in Figure 8.6.

Study this program and make sure that you understand how it works. Line 5 clears the screen. Line 10 draws the top of the box from left to right. Line 20

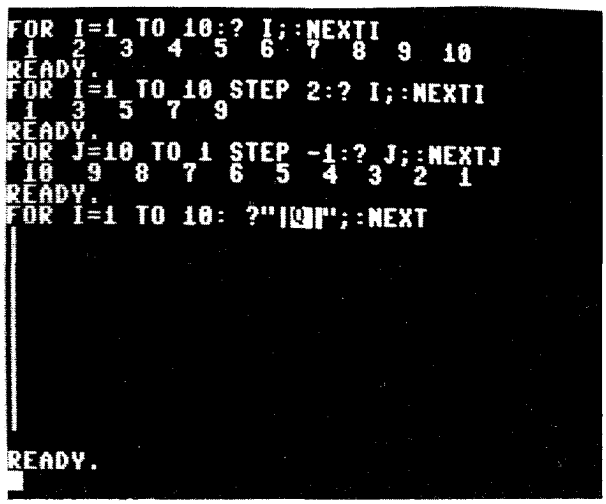


Figure 8.4 Examples of using the FOR...NEXT loop in the immediate mode.

Figure 8.5 Listing of program to draw a large box.

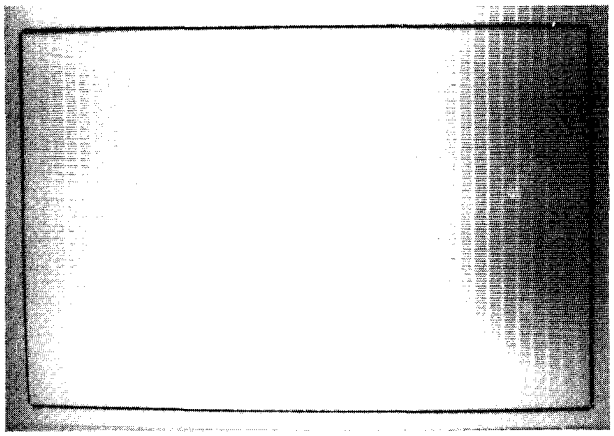
```
2 REM  PROGRAM TO DRAW A LARGE BOX
5 PRINT"■ ";
10 FOR I=1 TO 20:PRINT"_";:NEXT:PRINT"■";
20 FOR I=1 TO 20:PRINT"■ ■";:NEXT:PRINT"■";
30 FOR I=1 TO 20:PRINT"←■";:NEXT:PRINT"■";
40 FOR I=1 TO 20:PRINT"■ ■";:NEXT
50 FOR I=1 TO 2000:NEXT

READY.
```

draws the right side of the box from top to bottom. Line 30 draws the bottom of the box from right to left. Note that each time the PRINT statement in line 30 is executed, one line segment is printed, followed by two CURSOR LEFT movements. Line 40 draws the left side of the box from bottom to top. The statement **50 FOR I=1 TO 2000: NEXT** creates a time delay so you can view the box briefly before the "READY." message is displayed.

In each of the FOR statements in lines 10-40 the index I goes from 1 to 20, so 20 line segments are drawn on each side of the box. Therefore, the box should be square. But from Figure 8.6 you can see that the width is greater than the height on the VIC 20 and the height is greater than the width on the Commodore 64. (This result may vary somewhat depending upon the type of television or monitor being used for display.) In the case of the VIC 20, when the dots are displayed on the video screen, they are closer together along a vertical line than along a horizontal line. Thus, a vertical line segment consisting of eight dots will be somewhat shorter than a horizontal one. On the Commodore 64, the situation is reversed.

In order to correct for this inherent difference in scale between horizontal and vertical lines, measure the sides of whichever rectangle is appropriate in

(a)



(b)

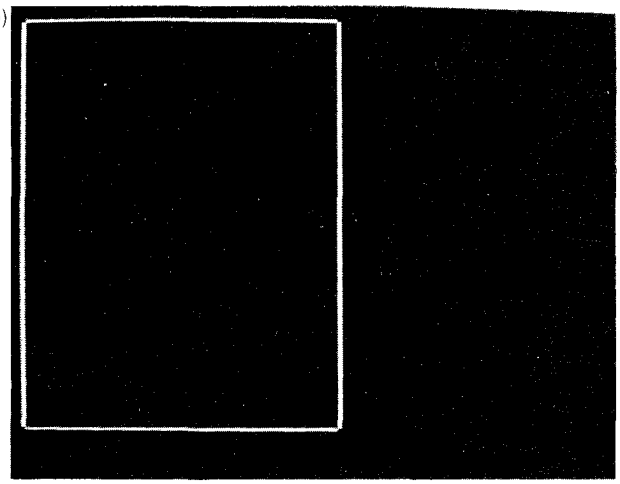


Figure 8.6 Result of running the program in Figure 8.5 on (a) the VIC 20 and (b) the Commodore 64.

Figure 8.6. We measured a width of 6 ½ inches and a height of 4 ¼ inches on the VIC 20 screen and a width of 4 ¾ inches and a height of 6 ¼ inches on the Commodore 64 screen. Thus, the ratio of width to height is

$$\frac{6\ \frac{1}{2}}{4\ \frac{1}{4}} = \frac{6.5}{4.25} = 1.53$$

on the VIC 20 and

$$\frac{4\ \frac{3}{4}}{6\ \frac{1}{4}} = \frac{4.75}{6.25} = .76$$

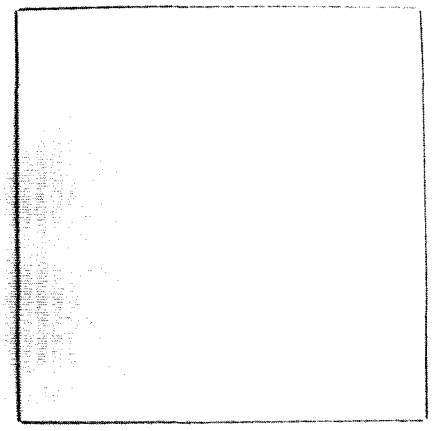
on the Commodore 64. Therefore, instead of making the horizontal lines 20 units long, we really should make them $20/1.53 = 13.07 \approx 13$ units long in the case of the VIC 20. In the case of the Commodore 64, we should make them $20/.76 = 26.31 \approx 26$ units long. When we change the values of 20 in lines 10 and 30 to 13 for the VIC 20 and 26 in the case of the Commodore 64, we get Figure 8.7, both parts of which are nearly squares.

If you look closely at the boxes in Figure 8.6 and Figure 8.7, you will notice that the corners do not meet. This is because as we draw the box, the cursor is always on the *outside* of the box. We could fill in the corners by using the graphic characters on the O,P,L, and @ keys. However, in this case the cursor must be *inside* the box when drawing the sides. This can be illustrated by using the BASIC statements shown in Figure 8.8a to draw two sides of the box.

Line 150 clears the screen. Line 160 moves the cursor to the HOME position and then prints the symbol on the O key to form the corner. The FOR...NEXT loop then prints seventeen line segments along the top. Line 170 moves the cursor back HOME and then down one. The FOR...NEXT loop then prints thirteen line segments along the left edge of the screen. Finally, the cursor is moved five more lines down the screen so that the READY message will appear at the bottom of the screen. (Lines 160 and 170 will be used later in this chapter to form part of the flag.)

Figure 8.7 Dividing the length of all horizontal lines in Figure 8.6 by a factor will produce a square: (a) VIC 20 with factor 1.53. (b) Commodore 64 with factor .76.

(a)



(b)

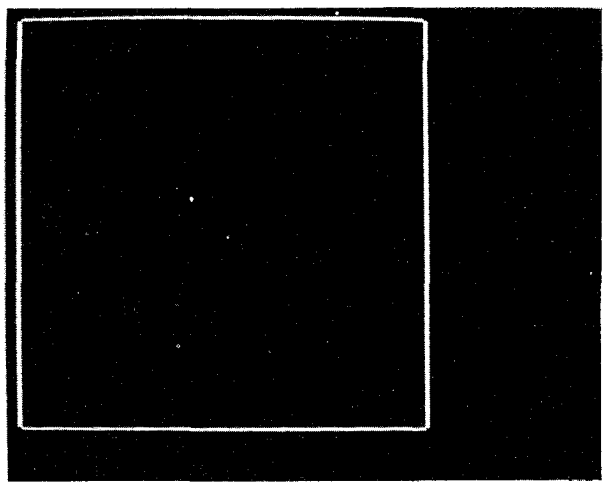
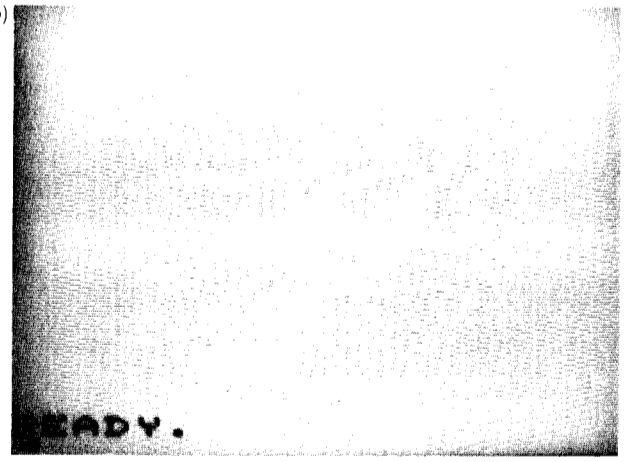


Figure 8.8 The program in (a) produces the two intersecting lines in (b).

(a)
```
150 PRINT "▒"
160 PRINT "▒▒";:FOR I=1 TO 17:PRINT "▒";:NEXT I
170 PRINT "▒▒";:FOR I=1 TO 13:PRINT "▒▒";:NEXT I:PRINT "▒▒▒▒▒"

READY.
```

(b)



NESTED FOR...NEXT LOOPS

FOR...NEXT loops may be *nested.* This means that we can put one FOR...NEXT loop *completely* within another one. When this is done, the inner FOR...NEXT loop is executed completely during *each* pass through the outer loop. This makes it easy to perform fairly complex operations.

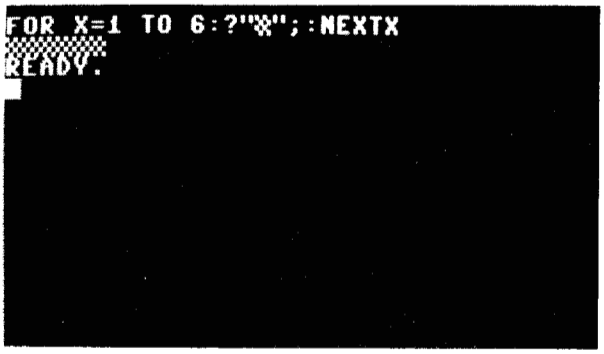
Plotting Areas

In order to see how nested FOR...NEXT loops work, type in the following FOR...NEXT loop in the immediate mode:

FOR X=1 TO 6: ?"LOGO+";: NEXTX

This statement will display a row of six graphic symbols, as shown in Figure 8.9.

Figure 8.9 Displaying one row of graphic symbols.

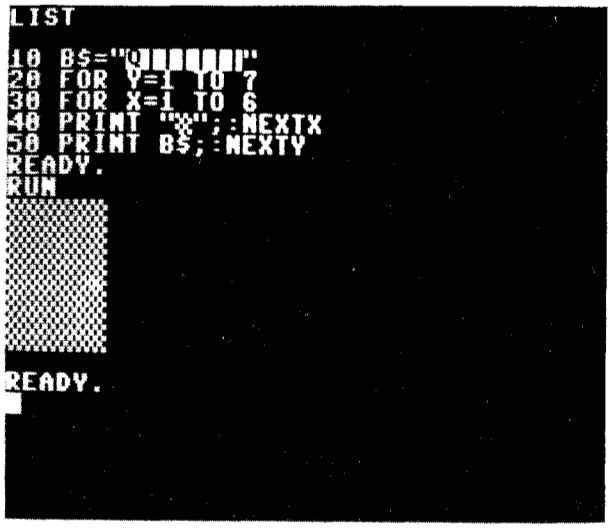


Now suppose you would like to display a rectangular area consisting of seven of the rows shown in Figure 8.9. You can do this with the following algorithm:

```
for y=1 to 7
  draw one row of length 6
  move cursor down one and back 6
next y
```

The BASIC program to do this is shown in Figure 8.10. Line 10 defines the string B$ which consists of one CURSOR DOWN and six CURSOR LEFT movements. Lines 20-50 define the outer loop, and lines 30-40 define the inner loop. This inner loop is the FOR...NEXT loop shown in Figure 8.9 that draws one row of length 6. The statement **PRINT B$** in line 50 will move the cursor down one and back six. Therefore, the next time through the outer Y FOR...NEXT loop, the row plotted by the X FOR...NEXT loop will be displayed directly below the previous row. After seven trips through the Y loop, an area seven rows high will have been plotted as shown in Figure 8.10. You should study this figure and make sure you understand how nested FOR...NEXT loops work.

Figure 8.10 Nested FOR...NEXT loops can be used to plot areas.



Plotting an Array of Points

Modify the program shown in Figure 8.10 by making the following changes:

1. add two blank spaces following the graphic symbol in the string in line 40
2. change the PRINT statement in line 50 to **PRINT B$;B$;B$;**

If you make these changes and run the modified program, you should get the result shown in Figure 8.11. Note that the inner loop still plots a single row consisting of six concatenated strings. However, now each string contains three characters (the graphic symbol plus two blanks). This means that each row is now eighteen characters long. Since B$ is plotted three times in line 50, the cursor is moved a total of three locations down and eighteen back.

Plotting the Star Field

When we display the flag later in this chapter we will need to plot the star field. We will use the small squares located on the left front of the C (V on the VIC 20) and F keys to represent the stars. These will be arranged according to the patterns shown in Figure 8.12.

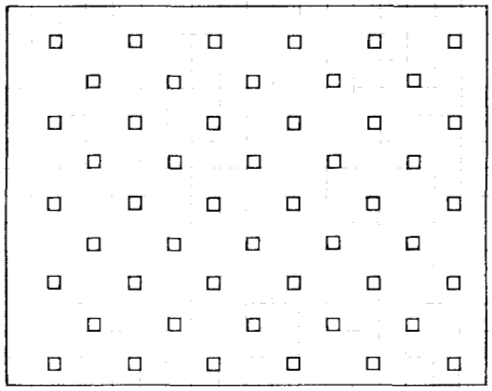
If you look carefully at these patterns, you will see that they consist of two rectangular arrays of points;

Figure 8.11 (a) Modified program to display an array of points. (b) Execution of program in (a).

(a)
```
10 B$="■■■■■■■■"
20 FOR Y=1 TO 7
30 FOR X=1 TO 6
40 PRINT "■  ";:NEXTX
50 PRINT B$;B$;B$;:NEXTY

READY.
```

(b)
```
RUN
■  ■  ■  ■  ■  ■

■  ■  ■  ■  ■  ■

■  ■  ■  ■  ■  ■

■  ■  ■  ■  ■  ■

■  ■  ■  ■  ■  ■

■  ■  ■  ■  ■  ■

READY.
```

(a)



(b)

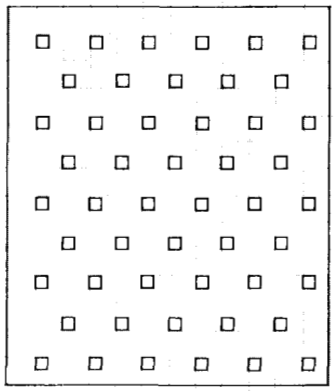


Figure 8.12 Pattern used to display the star field in the flag: (a) Commodore 64 and (b) VIC 20.

one is a 5 × 6 array using the graphic character LOGO C (LOGO V on the VIC 20) and the other is a 4 × 5 array using the graphic character LOGO F. These two rectangular arrays will be plotted separately.

The first rectangular array can be plotted using the following algorithm:

```
clear screen
home cursor
for y=1 to 5
   move cursor to beginning of next line
   for x=1 to 6
      print "space LOGO C space"; ("space LOGO V"
      on the VIC 20)
   next x
   move cursor to beginning of next line twice
next y
```

You should study this algorithm to see how it produces the 5 × 6 array of points shown in Figure 8.12. The BASIC program for implementing this algorithm on the Commodore 64, and its execution, are shown in

Figure 8.13. Note that the statement PRINT is used to "move cursor to beginning of next line."

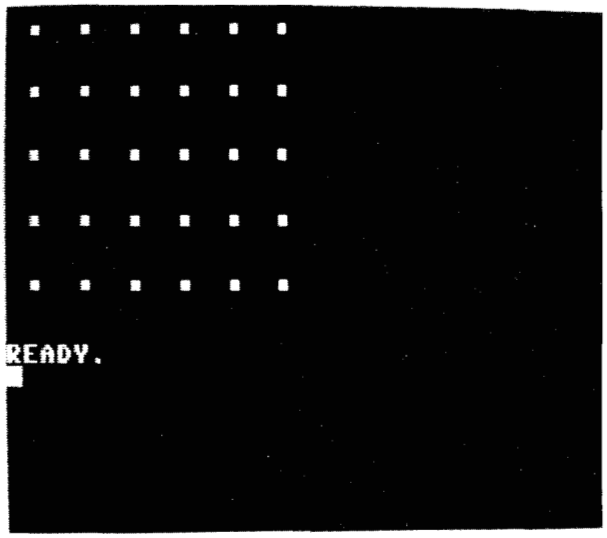
The second 4 × 5 rectangular array shown in Figure 8.12 needs to be interleaved with the first array of points. This can be accomplished by adding the following algorithm:

```
home cursor
move cursor down 2
for y=1 to 4
   print space
   for x=1 to 5
      print "space space LOGO F"; ("space LOGO F"
         on the VIC 20)
   next x
   move cursor to beginning of next line 3 times
next y
```

Figure 8.13 (a) Commodore 64 BASIC program to display first rectangular array of points for star field. (b) Execution of statements in (a).

(a)
```
90 PRINT "■"
100 PRINT "■";:FOR Y=1 TO 5:PRINT
110 FOR X=1 TO 6:PRINT "  ■ ";:NEXTX
120 PRINT:PRINT:NEXTY

READY.
```

(b)



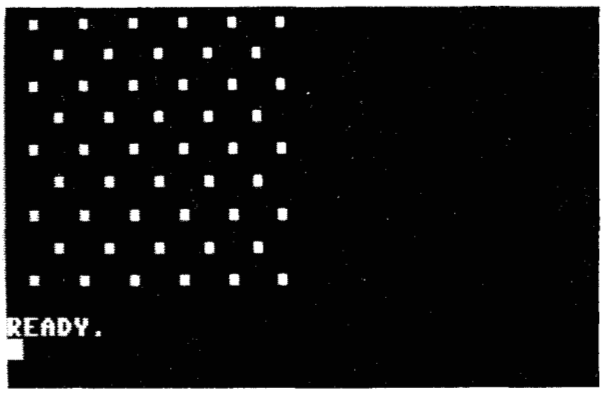
The BASIC statements for implementing this algorithm on the Commodore 64 are given in lines 130-150 in Figure 8.14a. Lines 90-120 are the same as the ones shown in Figure 8.13a. Therefore, when all of the statements in Figure 8.14a are executed, the array shown in Figure 8.13b is plotted first, followed by the interleaved 4 × 5 array, as shown in Figure 8.14b.

(Statements 100-150 will be used later in the chapter to plot the star field in the flag.)

Figure 8.14 (a) Commodore 64 BASIC program to display star field. (b) Star field displayed by program in (a).

(a)
```
90 PRINT"■"
100 PRINT "■";:FOR Y=1 TO 5:PRINT
110 FOR X=1 TO 6:PRINT "  ■ ";:NEXTX
120 PRINT:PRINT:NEXTY
130 PRINT"■■■";:FOR Y=1 TO 4:PRINT "  ";
140 FOR X=1 TO 5:PRINT"  ■";:NEXTX
150 PRINT:PRINT:PRINT:NEXTY

READY.
```

(b)



Making Stripes

The one additional thing we need to learn in order to display our flag is how to make stripes. In this section we will write a general program that can display any size striped pattern made from any graphic characters. The program will ask the user to enter the following values from the keyboard:

1. the number of stripes, N, to be plotted
2. the width of each stripe, W
3. the length of each stripe, L
4. the two characters, A$ and B$, from which the stripes will be formed.

Given these variables, Figure 8.15 shows an algorithm that will display N stripes, each of width W and length L, starting with the A$ pattern.

Figure 8.15 Algorithm for displaying N stripes, each of width W and length L, starting with the pattern A$.

```
clear screen
C$=A$
for y=1 to N
      for i=1 to W
            for x=1 to L
               print C$;
            next x
            move cursor to beginning of next line
      next i
      if C$=A$
      then C$=B$
      else C$=A$
next y
```

In this algorithm the innermost FOR...NEXT loop (for x) will plot one row of L characters C$. (The string C$ is initially equated to A$.) The middle FOR...NEXT loop (*for* i) will plot W rows containing the C$ character. This will result in a stripe of width W. After this FOR...NEXT loop is completed, the character in C$ is changed to the other stripe character, using the *if...then...else* statement. The outer FOR...NEXT loop (*for* y) will continue to plot stripes until N stripes have been plotted.

A listing of the BASIC program corresponding to this algorithm is shown in Figure 8.16. Type in this program and run it. A sample run of this program is shown in Figure 8.17. You should try making different kinds of stripes using this program. Another sample run of this program is shown in Figure 8.18. We will use the values shown in this example to help display our flag.

Figure 8.16 BASIC listing of program to make stripes.

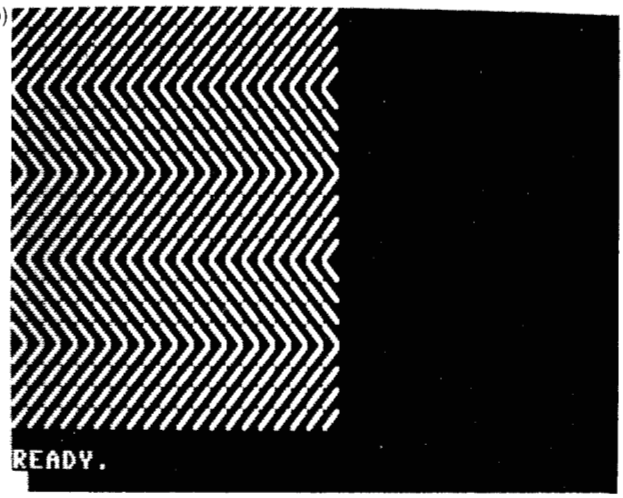
```
10 REM PROGRAM TO MAKE STRIPES
20 INPUT"ENTER NUMBER OF STRIPES";N
30 INPUT"ENTER WIDTH OF EACH STRIPE";W
40 INPUT"ENTER LENGTH OF EACH STRIPE";L
50 INPUT"ENTER 2 CHARACTERS";A$,B$
60 PRINT"□";:C$=A$
70 FOR Y=1 TO N
80 FOR I=1 TO W
90 FOR X=1 TO L:PRINT C$;:NEXTX
100 PRINT:NEXTI
110 IF C$=A$ THEN C$=B$:GOTO 130
120 C$=A$
130 NEXTY

READY.
```

Figure 8.17 Sample run of program shown in Figure 8.16.

(a)

```
RUN
ENTER NUMBER OF STRIPES? 5
ENTER WIDTH OF EACH STRIPE? 4
ENTER LENGTH OF EACH STRIPE? 20
ENTER 2 CHARACTERS? /,\
```

(b)



(a)

```
RUN
ENTER NUMBER OF STRIPES? 12
ENTER WIDTH OF EACH STRIPE? 2
ENTER LENGTH OF EACH STRIPE? 39
ENTER 2 CHARACTERS? "■","■"
```

(b)

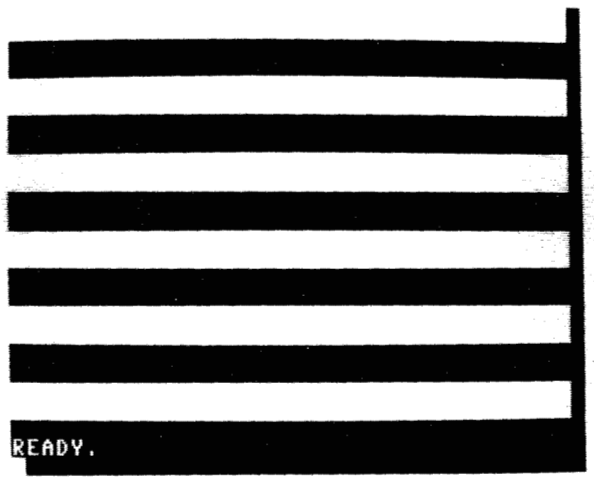


Figure 8.18 A second run of program shown in Figure 8.16. (For the VIC 20, use 11 stripes of lengths 21.)

Notice that in Figure 8.18 we asked for twelve stripes, (eleven is correct for the VIC) starting with the dark stripe. This is actually what was plotted, but the first stripe was scrolled off the top of the screen when the ready message was printed. One way to keep the READY message from being printed (and therefore from scrolling the pattern off the screen) is to end the program with a statement such as **185 GOTO 185.** This will cause line 185 to loop to itself forever (or until you press the STOP key). But the READY message will not be displayed because the program is still executing. We will use this trick when displaying our flag.

DISPLAYING THE FLAG

The American flag has thirteen stripes. If we use two screen lines for each stripe, we will require twenty-six lines. But the Commodore 64 has only twenty-five screen lines and the VIC 20 has only twenty-three. Therefore, we will have to compromise and plot only one screen line for the bottom red stripe of the flag in the case of the Commodore 64 and omit the bottom red stripe and half of the white stripe above it in the case of the VIC 20. The star field shown in Figure 8.12 is fourteen screen lines high, which corresponds to the upper seven stripes of the flag. This is the correct size of the star field.

The BASIC program shown in Figures 8.19a and 8.19b will display the stripes of the flag. Lines 20-60 of Figure 8.19a contain the algorithm shown in Figure 8.15, with N=12, W=2, and L=39. This will produce the first twelve stripes on the Commodore 64. Line 70 will display one screen line of the thirteenth stripe. Line 185 prevents the flag from being scrolled off the top of the screen. Figure 8.20a shows the stripes produced by the program in Figure 8.19a.

Lines 20-60 of Figure 8.19b contain the algorithm of Figure 8.15, with N=11, W=2, and L=21. This produces the first eleven stripes on the VIC 20. The twelveth stripe, which is white on the flag, occurs by default since the background color on the VIC 20 is white. However, this stripe is only one screen line instead of two. Figure 8.20b shows the stripes produced by the program in Figure 8.19b.

We now need to add the "blue" field to Figure 8.20. This will be done by blanking out a 14×18 (14×12 on the VIC 20) area in the upper left-hand corner of the screen. The following algorithm will do this:

```
home cursor
for y=0 to 13
   for x=0 to 17 (11 on the VIC 20)
      print" ";
   next x
   move cursor to beginning of next line
next y
```

This algorithm is accomplished by adding lines 80 and 90 to our programs as shown in Figure 8.21. The result of executing these new programs is shown in Figure 8.22.

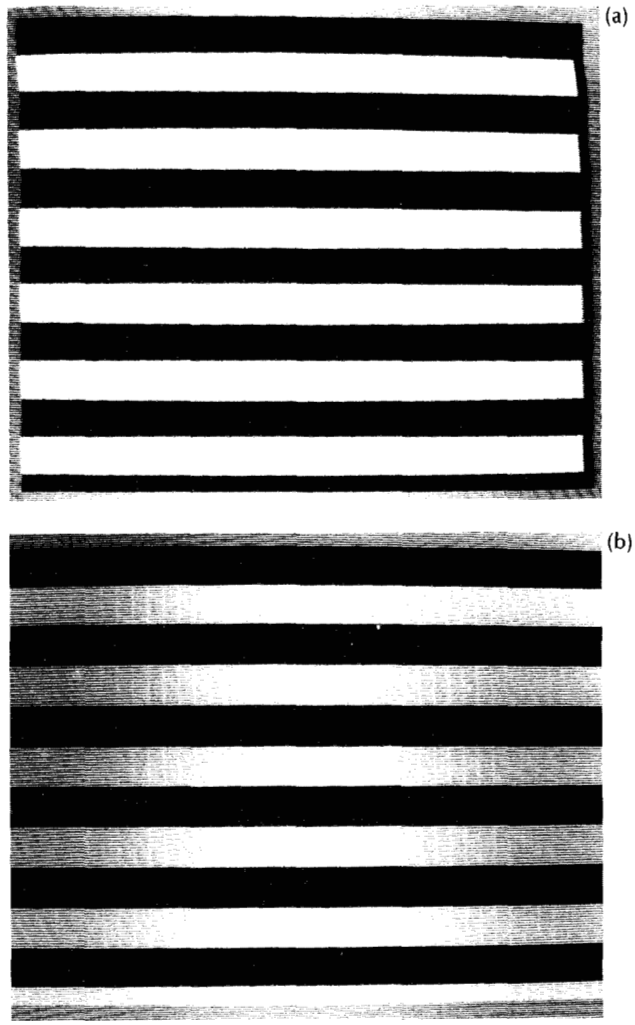


Figure 8.20 Result of executing program shown in Figure 8.19. (a) Commodore 64. (b) VIC 20.

Figure 8.19 Program to display the 13 stripes of the flag. (a) Commodore 64. (b) VIC 20.

(a)
```
10 REM  PROGRAM FOR DISPLAYING FLAG
20 PRINT"■";:A$="■":B$="■":C$=A$
30 FOR Y=1 TO 12: FOR N=1 TO 2:PRINT C$;
40 FOR X=1 TO 39: PRINT "■ ■";:NEXTX
45 PRINT:NEXTN
50 IF C$=A$ THEN C$=B$:GOTO 60
55 C$=A$
60 NEXTY
70 PRINT "■";:FOR N=1 TO 39:PRINT"■ ■";:NEXTN
185 GOTO 185

READY.
```

(b)
```
10 REM  PROGRAM FOR DISPLAYING FLAG
20 PRINT"■";:A$="■":B$="■":C$=A$
30 FOR Y=1 TO 11: FOR N=1 TO 2:PRINT C$;
40 FOR X=1 TO 21: PRINT "■ ■";:NEXTX
45 PRINT:NEXTN
50 IF C$=A$ THEN C$=B$:GOTO 60
55 C$=A$
60 NEXTY
185 GOTO 185

READY.
```

We now need to add the star field. This is done by adding lines 100-150 from Figure 8.14. The resulting programs and their execution are shown in Figures 8.23 and 8.24. Note we have narrowed the horizontal distance between stars in the case of the VIC 20 by omitting one space when we print each star in lines 110 and 140. There is a subtle difference between the programs for the Commodore 64 and VIC 20 in displaying the stars. Since the background color on the Commodore 64 is blue, we set the character color to white in line 100. Thus, when we print we get white characters on a blue background. However, in the case of the VIC 20, the background color is white. Therefore, in line 100, we leave the color blue and employ reverse video everytime we specify the character to print. The result is again white characters on a blue background.

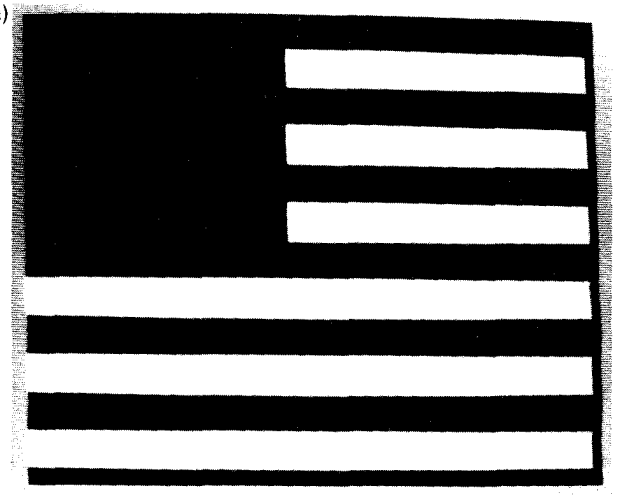
Figure 8.21 Programs to add the "blue" field to the flag. (a) Commodore 64. (b) VIC 20.

```
10 REM  PROGRAM FOR DISPLAYING FLAG
20 PRINT"□";:A$="■":B$="■":C$=A$
30 FOR Y=1 TO 12: FOR N=1 TO 2:PRINT C$;
40 FOR X=1 TO 39: PRINT "■ ■";:NEXTX
45 PRINT:NEXTN
50 IF C$=A$ THEN C$=B$:GOTO 60
55 C$=A$
60 NEXTY
70 PRINT "■";:FOR N=1 TO 39:PRINT"■ ■";:NEXTN
80 PRINT "■■";:FOR Y=0 TO 13
90 FOR X=0 TO 17:PRINT "■ ■";:NEXTX:PRINT:NEXTY
185 GOTO 185

READY.
```

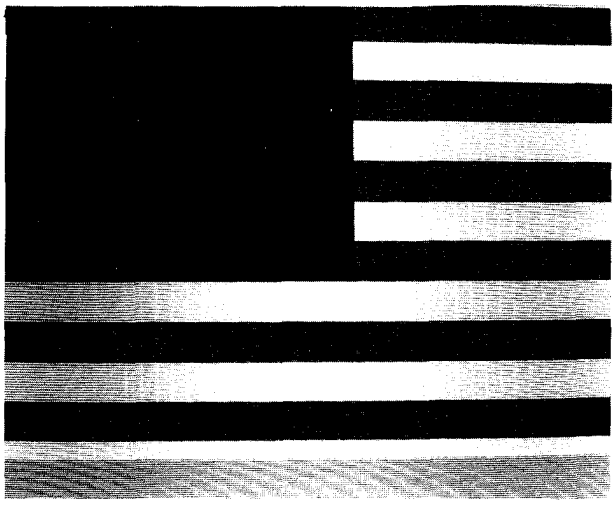
```
10 REM  PROGRAM FOR DISPLAYING FLAG
20 PRINT"□";:A$="■":B$="■":C$=A$
30 FOR Y=1 TO 11: FOR N=1 TO 2:PRINT C$;
40 FOR X=1 TO 21: PRINT "■ ■";:NEXTX
45 PRINT:NEXTN
50 IF C$=A$ THEN C$=B$:GOTO 60
55 C$=A$
60 NEXTY
80 PRINT "■■";:FOR Y=0 TO 13
90 FOR X=0 TO 11:PRINT "■ ■";:NEXTX:PRINT:NEXTY
185 GOTO 185

READY.
```

Figure 8.22 Result of executing program shown in Figure 8.21: (a) Commodore 64 and (b) VIC 20.

(a)



(b)



Finally we need to add a contrasting edge to the top and left-hand side of the "blue" field. This can be done by adding lines 160 and 170 from Figure 8.8. The final programs and flags are shown in Figures 8.25 and 8.26.

EXERCISE 8-1

Use FOR...NEXT loops to implement the checkerboard algorithm given in Figure 7.8. Run the program and display the brick wall shown in Figure 7.9.

EXERCISE 8-2

Write a program that will compute and print the cubes of the odd integers between 1 and 20.

EXERCISE 8-3

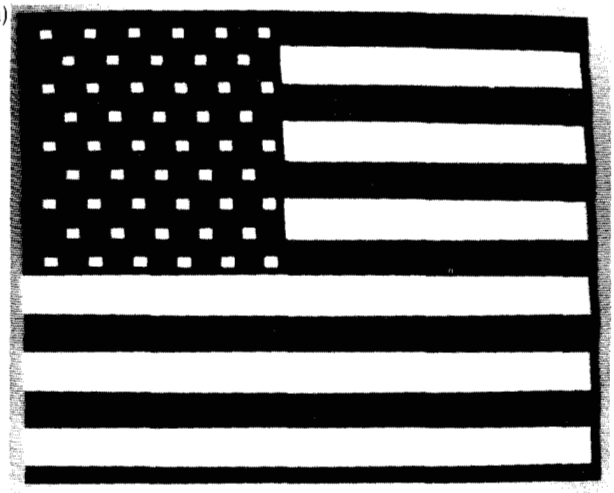
There are certain three-digit numbers, called "magic numbers." The sum of the cubes of the digits of a magic number equals the number itself.

Figure 8.23 Programs to add the star field to the flag.
(a) Commodore 64. (b) VIC 20.

```
10 REM  PROGRAM FOR DISPLAYING FLAG
20 PRINT"■";:A$="■":B$="■":C$=A$
30 FOR Y=1 TO 12: FOR N=1 TO 2:PRINT C$;
40 FOR X=1 TO 39: PRINT "■ ■";:NEXTX
45 PRINT:NEXTN
50 IF C$=A$ THEN C$=B$:GOTO 60
55 C$=A$
60 NEXTY
70 PRINT "■";:FOR N=1 TO 39:PRINT"■ ■";:NEXTN
80 PRINT "■■";:FOR Y=0 TO 13
90 FOR X=0 TO 17:PRINT "■ ■";:NEXTX:PRINT:NEXTY
100 PRINT "■■";:FOR Y=1 TO 5:PRINT
110 FOR X=1 TO 6:PRINT "  ■ ";:NEXTX
120 PRINT:PRINT:NEXTY
130 PRINT"■■■";:FOR Y=1 TO 4:PRINT " ";
140 FOR X=1 TO 5:PRINT"  ■";:NEXTX
150 PRINT:PRINT:PRINT:NEXTY
185 GOTO 185

READY.


20 PRINT"■";:A$="■":B$="■":C$=A$
30 FOR Y=1 TO 11: FOR N=1 TO 2:PRINT C$;
40 FOR X=1 TO 21: PRINT "■ ■";:NEXTX
45 PRINT:NEXTN
50 IF C$=A$ THEN C$=B$:GOTO 60
55 C$=A$
60 NEXTY
80 PRINT "■■";:FOR Y=0 TO 13
90 FOR X=0 TO 11:PRINT "■ ■";:NEXTX:PRINT:NEXTY
100 PRINT "■";:FOR Y=1 TO 5:PRINT
110 FOR X=1 TO 6:PRINT "■ ■";:NEXTX
120 PRINT:PRINT:NEXTY
130 PRINT"■■■";:FOR Y=1 TO 4:PRINT "■ ";
140 FOR X=1 TO 5:PRINT"■ ■";:NEXTX
150 PRINT:PRINT:PRINT:NEXTY
185 GOTO 185

READY.
```

Figure 8.24 Result of executing programs shown in Figure 8.23. (a) Commodore 64. (b) VIC 20.

(a)



(b)

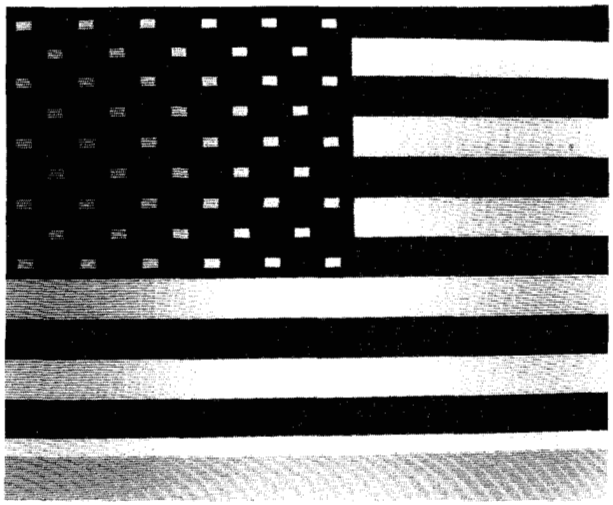
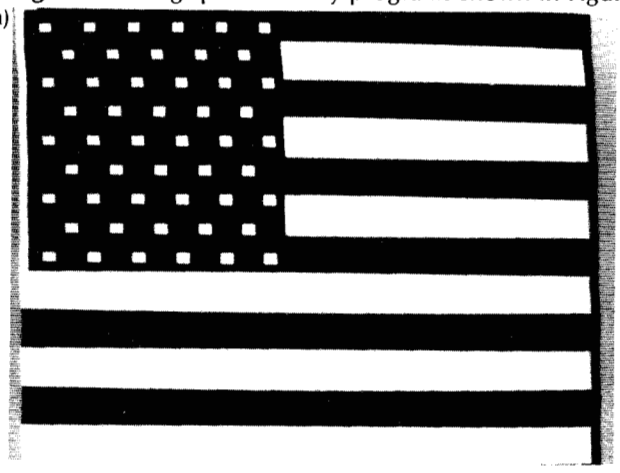


Figure 8.25 Final programs to display flag. (a) Commodore 64. (b) VIC 20.

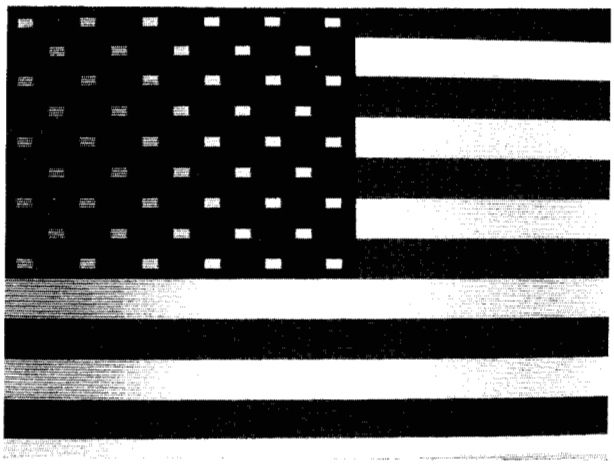
```
10 REM  PROGRAM FOR DISPLAYING FLAG
20 PRINT"■";:A$="■":B$="■":C$=A$
30 FOR Y=1 TO 12: FOR N=1 TO 2:PRINT C$;
40 FOR X=1 TO 39: PRINT "■ ";:NEXTX
45 PRINT:NEXTN
50 IF C$=A$ THEN C$=B$:GOTO 60
55 C$=A$
60 NEXTY
70 PRINT "■";:FOR N=1 TO 39:PRINT"■ ";:NEXTN
80 PRINT "■■";:FOR Y=0 TO 13
90 FOR X=0 TO 17:PRINT "■ ";:NEXTX:PRINT:NEXTY
100 PRINT "■■";:FOR Y=1 TO 5:PRINT
110 FOR X=1 TO 6:PRINT "  * ";:NEXTX
120 PRINT:PRINT:NEXTY
130 PRINT"■■■";:FOR Y=1 TO 4:PRINT "  ";
140 FOR X=1 TO 5:PRINT"  ■";:NEXTX
150 PRINT:PRINT:PRINT:NEXTY
160 PRINT"■■";:FOR I=1 TO 17:PRINT"_";:NEXTI
170 PRINT"■■";:FOR I=1 TO 13:PRINT"I ■";:NEXTI
180 PRINT"■■■■■■■■■";
185 GOTO 185

READY.

20 PRINT"■";:A$="■":B$="■":C$=A$
30 FOR Y=1 TO 11: FOR N=1 TO 2:PRINT C$;
40 FOR X=1 TO 21: PRINT "■ ";:NEXTX
45 PRINT:NEXTN
50 IF C$=A$ THEN C$=B$:GOTO 60
55 C$=A$
60 NEXTY
80 PRINT "■■";:FOR Y=0 TO 13
90 FOR X=0 TO 11:PRINT "■ ";:NEXTX:PRINT:NEXTY
100 PRINT "■";:FOR Y=1 TO 5:PRINT
110 FOR X=1 TO 6:PRINT "■ *";:NEXTX
120 PRINT:PRINT:NEXTY
130 PRINT"■■■";:FOR Y=1 TO 4:PRINT "■ ";
140 FOR X=1 TO 5:PRINT"■ ■";:NEXTX
150 PRINT:PRINT:PRINT:NEXTY
160 PRINT"■■■";:FOR I=1 TO 11:PRINT"■_";:NEXTI
170 PRINT"■■";:FOR I=1 TO 13:PRINT"■ ■";:NEXTI
180 PRINT"■■■■■■■■■■";
185 GOTO 185

READY.
```

Figure 8.26 Flags produced by programs shown in Figure 8.25. (a) Commodore 64. (b) VIC 20.

(a)



(b)



Write a program that will print the value of all such magic numbers in a column, in which each line is of the form **A MAGIC NUMBER IS --------.**

Use three nested FOR...NEXT loops with index variables I,J,K. The values of I,J, and K are taken as the digits of a number between 100 and 999. The number N is computed as **N=100*I+10*J+K.** N can then be compared to the sum of the cube of its digits to see if it is "magic."

EXERCISE 8-4

Write a program that will draw a large checkerboard (8×8), suitable for playing a game of checkers, on the screen.

EXERCISE 8-5

Write a program that will ask the user to supply the size of a square box and that will then draw the box. The program should be able to make the box as square as possible and test to make sure that it will fit on the screen.

EXERCISE 8-6

An area seven characters wide and nine characters high, made up of the left graphic character on the + key, can be used to represent a face-down playing card. Write a program that will display this figure on the screen.

EXERCISE 8-7

Write a program that will produce a "random" stripe pattern that fills the screen. The stripe pattern should be made up of two graphic characters that can be entered from the keyboard. Each horizontal line on the screen should have a 50/50 chance of being made from one of the two graphic characters. The width of each stripe in the pattern will then vary randomly.

9 SUBROUTINES: LEARNING TO USE GOSUB AND RETURN

Often you will have a sequence of BASIC statements that you would like to execute at several different locations within a program. Instead of having to repeat this sequence of statements every time you want to use it, you can write the statements only once as a *subroutine* and then *call* the subroutine each time you want to execute these statements.

Subroutines are also useful as a means of writing programs in a *modular* fashion. This becomes more and more important as the size of a program grows. Program segments that perform particular functions can be written as subroutines and then called when a given function needs to be performed. Because the Commodore 64/VIC 20 screen can display a maximum of only twenty-three/twenty-one program lines (leaving two lines for the READY message and the cursor), you should try to code your main program and your subroutines so that each fits within twenty-three/twenty-one screen lines. This will allow you to read and study a complete program segment without having to scroll the screen. This technique of *modularizing* your program will greatly simplify the process of debugging and modifying your program. It is the secret that allows you to write long programs with almost the same ease that you write short programs.

In this chapter you will learn how to:

1. use the GOSUB and RETURN statements
2. write a "Move to X,Y" subroutine that will move the cursor to a particular X,Y location on the screen
3. use the "Move to X,Y" subroutine to plot lines, arrays of points, and curves
4. display a three-dimensional letter on the screen
5. display your name in three dimensions on the screen.

THE GOSUB AND RETURN STATEMENTS

The general form of the GOSUB statement is **GOSUB** *line number*. When this statement is executed, the program branches to the statement at line "*line number.*" For example, the statement **GOSUB 500** will cause the program to branch to line 500 which contains the first statement of the subroutine. It looks as if **GOSUB 500** behaves the same way as **GOTO 500;** however, there is an important difference. The Commodore 64/VIC 20 will "remember" where the statement **GOSUB 500** is located in the program. You must include the statement **RETURN** at the end of the subroutine. When the RETURN statement is executed, the program will branch back to the next statement following **GOSUB 500.** This process is shown in Figure 9.1.

Now it looks as though you would accomplish the result in Figure 9.1 by using the two statements **60**

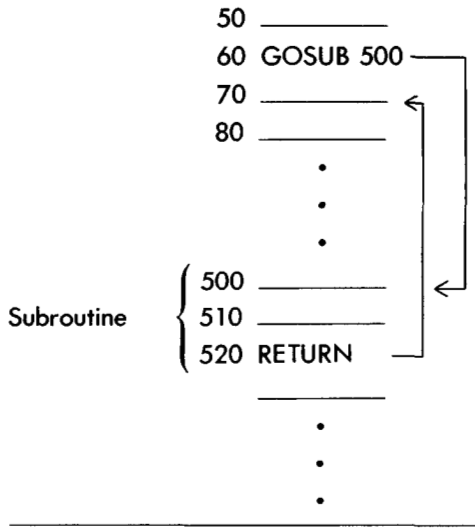
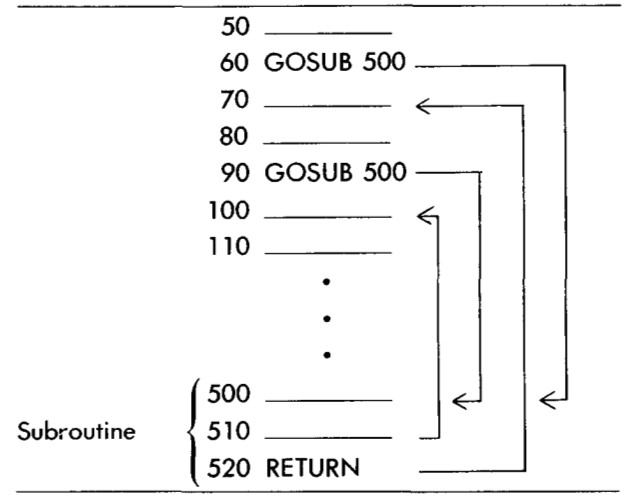


Figure 9.1 Forming a subroutine using GOSUB and RETURN.

GOTO 500 and **520 GOTO 70.** Although this would be true in Figure 9.1, it would not work if you wanted to call the same subroutine from two *different* locations in the program as shown in Figure 9.2. In this case the statement **60 GOSUB 500** will branch to the subroutine at line 500 then RETURN to line 70. However, the statement **90 GOSUB 500** will also branch to the subroutine at line 500 but will then RETURN to line 100. The Commodore 64/VIC 20 always remembers the point from which it branches to a subroutine, and it will always return to that point.

Figure 9.2 Calling a subroutine from two different locations within a program.



You can even call a subroutine from within another subroutine. The Commodore 64/VIC 20 will always find its way back by retracing its steps, as shown in Figure 9.3. Line 60 branches to the subroutine at line 500. Line 510, which is within this subroutine, branches to a second subroutine at line 600. The RETURN statement on line 620 will branch back to line 520, the statement following the GOSUB 600 statement. This happens to be the RETURN statement of the subroutine that begins at line 500. It will then branch back to line 70, the statement following the GOSUB 500 statement.

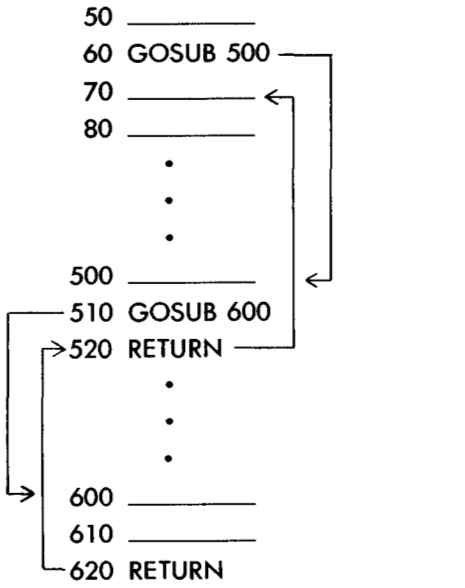
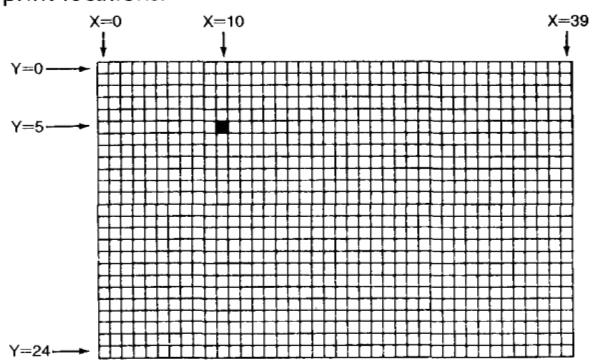


Figure 9.3 One subroutine can call another subroutine.

The "Move to X,Y" Subroutine

The Commodore 64 screen contains 1000 print locations. There are twenty-five rows, each containing forty column positions. We will number the rows 0-24 from top to bottom and call these the Y coordinates. We will number the columns 0-39 from left to right and call these the X coordinates. This labeling of the Commodore 64 screen is shown in Figure 9.4. The shaded spot in Figure 9.4 is located at the position

Figure 9.4 The Commodore 64 screen contains 1000 print locations.



X=10, Y=5. Any point on the screen can be identified by specifying its X and Y coordinates.

On the VIC 20, there are only 506 print locations. There are twenty-three rows, each containing twenty-two column positions. Thus, on the VIC 20, the Y coordinates or rows are numbered 0-22 from top to bottom. The X coordinates or columns are numbered 0-21 from left to right. The VIC 20 screen looks just like Figure 9.4 except the maximum X coordinate is X=21 and the maximum Y coordinate is Y=22.

It would be useful to have a subroutine that would move the cursor to any position on the screen specified by the values of X and Y. We could then print anything we want at this location.

If the cursor is at the beginning of a line, we can move it to position X on that line with the statement **PRINT TAB(X).** We can move the cursor to the beginning of line Y by moving the cursor to "home" and then spacing down by executing Y PRINT statements. (If Y=0, then we will not execute any PRINT statements.) The algorithm for moving the cursor to location X,Y on the screen can be written in pseudocode as shown in Figure 9.5. The algorithm can be written in BASIC as shown in Figure 9.6.

Figure 9.5 Algorithm for a subroutine to move cursor to position X,Y.

```
home cursor
if Y=O
then tab(X)
else for i=1 to Y
        move cursor to beginning of next line
     next i
     tab(X)
```

Figure 9.6 BASIC subroutine to move cursor to position X,Y.

```
500 REM MOVE TO X,Y ON SCREEN
510 PRINT "home";: IF Y=0 THEN 530
520 FOR I=1 TO Y: PRINT: NEXT
530 PRINT TAB(X);: RETURN
```

As an example of using this subroutine, the BASIC program shown in Figure 9.7 will do the following:

clear the screen

set X=10 and Y=5

call the "move to X,Y" subroutine

print a circular spot

Note that in this program the statement **100 END** is required to stop the program. Otherwise the program would continue, and the subroutine at line 500 would be executed again. The execution of the program in Figure 9.7 is shown in Figure 9.8.

Figure 9.7 BASIC listing of program to print a spot at location X=10, Y=5.

```
5 PRINT "□"
10 X=10
20 Y=5
25 GOSUB 500:PRINT "●"
100 END
500 REM  MOVE TO X,Y ON SCREEN
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

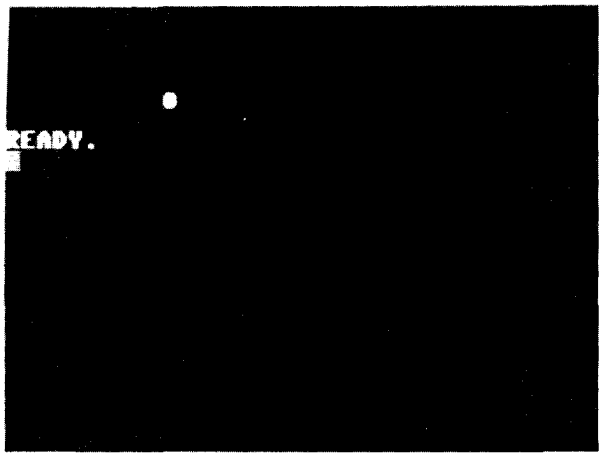


Figure 9.8 Result of executing the program shown in Figure 9.7.

EXAMPLES OF PLOTTING

The program shown in Figure 9.7 can easily be modified to plot lines, arrays of points, and curves.

Plotting Lines

A vertical line of dots starting at location X=10, Y=5 and going to location X=10, Y=20 can be plotted by changing line 20 in Figure 9.7 to **20 FOR Y=5 TO 20** and adding the statement **30 NEXTY.** The resulting program and its execution are shown in Figure 9.9.

If you now change line 20 to **20 FOR Y=5 TO 20 STEP 3** you will obtain the results shown in Figure 9.10. Note that in this case dots are plotted at the Y locations 5, 8, 11, 14, 17, and 20. Try changing the value of X and the range and step size of Y in Figure 9.10a.

Plotting Arrays of Points

By including X in a FOR loop in Figure 9.10, we can plot an array of points. Change line 10 to **10 FOR X=0**

Figure 9.9 (a) BASIC program to plot a vertical line of dots. (b) Execution of program in (a).

(a)
```
5 PRINT "□"
10 X=10
20 FOR Y=5 TO 20
25 GOSUB 500:PRINT "●"
30 NEXTY
100 END
500 REM  MOVE TO X,Y ON SCREEN
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

(b)

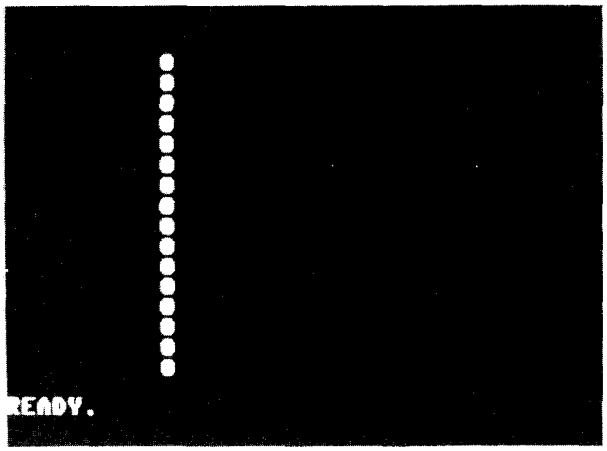
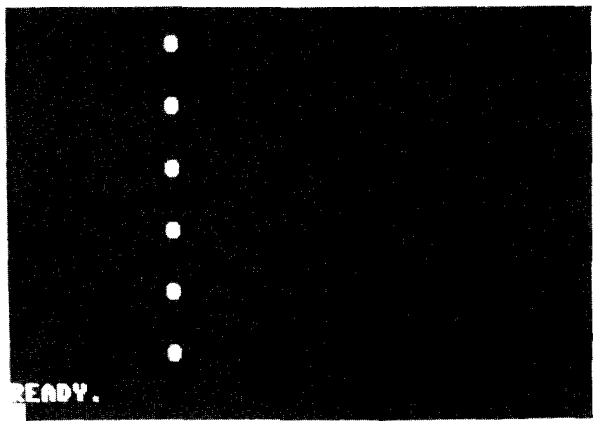


Figure 9.10 Result of adding STEP 3 to line 20 in Figure 9.9.

```
5 PRINT "□"
10 X=10
20 FOR Y=5 TO 20 STEP 3
25 GOSUB 500:PRINT "●"
30 NEXTY
100 END
500 REM  MOVE TO X,Y ON SCREEN
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```



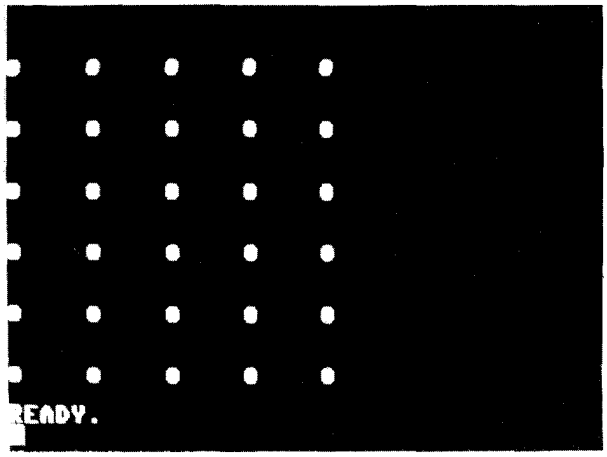
TO 20 STEP 5 and add the statement **40 NEXTX.** We now have a pair of nested FOR...NEXT loops. Therefore, the vertical line made up of the six dots shown in Figure 9.10 should occur at each of the following values of X: 0, 5, 10, 15, and 20.

The resulting listing and execution of this program are shown in Figure 9.11. You should modify this program by changing the ranges and step sizes of X and Y and by changing the graphic character that is plotted. There are *lots* of different possibilities. Try them.

Figure 9.11 Result of plotting X in a FOR...NEXT loop in Figure 9.10.

(a)
```
5 PRINT "□"
10 FOR X=0 TO 20 STEP 5
20 FOR Y=5 TO 20 STEP 3
25 GOSUB 500:PRINT "●"
30 NEXTY
40 NEXTX
100 END
500 REM  MOVE TO X,Y ON SCREEN
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

(b)



Plotting Curves

In Figure 9.11 the values of both X and Y are changed by FOR statements. However, if we know how the value of Y is related to the value of X, we can let the Commodore 64/VIC 20 calculate Y. For example, the curve Y = X should be a straight line along which each value of Y is equal to the value of X.

In order to plot this curve make the following changes in the program shown in Figure 9.11:

change line 10 to **10 FOR X=0 TO 19**
change line 20 to **20 Y=X**
delete line 30.

The resulting program and its execution are shown in Figure 9.12.

If you can write an equation for Y involving the value of X, then the program in Figure 9.12a can be used to plot this curve by simply changing the definition of Y on line 20. You may also want to change the range of X on line 10. For example, if you change line 20 to **20 Y=10−9*SIN(2*π*X/14)** you will obtain the curve shown in Figure 9.13. Try changing the word **SIN** in line 20 to **COS.** Also try changing the values of 9 (to something smaller) and 14 in line 20. On the Commodore 64, increase 19 in statement 10 to 39.

Figure 9.12 (a) Program to plot the curve Y=X. (b) Execution of program in (a).

(a)
```
5 PRINT "□"
10 FOR X=0 TO 19
20 Y=X
25 GOSUB 500:PRINT "■"
40 NEXTX
100 END
500 REM  MOVE TO X,Y ON SCREEN
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

(b)

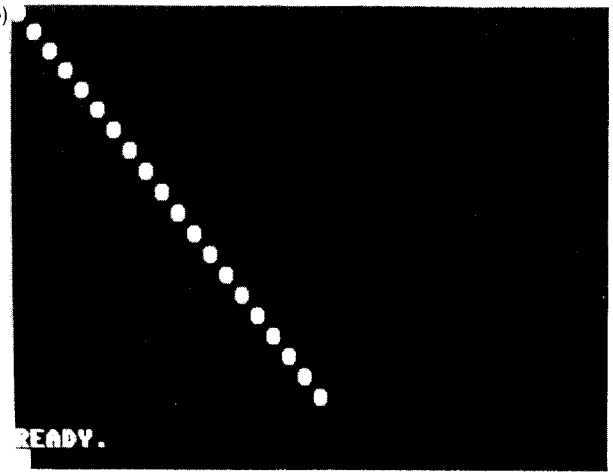
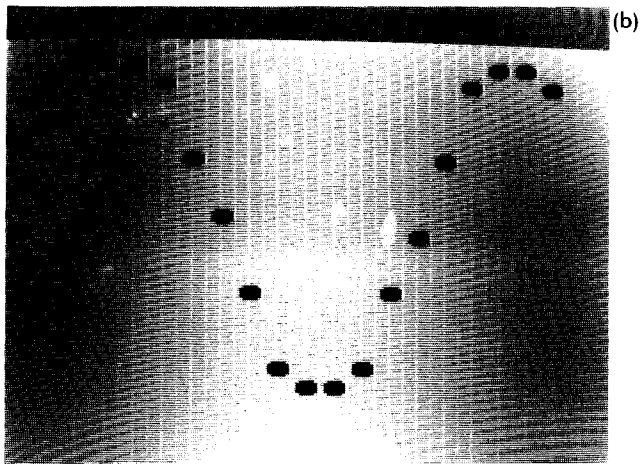


Figure 9.13 (a) Listing of program to plot the curve Y=10−9*SIN(2*π*X/14). (b) Execution of program shown in (a).

(a)
```
5 PRINT "□"
10 FOR X=0 TO 19
20 Y=10-9*SIN(2*π*X/14)
25 GOSUB 500:PRINT "■"
40 NEXTX
100 END
500 REM  MOVE TO X,Y ON SCREEN
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

(b)



EXERCISE 9-1

Run the program shown in Figure 9.13 four times, changing line 20 to each of the following statements:

a. 20 Y=10−4*LOG((X+1)/4)

b. 20 Y=20−20*EXP(−X/10)

c. 20 Y=X↑2/70

d. 20 Y=3*SQR(X)

THREE-DIMENSIONAL NAMES

As another example of using subroutines we will write a program that will display the name JEFF in three-dimensional block letters. We will begin by writing three subroutines: one for displaying a J, one for displaying an F, and one for displaying an E. In fact, the subroutine that displays an E will first call the subroutine that displays an F and then add the bottom horizontal piece to form an E. We will then write a main program that displays the name JEFF.

How to Make a 3-D Letter

The front face of each 3-D letter that we form will be ten rows high and four columns wide. The first step in making a 3-D letter is to sketch the block letter on a sheet of quadrille paper as shown in Figure 9.14.

The front face of the letter is printed with the graphic symbol formed by LOGO+. The edges of the letter are formed by a combination of reverse video space, SHIFT N, and SHIFT £.

To make it easier, we will construct a series of single row *primitives* that can then be combined in various ways to form different letters. The defining statements for these primitives are shown in lines 20-70 of Figure 9.15. These statements must be included at the beginning of a program that will display 3-D letters. In Figure 9.15 line 90 will branch to line 1000, where the

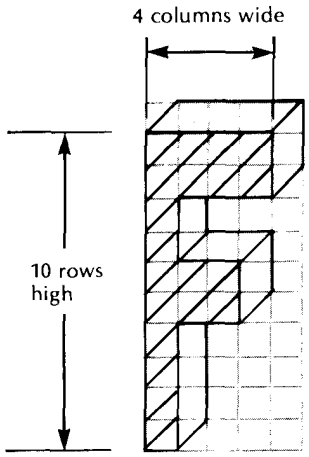


Figure 9.14 Begin by making a sketch of the letter.

main program that displays the name will be located. The subroutines for the various letters will be located between line 100 and line 1000.

The strings B2$, B4$, and B5$ defined in line 20 are cursor movement strings that move the cursor down one and left two, four, and five positions, respectively.

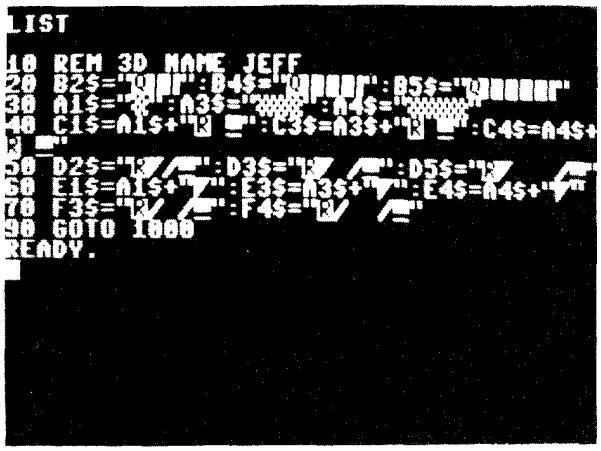


Figure 9.15 Primitives used to draw 3-D letters.

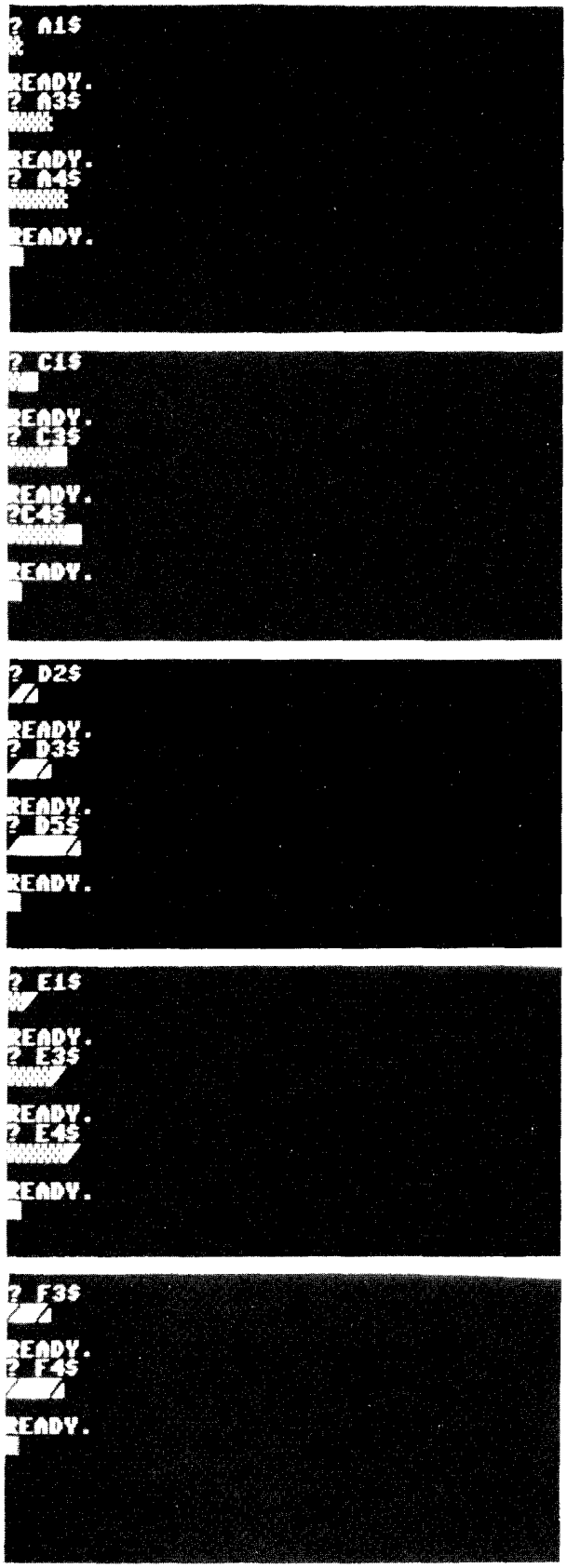
The strings A1$, A3$, and A4$ defined in line 30 produce rows of the LOGO+ graphic symbol of width 1, 3, and 4 respectively as shown in Figure 9.16a.

The strings C1$, C3$, and C4$ defined in line 40 are combinations of A1$, A3$, and A4$, followed by a reverse video space as shown in Figure 9.16b.

The strings D2$, D3$, and D5$ defined in line 50 can be used to draw the top parts of letters with total widths of 2, 3, and 5, respectively. These four strings are displayed in Figure 9.16c.

The strings E1$, E3$, and E4$ defined in line 60 are combinations of A1$, A3$, and A4$, followed by the SHIFT £ graphic symbol. These strings are displayed in Figure 9.16d. They are used to form the bottom edge of a segment. For example, E4$ is used to form the

Figure 9.16 The primitive strings used to form 3-D letters.



bottom of the top bar on the F in Figure 9.14; E3$ is used to form the bottom of the short bar of the F, and E1$ is used to form the very bottom of the F.

The strings F3$ and F4$ defined in line 70 have a total width of three and four spaces, respectively. They begin and end with a reverse video SHIFT N character and contain reverse video spaces in between. They are displayed in Figure 9.16e.

To form a 3-D letter, you simply combine the strings shown in Figure 9.16 with the cursor movement strings B2$, B4$, and B5$. We will illustrate this procedure by forming the letters J, F, and E.

Making a J

To make a J, begin by sketching the J on quadrille paper as shown in Figure 9.17. Now start at the top row and compare each line of the sketch with the row primitives shown in Figure 9.16. For example, the top row consists of three spaces plus the string D2$. Always start printing the letter from the upper left-hand corner of the 11-row × 5-column rectangle. You can then move this starting position to any location you want on the screen (by using the "Move to X,Y" subroutine).

After printing the top row, the cursor will be just to the right end of this row (follow all strings with a semicolon). Therefore, to display the long vertical post of the J, we need to print the strings B2$; (move cursor down 1 and left 2) and C1$; eight times. This can be accomplished with the statement **FOR I=1 TO 8:PRINT B2$;C1$;:NEXT.** The complete subroutine for displaying the 3-D J is shown in Figure 9.18. Line 110 draws the long vertical post. The cursor will then be just to the right end of row 9. Line 120 moves the cursor to the beginning of row 10 (B5$) and then up three rows. The strings D2$;B2$;C1$;B2$;A1$ then display the short post of the J. There is no primitive in Figure 9.16 that will draw the top edge of the bottom of the J. Therefore, we must include a string for this, made up of a reverse video SHIFT N followed by one reverse video space. In order to move the cursor down 1 and left 3 we use B4$ followed by a space (there is no B3$ primitive in Figure 9.16). Row 10 is just a C4$ primitive, and row 11 is an E4$ primitive. Line 130 is the RETURN statement that must be used to end all subroutines.

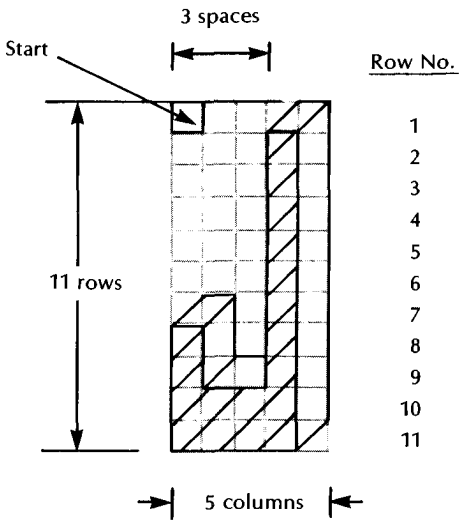


Figure 9.17 Sketch of 3-D J.

To make sure that this subroutine draws the J properly, add the statements shown in Figure 9.18 to the statements in Figure 9.15. Also add the statement **1000 END.** Now when you RUN this program it will look as though nothing happened. However, all of the strings in lines 20-70 will have been defined. You can now execute the statement GOSUB 100 in the *immediate* mode, and you should get the result shown in Figure 9.19.

Figure 9.19 Testing the subroutine shown in Figure 9.18.

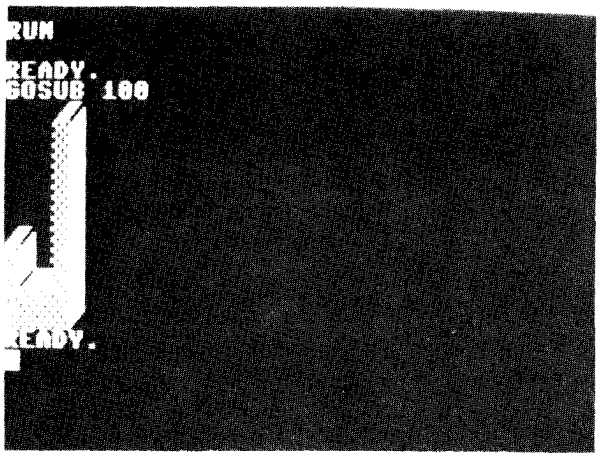


Figure 9.18 Subroutine to display 3-D J.

```
100 REM  3-D J
110 PRINT SPC(3);D2$;:FOR I=1 TO 8:PRINT B2$;C1$;:NEXT
120 PRINT B5$;"TTT";D2$;B2$;C1$;B2$;A1$;"■/ ■";B4$;" ";C4$;B5$;E4$;
130 RETURN

READY.
```

Making an F

The sketch of a 3-D F is shown in Figure 9.20a. By comparing this sketch with the row primitives shown in Figure 9.16 you can generate the table shown in Figure 9.20b. This table is then included in the PRINT statements shown in Figure 9.21, where the cursor movement strings B5$, B4$, and B2$ have been inserted in the appropriate places.

Add the subroutine shown in Figure 9.21 to the program, and then verify that it will draw an F as shown in Figure 9.22.

Making an E

The sketch of a 3-D E is shown in Figure 9.23. Since the E is just an F with a bottom bar added to it, you can make an E by first making an F (say **GOSUB 200**) and then adding a bottom bar. The subroutine to do this is shown in Figure 9.24.

After displaying the F using **GOSUB 200**, the cursor will be just to the right of the last row in the F. By moving the cursor up two lines and left one position, the bottom bar of the E can be drawn using the strings F4$, C3$, and E3$ as shown in Figure 9.24.

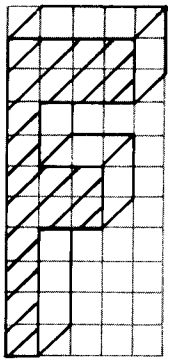
Now add the subroutine shown in Figure 9.24 to the program, and then verify that it will draw an E as shown in Figure 9.25.

Displaying the Name JEFF

Now that we have subroutines that will display the letters J, F, and E, we can combine them to form the name JEFF. We will need to move the cursor to a particular location on the screen and then call the appropriate letter subroutine. We will therefore need

Figure 9.20 (a) Sketch of a 3-D F. (b) Primitives used to form the 3-D F.

(a)



Row No.
1
2
3
4
5
6
7
8
9
10
11

(b)

Row No.	String
1	D5$
2	C4$
3	E4$
4	C1$
5	A1$;F3$
6	C3$
7	E3$
8	C1$
9	C1$
10	C1$
11	E1$

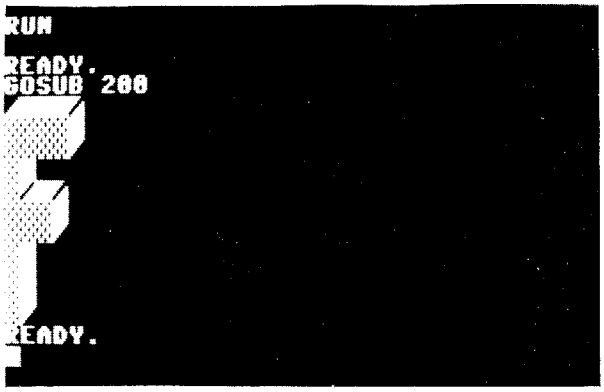


Figure 9.22 Testing the subroutine shown in Figure 9.21.

Figure 9.23 A 3-D E is a 3-D F plus a bottom bar.

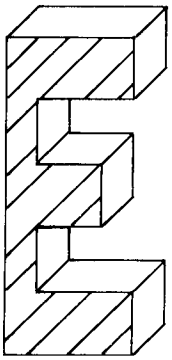


Figure 9.21 Subroutine to display 3-D F.

```
200 REM  3-D F
210 PRINT D5$;B5$;C4$;B5$;E4$;B5$;C1$;B2$;A1$;F3$;B4$;C3$;B4$;E3$;B4$;
220 FOR I=1 TO 3:PRINT C1$;B2$;:NEXT:PRINT E1$;
230 RETURN

READY.
```

Figure 9.24 Subroutine to display 3-D E.

```
300 REM  3-D E
310 GOSUB 200:PRINT"■■■";F4$;B4$;C3$;B4$;E3$;
320 RETURN

READY.
```

our "Move to X,Y" subroutine shown in Figure 9.6. A listing of this subroutine is shown in Figure 9.26. Add this subroutine to the program.

Line 1000 now contains the END statement. We must change this so that it is the beginning of the main program that will display the name JEFF. (Remember that line 90 in Figure 9.15 contains the statement **GOTO 1000**.) We will display the four letters J, E, F, and F with their upper left-hand corners at the following screen positions:

Figure 9.25 Testing the subroutine shown in Figure 9.24.

J X=1 Y=8 (X=6 for Commodore 64)
E X=6 Y=8 (X=14 " " ")
F X=11 Y=8 (X=22 " " ")
F X=16 Y=8 (X=30 " " ")

The main program to display these four letters is shown in Figure 9.27. Line 1005 clears the screen. Line 1010 displays J at X=1, Y=8. Line 1020 displays E at X=6, Y=8. Note that since the value of Y has not changed we need not specify its value again. Line 1030 displays F at X=11, Y=8, and line 1040 displays another F at X=16, Y=8. Line 1050 will prevent the READY message from being displayed by looping on itself endlessly until the STOP key is pressed.

Add the main program shown in Figure 9.27 to the rest of the program, and type **RUN.** The result should be as shown in Figure 9.28.

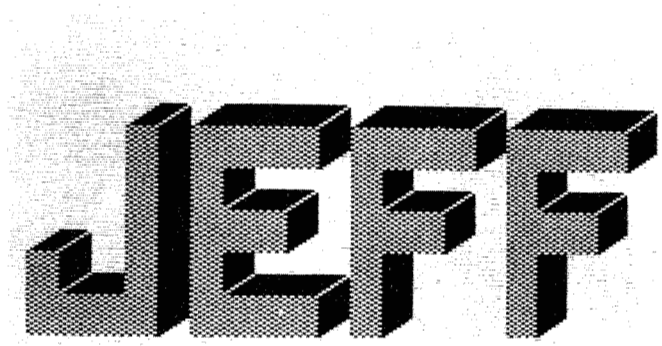
Figure 9.26 Subroutine "Move to X,Y".

```
500 REM MOVE TO X,Y ON SCREEN
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

Figure 9.27 Main program that displays the 3-D name JEFF.

```
1000 REM  MAIN PROGRAMTO DRAW JEFF
1005 PRINT"■"
1010 X=1:Y=8:GOSUB 500:GOSUB 100
1020 X=6:GOSUB 500:GOSUB 300
1030 X=11:GOSUB 500:GOSUB 200
1040 X=16:GOSUB 500:GOSUB 200
1050 GOTO 1050

READY.
```

(a)



(b)

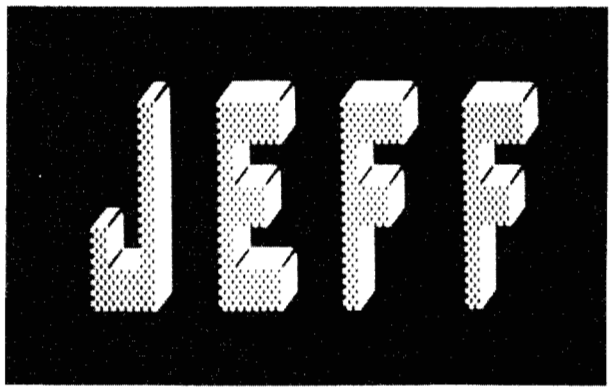


Figure 9.28 Result of executing program listed in Figures 9.25, 9.18, 9.21, 9.24, 9.26, and 9.27: (a) VIC 20 (b) Commodore 64.

EXERCISE 9-2

Modify the name program to display the word PET as shown in Figure 9.29. You will need to add subroutines to display a P and a T. Hint: A P can easily be made from an F. (PET was a Commodore computer preceding the VIC 20 and Commodore 64.)

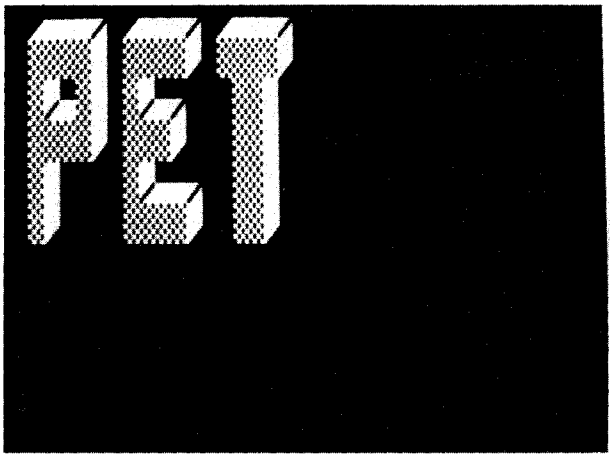


Figure 9.29 Display this word in Exercise 9-2.

EXERCISE 9-3

Modify the name program so that it will display your name, nickname, or another word containing four or fewer letters.

EXERCISE 9-4

A baseball player hits a ball with an initial velocity V (in feet per second) at an angle of A degrees to the ground, as shown in Figure 9.30. The height H (in feet) of the ball above the ground is given in terms of the distance X (in feet) from home plate by the equation

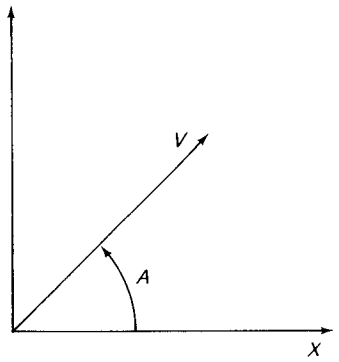


Figure 9.30 Diagram for Exercise 9-4.

$$H = \frac{-16.1\ X^2}{V^2 \cos^2 A} + X \tan A$$

Write a program that will (1) accept the values of V and A typed from the keyboard (2) compute H for one-foot increments in X until the ball hits the ground, and (3) plot the trajectory of the ball on the screen. Run the program for values of V = 40 feet per second and A = 45 degrees.

Caution: The functions COS(A) and TAN(A) must have A expressed in radians. Remember that $180° = \pi$ radians.

10 MAKING BAR GRAPHS: LEARNING ABOUT READ...DATA

You know two ways to assign a value to a memory cell name. One is to use an assignment statement such as **A=3;** the other is to use an INPUT statement such as **INPUT A.** In the second case the value is entered through the keyboard.

In this chapter you will learn another method of assigning values to memory cell names. The values to be assigned are stored *in the program* in DATA statements. They are assigned to memory cell names by using a READ statement. Another method of entering data through the keyboard will be described in Chapter 11.

In this chapter you will learn to:

1. use the READ, DATA, and RESTORE statements
2. make horizontal bar graphs
3. make vertical bar graphs containing multiple bars
4. scale and label bar graphs.

THE READ, DATA, AND RESTORE STATEMENTS

The DATA statement must be used in the *deferred mode.* Although the READ and RESTORE statements are normally also used in the deferred mode, we will illustrate their use by storing data in a DATA statement (in the deferred mode) and then using the READ and RESTORE statements in the immediate mode.

Type in the statement **10 DATA 5,10** and then type:

```
READ A
?A
READ A
?A
READ A
```

as shown in Figure 10.1. The first time you type **READ A,** the first data value in the DATA statement (5) is stored in A. The second time you type **READ A,** the second data value in the DATA statement (10) is stored in A. The third time you type **READ A** an error message, **?OUT OF DATA ERROR,** is displayed, because there are no more data values in the DATA statement.

When a program is executed, a *pointer* points to the first data value in the DATA statement. (Two or more DATA statements in a program are treated as a single long DATA statement.) As data values are "used up" by being read in READ statements, the pointer moves along to the next unused data value. If the pointer gets to the end of the data values in the DATA statement, and another READ statement is executed, then the **?OUT OF DATA ERROR** message will be displayed.

```
10 DATA 5,10
READ A

READY.
?A
 5

READY.
READ A

READY.
?A
 10

READY.
READ A

?OUT OF DATA  ERROR
READY.
```

Figure 10.1 The READ statement reads successive values from a DATA statement.

The pointer can be reset at any time to the first data value in the DATA statement by using the statement **RESTORE.** Also, more than one value can be read with a single READ statement. In order to see this, type in the following statements as shown in Figure 10.2:

10 DATA 5, 10, 15, 20
READ A,B,C
? A,B,C
RESTORE
READ B,C
? A,B,C

Figure 10.2 The RESTORE statement moves the pointer to the first data value in the DATA statement.

```
10 DATA 5,10,15,20
READ A,B,C

READY.
? A,B,C
 5            10           15

READY.
RESTORE

READY.
READ B,C

READY.
? A,B,C
 5            5            10

READY.
```

Note that in this case the first READ statement stores the values 5, 10, and 15 in A, B, and C, respectively. The RESTORE statement then moves the pointer back to the first data value (5). Therefore, the next READ statement will store the values 5 and 10 in B and C, respectively. Note that the value of A remains unchanged and is still equal to 5.

Now add the second DATA statement **20 DATA 25,30,35.** This will automatically restore the pointer to the first data value (5) in line 10. Type the following statements as shown in Figure 10.3:

READ A,B,C
? A,B,C
READ A,B,C
? A,B,C
READ A,B

```
10 DATA 5,10,15,20
20 DATA 25,30,35
READ A,B,C

READY.
? A,B,C
 5            10           15

READY.
READ A,B,C

READY.
? A,B,C
 20           25           30

READY.
READ A,B

?OUT OF DATA  ERROR
READY.
```

Figure 10.3 There must be data values for *all* variable names in a READ statement.

DATA statements may occur anywhere in a program. Two or more DATA statements are effectively combined into one long DATA statement in the order in which they occur in the program. In the last READ statement in Figure 10.3, there is no value for B, and therefore the **?OUT OF DATA ERROR** message is displayed.

Strings can be included in a DATA statement. In this case the corresponding variable name in the READ statement must be a string variable. For example, change the DATA statement in line 20 to 20 DATA ACE, "LOGO+" and then type

READ A,B,C,D,A$,B$
? A,B,C,D,A$,B$

as shown in Figure 10.4. Note that the string ACE in the DATA statement does not have to be enclosed between quotation marks. However, graphic characters and strings containing blanks, commas, and colons must be enclosed between quotation marks.

Note also that the numerical variables A and B are completely different memory cells from the string

variables A$ and B$. The Commodore 64/VIC 20 will not get these mixed up.

The READ and DATA statements are particularly useful when you have a list of data whose values do not change in the program and which are read by the same READ statement. Examples of using these statements will be given in the following sections.

```
10 DATA 5,10,15,20
20 DATA ACE,*
30 READ A,B,C,D,A$,B$

READY.
? A,B,C,D,A$,B$
 5          10          15          20
ACE         *

READY.
```

Figure 10.4 String variables can be used in a READ statement to read strings in a DATA statement.

HORIZONTAL BAR GRAPHS

Bar graphs are very useful for providing a quick visual picture of the relative sizes of various quantities. The simplest kind of bar graph that you can draw on the Commodore 64/VIC 20 is a horizontal line whose length is proportional to a particular quantity.

As an example, suppose you want to compare graphically the four values 12, 21, 5, and 17. You can plot four lines with lengths 12, 21, 5, and 17 using the program shown in Figure 10.5.

Figure 10.5 Program to plot four lines of length 12, 21, 5, and 17.

```
LIST

10 REM BAR GRAPH EXAMPLE
20 DATA 12,21,5,17
30 G$="*"
50 FOR J=1 TO 4
60 READ L:GOSUB 400
70 NEXTJ
90 END
400 FOR I=1 TO L:PRINT G$;:NEXT:PRINT:P
INT:RETURN
READY.
RUN
************

********************

*****

*****************

READY.
```

In this program line 20 is a DATA statement that contains the lengths of the four bars to be plotted. Line 30 defines the graphic character that will be used to draw the lines and stores this character in the string variable G$. Different types of bars can be plotted by using different graphic characters. Lines 50-70 form a FOR...NEXT loop that is executed once for each bar to be plotted.

Within this loop, line 60 reads the next length from the values given in the DATA statement and stores this length in the memory cell L. A bar of length L is then plotted using the subroutine in line 400. This subroutine prints L copies of the graphic symbol stored in G$ to form the bar. The first PRINT statement at the end of the line 400 causes the cursor to be moved to the beginning of the next line. The second PRINT statement causes a line to be skipped.

Note that the END statement in line 90 is necessary to prevent the program from executing line 400 again. (This would produce the error message **?RETURN WITHOUT GOSUB ERROR IN 400**.)

The basic ideas shown in Figure 10.5 can be used to produce useful bar graphs of real data, as illustrated in the next section.

Population of the New England States

The populations of the six New England states are shown in Table 10.1. The program given in Figure 10.5 has been modified as shown in Figure 10.6 to plot six bars corresponding to the data in Table 10.1

TABLE 10.1 *Population of the New England States*

State	*Population*
ME	1,124,660
NH	920,610
VT	511,456
MA	5,737,037
CT	3,107,576
RI	947,154

Lines 100-150 are six DATA statements containing the information in Table 10.1. Note that each DATA statement contains a string (the name of the state) and a numerical value (the state's population). For each pass through the FOR...NEXT loop (lines 50-80), line 60 stores the next state name in S$ and its population in P.

Each graphic symbol defined in line 30 will represent a certain number of people. In order to determine how many people this should be, you must choose a value that will insure that the longest bar will fit on the screen. The state name plus a space will use

Figure 10.6 Program to produce a bar graph of the data in Table 10.1.

```
10 REM POPULATION BAR GRAPH
20 N=6
30 G$="▒"
40 PRINT"■■■■     POPULATION OF"
45 PRINT "  NEW ENGLAND STATES":PRINT:PRINT
50 FOR J=1 TO N
60 READ S$,P
70 L=P/200000+.5:GOSUB 400
80 NEXTJ
90 END
100 DATA ME,1124660
110 DATA NH,920610
120 DATA VT,511456
130 DATA MA,5737037
140 DATA CT,3107576
150 DATA RI,947154
400 PRINTS$;" ";:FOR I=1 TO L:PRINT G$;:NEXT:PRINT:PRINT:RETURN

READY.
```

three columns of a screen line. Therefore, the longest possible bar is nineteen spaces long on the VIC 20 and thirty-seven spaces long on the Commodore 64. The highest population is that of Massachusetts, 5,737,037. Therefore, each graphic symbol must represent more than 5,737,037/19 = 301949 people in the case of the VIC 20 and more than 5,737,037/37 = 155055 in the case of the Commodore 64. We will therefore choose each graphic symbol to represent a population of 400,000 in the case of the VIC 20 and 200,000 in the case of the Commodore 64. The latter value is used (in line 70) in Figure 10.6. VIC users should change it to 400,000.

Given a population P, line 70 calculates the number of graphic symbols to be plotted. This is the length of the bar, L. In the equation

$$L = P/200000 + 0.5$$

(for the Commodore 64) the 0.5 will round to the nearest 200,000 people, because the above equation is equivalent to the equation

$$L = \frac{P + 100000}{200000}$$

Note that the number of symbols plotted by line 400 of the subroutine will be equal to the integer part of L. The analogous equation for the VIC 20 rounds to the nearest 400,000 people.

The subroutine in line 400 has been modified to print the state name, stored in S$, to the left of each bar. Line 40 clears the screen, moves the cursor down two lines, prints the title of the graph, and then skips two lines. The result of running this program on the Commodore 64 is shown in Figure 10.7.

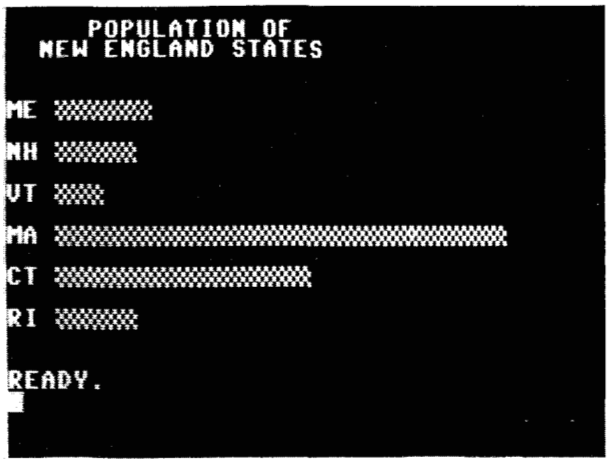


Figure 10.7 Result of running the program shown in Figure 10.6

Adding a Scale

Although the bar graph shown in Figure 10.7 illustrates the relative sizes of the six state populations, it does not provide the actual values of these populations. We can correct this by adding a scale to the bottom of the graph.

Since each graphic symbol represents a population of 400,000 in the case of the VIC 20, then five squares represent two million people. A subroutine that prints such a scale is shown in Figure 10.8a. In the case of the Commodore 64, five squares represent only one million people. A subroutine that prints such a scale is shown in Figure 10.8b. The latter scale shows the numbers 0,1,2,3,4,5, and 6 (millions) whereas the scale for the VIC 20 shows only 0,2,4, and 6. These subroutines are called in line 85 of the revised main program. The main program for the Commodore 64 is

shown in Figure 10.9. On the VIC 20, change the 200 000 in line 70 to 400 000. The result of executing these new programs is shown in Figure 10.10.

Splitting the Difference

It is possible to make these population bar graphs a little more accurate by using the left graphic symbol on the – key to represent half of the symbol in G$, in this case, 200,000 people on the VIC 20 and 100,000 people on the Commodore 64. This key can then be used at the end of a bar to make the length of the bar proportional to the population to within the nearest 200,000 people on the VIC 20 and 100,000 people on the Commodore 64. The relationship between the previous program and the new one in the case of the Commodore 64 is shown in Figure 10.11.

In order to add this new feature, we will eliminate the 0.5 in the equation on line 70 of the program in Figure 10.9. We will then plot a number of full squares equal to the integer part of L. The fractional part of L will determine whether to plot another full square, a half-square, or no square at all, according to the following scheme:

Fractional Part of L	*Plot*
less than 0.25	no square
between 0.25 and 0.75	half-square
greater than 0.75	full square

We can accomplish this by modifying the subroutine that draws the bar, as shown in Figure 10.12. The modified main program for the Commodore 64 is shown in Figure 10.13. VIC 20 users should use 400,000 in line 70.

In Figure 10.12, line 410 plots L full squares (L is calculated in line 70 in Figure 10.13). The fractional part of L is calculated in line 420 in Figure 10.12. You should convince yourself (try some examples) that this

Figure 10.8 Subroutine to display scale: (a) VIC 20 and (b) Commodore 64.

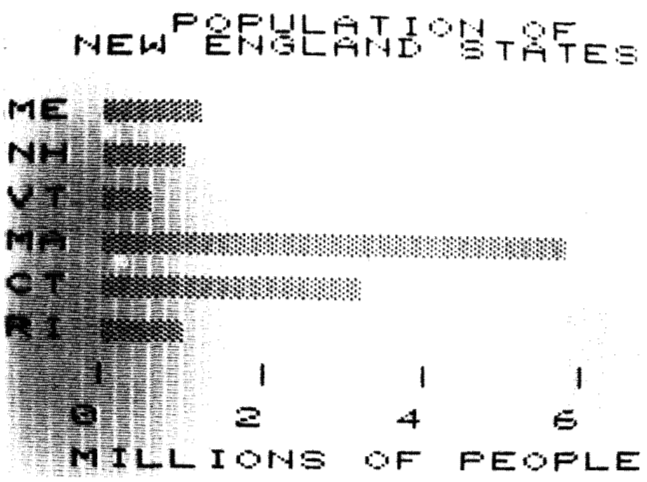
(a)
```
600 REM ADD SCALE
610 PRINT "  ";:FOR I=1 TO 4:PRINT" |    ";:NEXT:PRINT
620 PRINT" ";:FOR I=0 TO 3:PRINT2*I;"   ";:NEXT:PRINT:PRINT
630 PRINT TAB(2);"MILLIONS OF PEOPLE"
640 RETURN

READY.
```

(b)
```
600 REM ADD SCALE
610 PRINT "  ";:FOR I=1 TO 7:PRINT" |    ";:NEXT:PRINT
620 PRINT" ";:FOR I=0 TO 6:PRINTI;"   ";:NEXT:PRINT:PRINT
630 PRINT TAB(8);"MILLIONS OF PEOPLE"

READY.
```

Figure 10.9 Revised main program that calls subroutine to add a scale.

```
10 REM POPULATION BAR GRAPH
20 N=6
30 G$="▒"
40 PRINT"■■■     POPULATION OF"
45 PRINT "  NEW ENGLAND STATES":PRINT:PRINT
50 FOR J=1 TO N
60 READ S$,P
70 L=P/200000+.5:GOSUB 400
80 NEXTJ
85 GOSUB 600:REM ADD SCALE
90 END
100 DATA ME,1124660
110 DATA NH,920610
120 DATA VT,511456
130 DATA MA,5737037
140 DATA CT,3107576
150 DATA RI,947154
400 PRINTS$;" ";:FOR I=1 TO L:PRINT G$;:NEXT:PRINT:PRINT:RETURN

READY.
```

(a)



(b)

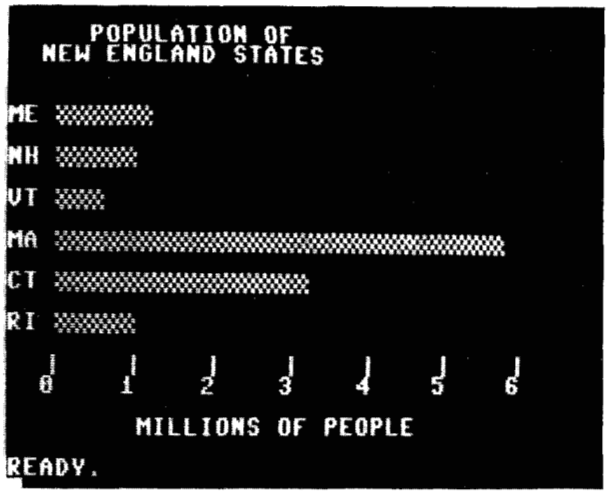


Figure 10.10 Result of running the program shown in Figure 10.9: (a) VIC 20 and (b) Commodore 64.

equation works. If E is greater than 75 (corresponding to a fractional part of L greater than 0.75), then an extra full square is plotted by line 430. If E is greater than 25 but less than 75, a half-square is plotted in line 440. Line 450 returns the cursor if appropriate, skips a line, and returns from the subroutine.

Figure 10.12 Modified subroutine to draw a bar with a half-square at the end of the bar.

```
400 REM DRAW BAR WITH HALF-SQUARE
410 PRINT S$;" ";:FOR I=1 TO L:PRINT G$;:NEXT
420 E=L*100-INT(L)*100
430 IF E>75 THEN PRINT "▒";:GOTO 450
440 IF E>25 THEN PRINT "▌";
450 PRINT:PRINT:RETURN

READY.
```

The result of running this new program (for the Commodore 64) in Figures 10.12 and 10.13 is shown in Figure 10.14. If you compare this result carefully with the bar graph in Figure 10.10, you will note that the bars for all states now end with half-squares.

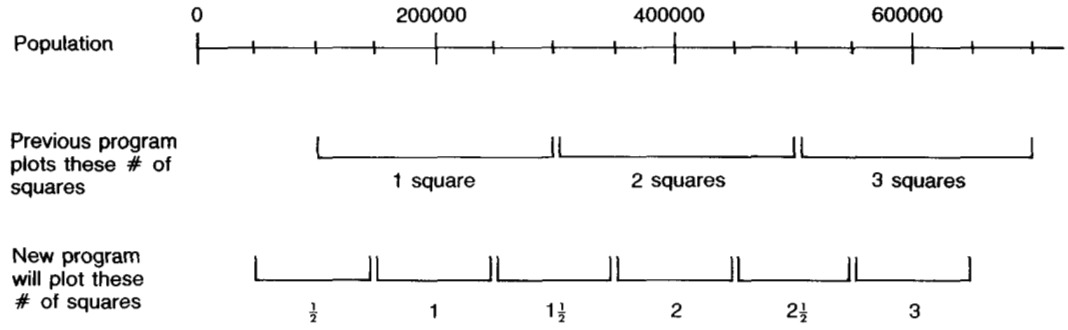
VERTICAL BAR GRAPHS

You can also draw vertical bars by using the "Move to X,Y" subroutine. This subroutine is shown in Figure 10.15 (line 500), together with a subroutine that plots a vertical bar from Y1 to Y2 (line 400). In line 410 if

Figure 10.13 The 0.5 has been dropped in the calculation of L in Line 70.

```
10 REM POPULATION BAR GRAPH
20 N=6
30 G$="▒"
40 PRINT"■■■     POPULATION OF"
45 PRINT "  NEW ENGLAND STATES":PRINT:PRINT
50 FOR J=1 TO N
60 READ S$,P
70 L=P/200000:GOSUB 400
80 NEXTJ
85 GOSUB 600:REM ADD SCALE
90 END
100 DATA ME,1124660
110 DATA NH,920610
120 DATA VT,511456
130 DATA MA,5737037
140 DATA CT,3107576
150 DATA RI,947154

READY.
```

Figure 10.11 The accuracy of a bar can sometimes be increased by adding a half-square at the end.



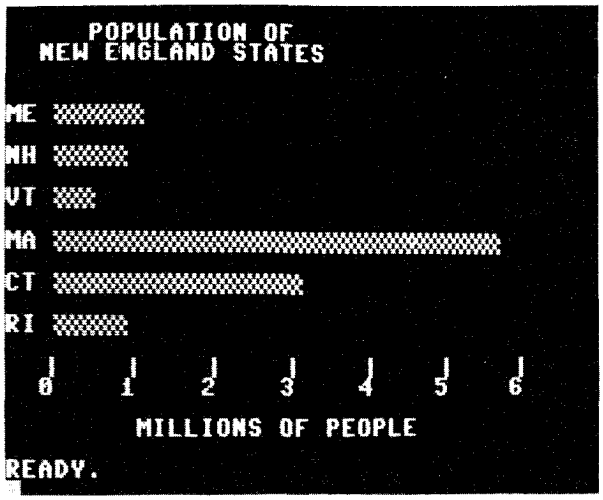


Figure 10.14 Result of running program given in Figures 10.12 and 10.13.

Y3=−1, then the bar will be plotted from bottom to top, and Y2 should be less than Y1. If Y3=1 then the bar will be plotted from top to bottom, and Y2 should be greater than Y1. The graphic character used to form the bar is stored in the string variable B$ and is printed in line 420.

Figure 10.15 Subroutine to plot a vertical bar from Y1 to Y2.

```
400 REM PLOT BAR FROM Y1 TO Y2
410 FOR Y=Y1 TO Y2 STEP Y3
420 GOSUB 500:PRINT B$
430 NEXT Y:RETURN
500 REM MOVE TO X,Y ON SCREEN
510 PRINT "@";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

In order to test this subroutine type in lines 400-530 shown in Figure 10.15. Then clear the screen and type the following statements in the immediate mode:

Y1=15: Y2=5: Y=−1: X=15: B$="LOGO+": GOSUB 400

The result should be as shown in Figure 10.16.

Plotting a Vertical Bar of Length V

We will now develop a general subroutine that will plot a vertical bar with a length proportional to the value V. The value of V can be either positive or negative. For a negative value of V the bar should be plotted in the downward direction. In order to increase the accuracy of the bar lengths, we will allow half-

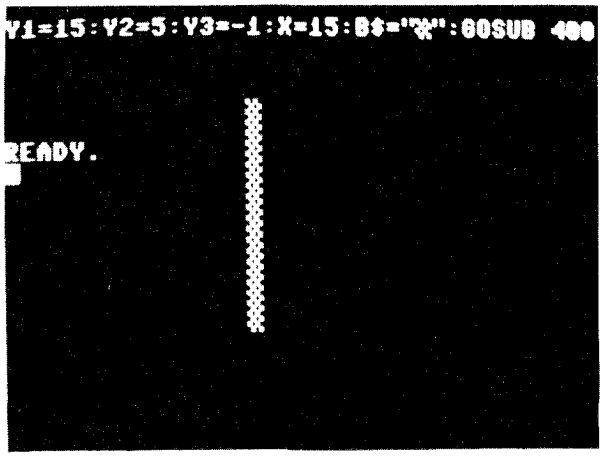


Figure 10.16 Test of subroutine shown in Figure 10.15.

squares to be plotted, as described in the section on "splitting the difference."

The half-square character that is used can be different for positive and negative bars. For example, if the bar is made with reverse video space (B$), then a positive half-square would be provided by the graphic symbol LOGO I. The negative half-square would be the reverse video of this same symbol as shown in Figure 10.17.

This subroutine will be used to plot the appropriate bar, given the following values:

V=value to be plotted
B$=full-square graphic character
B2$=positive half-square graphic character
B3$=negative half-square graphic character

Figure 10.17 "Half-square" characters are different for positive (B2$) and negative (B3$) bars.

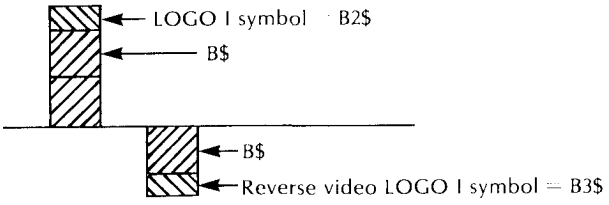


Figure 10.18 shows the ranges of values for which various combinations of squares will be plotted. The bottom of row 20 on the screen will define the "zero" value of V. From Figure 10.18 you see that a value of V between 0.25 and 0.75 will result in a half-square character being printed in row Y=20.

Similarly, a value of V between −1.25 and −1.75 will result in a full square character B$ being printed in row Y=21 and a half-square character B3$ being printed in row Y=22.

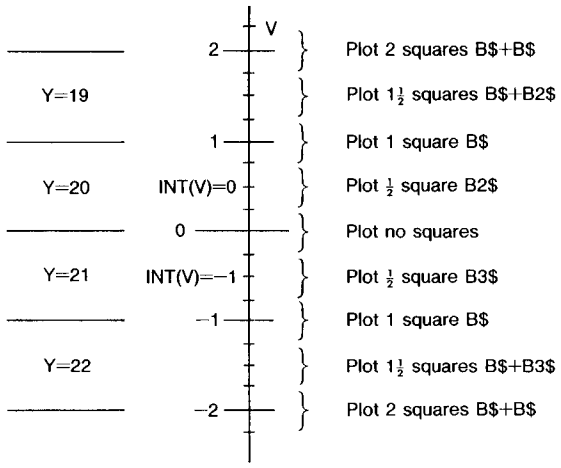


Figure 10.18 Screen layout for plotting vertical bar with length proportional to the value V.

When plotting a bar, the subroutine shown in Figure 10.15 can be used to plot the portion of the bar made up of the full-square B$ character. The end of the bar can then be plotted with a full-square, half-square, or no square, depending on the value of V, as shown in Figure 10.18. An algorithm to plot this bar is given in pseudocode in Figure 10.19. The case of V<0 will be considered in a separate subroutine. If the integer part of V, V1, is equal to zero, then the value of V is between 0 and 1. In this case we simply want to plot the end of the bar without plotting the B$ bar at all. Otherwise, we will plot a B$ bar of length V1, followed by the end of the bar.

The algorithm to plot the end of a positive bar is given in pseudocode in Figure 10.20. The idea here is the same as described in the "splitting the difference" section above. The fractional part of V on a scale of 0-100 is given by E.

BASIC subroutines for implementing the algorithms in Figures 10.19 and 10.20 are given in Figure 10.21. Lines 800-870 implement the algorithm in Figure 10.19, and lines 900-950 implement the algorithm in Figure 10.20. In line 830 the value of Y2 is set equal to Y1+1=21 before calling the subroutine in line 900. In line 920 of this subroutine the cursor is moved

Figure 10.19 Algorithm to plot a bar of length proportional to the value V.

```
Y1=20
VI=integer part of V
if V<0
then consider case of V<0
else if VI=0
     then plot end of positive bar
     else plot B$ bar of length VI
          plot end of positive bar
```

Figure 10.20 Algorithm to plot the positive end of the bar.

```
E=V*100-VI*100
Move cursor to position above top of bar
if E>75
then print full square
else if E>25
     then print half-square
```

to a value of Y=Y2−1=20, which is the location to be plotted if V1=0. In line 840, Y2 is equated to 21−V1 and Y3 is equated to −1. This will result in a B$ bar of length V1 being plotted from bottom to top by the subroutine at line 400.

The algorithm for considering the case of V<0 is given in Figure 10.22, and the algorithm to plot the negative end of a bar is given in Figure 10.23. Remember that V1=INT(V) will be equal to −1 for values of V between 0 and −1.

Figure 10.21 Subroutine to plot bar of length V (line 800) and subroutine to plot positive end of bar (line 900).

```
800 REM  PLOT VALUE V USING B$,B2$,B3$
810 Y1=20:VI=INT(V)
820 IF V<0 THEN GOSUB 1000:RETURN
830 IF VI=0 THEN Y2=Y1+1:GOSUB 900:RETURN
840 Y2=21-VI:Y3=-1
850 GOSUB 400:REM  PLOT B$ BAR
860 GOSUB 900:REM  PLOT + END OF BAR
870 RETURN
900 REM  PLOT + END OF BAR
910 E=V*100-VI*100
920 Y=Y2-1:GOSUB 500
930 IF E>75 THEN PRINT B$:GOTO 950
940 IF E>25 THEN PRINT B2$
950 RETURN

READY.
```

Figure 10.22 Algorithm to consider case of V<0.

```
Y1=21
if VI= −1
then plot end of negative bar
else plot B$ bar of length VI −1
     plot end of negative bar
```

Figure 10.23 Algorithm to plot negative end of bar.

```
E=|V*100|−|(VI+1)*100|
Move cursor to position below bottom of bar
if E>75
then print full square
else if E>25
     then print half-square
```

BASIC subroutines for implementing these algorithms are given in Figure 10.24. Lines 1000-1060 implement the algorithm in Figure 10.22, and lines 1100-1150 implement the algorithm in Figure 10.23.

Figure 10.24 Subroutine to consider the case of V<0 (line 1000) and subroutine to plot negative end of bar (line 1100).

```
1000 REM  CASE OF V<0
1010 Y1=21
1020 IF VI=-1 THEN Y2=Y1-1:GOSUB 1100:RETURN
1030 Y2=19-VI:Y3=1
1040 GOSUB 400:REM  PLOT B$ BAR
1050 GOSUB 1100:REM  PLOT - END OF BAR
1060 RETURN
1100 REM  PLOT - END OF BAR
1110 E=ABS(V*100)-ABS((VI+1)*100)
1120 Y=Y2+1:GOSUB 500
1130 IF E>75 THEN PRINT B$:GOTO 1150
1140 IF E>25 THEN PRINT B3$
1150 RETURN

READY.
```

In order to test these subroutines, type in the statements shown in Figures 10.21 and 10.24. Then clear the screen and type the following two lines in the *immediate mode*. You should obtain the result shown in Figure 10.25.

B$="RVS space OFF": B2$="LOGO I":
B3$="RVS LOGO I OFF"

V=+3.0: X=10: GOSUB 800

The line beginning with **V=+3.0** is "live" on the screen. This means that if you edit this line by changing the values of V and X, then when you press RETURN the new statement will be executed, and a new bar will be plotted. Edit this line to plot the following bars:

V=+0.6: X=11: GOSUB 800

V=−0.6: X=12: GOSUB 800

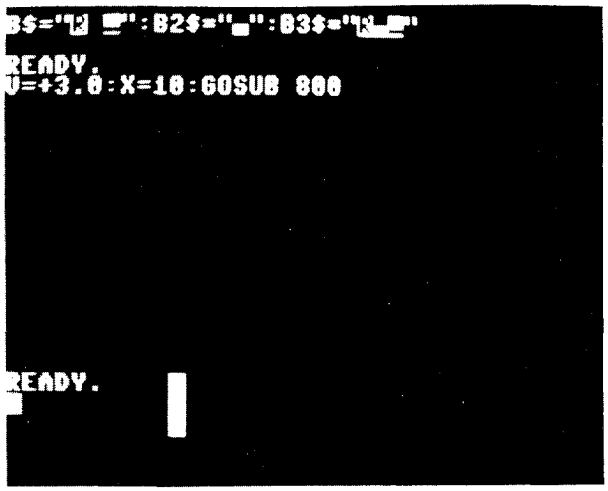


Figure 10.25 Testing subroutines given in Figures 10.21 and 10.24.

V=+1.3: X=13: GOSUB 800

V=−1.3: X=14: GOSUB 800

You should obtain the bars shown in Figure 10.26. Note that the last bar caused the top line to scroll off the screen. Try some other values.

This technique of using the immediate mode of execution to test subroutines is a good method, because you can make lots of tests without disturbing the program that you have stored in the computer.

We will now use these subroutines to plot a "multiple" bar graph that will display multi-year economic data.

Multiple Bar Graph for Economic Data

In this section we will develop a program to plot a "multiple" bar graph of the economic data given in Table 10.2.

For each year we will plot four bars, one for each economic factor. Each of these four bars will use a

TABLE 10.2 Economic Data adapted from data on p. 67 of the March 10, 1980 issue of *TIME* Magazine

	1976	*1977*	*1978*	*1979*	*1980*	
Inflation % change in C.P.I.	4.7	6.8	9.0	13.3	18.2	Jan. change at compound annual rate
Unemployment % of civilian labor force	7.8	7.0	6.0	5.8	6.2	Jan.
Growth % change in real G.N.P.	4.8	5.8	4.9	0.8	1.7	Projected 1st Q.
Personal income % change per capita	2.8	4.5	3.4	−0.8	−0.5	Projected 1st Q.

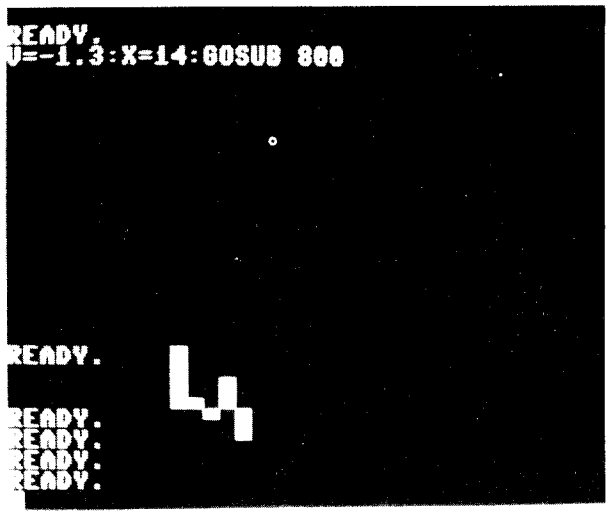


Figure 10.26 Further tests of subroutine given in Figures 10.21 and 10.24.

different color. The colors used are given in Table 10.3. Note that only "personal income" has negative values. The effect of using the same symbol for B$ and B2$ is to round to the next higher whole number when the fractional value exceeds 0.25.

TABLE 10.3 Colors Used in Economy Bar Graph

Data	*Color*
Inflation	yellow
Unemployment	green
Growth	purple
Personal Income	cyan

It is fairly easy to see that the twenty-two character width of the VIC 20 screen will accommodate a vertical bar graph for only four of the five years of data given in Table 10.2. However, if we develop a program for creating a full bar graph on the Commodore 64, it turns out only a few minor changes in the program can create a very similar display on the VIC 20 with the 1980 data deleted. Therefore, we will direct attention subsequently on the Commodore 64 program and then modify it for the VIC 20 at the end.

The main program for plotting this bar graph on the Commodore 64 is shown in Figure 10.27. Line 11 clears the screen. Line 15 defines the four color squares given in Table 10.3. Line 16 defines the needed color half-square graphic symbols. Lines 20-60 are DATA statements containing the data given in Table 10.2. Note that each DATA statement contains the data for one year, starting with 1976.

The value of X is initialized to zero in line 65. This is the column number in which the first bar will be plotted. The FOR...NEXT loop in lines 70-90 is executed five times (once for each year). Each time through this loop, four bars are plotted, corresponding to the data for one year. This is done by a subroutine that starts at line 200. Line 100 calls a subroutine at line 600 that prints the heading and scale for the graph. Line 150 branches on itself to prevent the READY message from being displayed.

The subroutine to plot the next four bars of the graph is shown in Figure 10.28. The READ statement in line 210 reads the next four values of inflation, unemployment, growth, and income, and stores these values in the memory cells I, U, G, and M. The inflation bar is plotted by lines 230-240, using the subroutine shown at line 800 of Figure 10.21. Note that the value of V has been assigned the inflation value I, B$ has been assigned the inflation full-character I1$, and B2$ has been assigned the inflation half-character I2$. Line 250 increments the column number by one so that the next bar will be plotted in the adjacent column. Lines 260-270 plot the unemployment bar. Lines 280-300 move to the next column and plot the growth bar. Lines 310-330 plot the income bar in the next column. Note that B3$ had to be defined in

Figure 10.27 Main program for plotting economy bar graph on the Commodore 64.

```
10 REM  THE ECONOMY
11 PRINT"□"
15 I1$="■■■■■■■■■■■■■■■■■■■■■■":U1$="■■■■■■■■■■■■■■■■■■■■■■":G1$="■■■■■■■■■■■■■■■■■■■■■■":M1$="■■■■■■■■■■■■■■■■■■■■■■"
16 I2$="▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌":U2$="▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌":G2$="▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌":M2$="▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌":M3$="▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌■■■■■■■■■■■■■■■■■■■■■■"
20 DATA 4.7,7.8,4.8,2.8
30 DATA 6.8,7,5.8,4.5
40 DATA 9,6,4.9,3.4
50 DATA 13.3,5.8,0.8,-.8
60 DATA 18.2,6.2,1.7,-.5
65 X=0
70 FOR J=1 TO 5
80 GOSUB 200:REM  PLOT 4 BARS
90 NEXTJ
100 GOSUB 600:REM  PRINT HEADING & SCALE
150 GOTO 150

READY.
```

line 320, because some of the income data is negative. Line 340 increases the column number X by 5. This will leave a four-column space between each group of four bars.

Figure 10.28 Subroutine to plot the next four bars on the graph.

```
200 REM  PLOT NEXT 4 BARS
210 READ I,U,G,M
230 V=I:B$=I1$:B2$=I2$
240 GOSUB 800:REM  PLOT TOTAL BAR
250 X=X+1
260 V=U:B$=U1$:B2$=U2$
270 GOSUB 800
280 X=X+1
290 V=G:B$=G1$:B2$=G2$
300 GOSUB 800
310 X=X+1
320 V=M:B$=M1$:B2$=M2$:B3$=M3$
330 GOSUB 800
340 X=X+5
350 RETURN

READY.
```

The subroutine to display the years at the bottom of the graph, a heading in the upper left corner, and a scale along the right side is shown in Figure 10.29. Lines 610-620 print the five years under the appropriate bar graphs. Lines 630-680 print the title of the graph and the legend to explain the four graphic symbols. Lines 690-730 print the scale along the right side of the graph.

The entire program for the Commodore 64 to plot the economy bar graph is given by the statements in Figures 10.27, 10.28, 10.15, 10.29, 10.21, and 10.24. The result of running this program is shown in Figure 10.30.

Although this program is relatively long, you can see that by breaking the program into functional modules you can more easily keep track of what is going on. This will also make it easier for you to modify this program to suit your own needs.

To modify the program for use on the VIC 20, it is necessary to drop four bars representing one year of data. We chose to drop the data for 1980. In this case, line 60 of the program containing the data for 1980 becomes unnecessary. In line 70, we change 5 to 4 so as to plot only four groups of bars. By changing 5 to 2 in line 340, we reduce the spacing between groups of four bars to a single column. To reduce the spacing the same for the titles under the bars, we change line 620 to read: 620 PRINT"1976 1977 1978 1979"; . The final semicolon is necessary to prevent an undesired scrolling of the screen. Finally, by changing X=36 to X=18 in line 690, we move the scale over to touch the final group of vertical bars. In Figure 10.31, we show the resulting bar graph of economic data produced by the VIC 20.

Figure 10.30 Bar graph of economic data given in Table 10.2 on the Commodore 64.

Figure 10.29 Subroutine to display heading and scale.

```
600 REM  DISPLAY YEARS, HEADING, AND SCALE
610 X=0:Y=22:GOSUB 500
620 PRINT"1976    1977    1978    1979    1980"
630 PRINT"■■■THE ECONOMY"
640 PRINT
650 PRINT I1$;I1$;"■ INFLATION"
660 PRINT U1$;U1$;"■ UNEMPLOYMENT"
670 PRINT G1$;G1$;"■ GROWTH IN GNP"
680 PRINT M1$;M1$;"■ PERSONAL INCOME"
690 X=36:Y=1:GOSUB 500:PRINT"%";
700 Y=5:GOSUB 500:PRINT"_15";
710 Y=10:GOSUB 500:PRINT"_10";
720 Y=15:GOSUB 500:PRINT"__5";
730 Y=20:GOSUB 500:PRINT"__0"
740 RETURN

READY.
```

Figure 10.31 Bar graph of economic data given in Table 10.2 on the VIC 20.

EXERCISE 10-1

The following table shows the amount of gasoline required to fill the gas tank of a certain station wagon:

Speedometer Reading	*Gallons to Fill Tank*
93769.3	15.5
94034.6	15.2
94249.1	14.8
94376.6	9.0
94558.0	10.5
94778.2	12.8
95037.0	14.9
95258.0	14.7
95499.3	15.3
95732.7	20.3
95941.2	15.0

Write and run a program that will:

a. store this data in DATA statements,

b. compute the gas mileage in miles per gallon for each fill-up, and plot the results as a bar graph,

c. compute and print out the average miles per gallon for all fill-ups shown in the table.

EXERCISE 10-2

Each entry in the following table gives a nation, its population, and its area (in square miles):

Nation	*Population*	*Area*
Australia	14,620,000	2,965,368
Canada	23,940,000	3,851,809
China	1,027,000,000	3,691,502
India	667,326,000	1,178,995
Japan	116,780,000	143,689
Soviet Union	266,670,000	8,649,489
United Kingdom	56,235,500	94,500
United States	226,504,825	3,615,122

Write and run a program that will:

a. store this data in DATA statements,

b. compute the population density for each nation in people per square mile, and

c. plot the results as a bar graph.

11 LEARNING TO GET WHAT YOU WANT: AN ALTERNATIVE TO INPUT

The only method you now know for entering data from the keyboard is to use the INPUT statement. One potential disadvantage of using the INPUT statement is that the data is not accepted by the Commodore 64/VIC 20 until you press the RETURN key. Sometimes you would like the Commodore 64/VIC 20 to accept a single character from the keyboard as soon as its key is pressed. The GET statement will do this. This statement is particularly useful for interactive games or other programs in which you need an immediate response to the pressing of a key.

In this chapter you will learn how to:

1. use the GET statement
2. write a program that will allow you to draw sketches interactively on the screen
3. write a program that will allow you to move a fighter plane around the screen and fire phasers from the plane's wings.

THE GET STATEMENT

The GET statement is used to store a single character typed on the keyboard in a string variable such as A$. The form of the GET statement is **GET A$** where A$ can be any string variable. In order to understand how the GET statement works you need to know what happens when you press a key.

Ten memory locations in the computer's RAM (read-write memory) have been set aside to form a keyboard input *buffer*. This buffer temporarily stores each character that you type. When you press a key, the corresponding character (actually a numerical code representing the character) is stored in the first memory location of the keyboard buffer. If a BASIC program then encounters the statement **GET A$** the character stored in the keyboard buffer (corresponding to the key you pressed) will be moved into memory cell A$. If the statement is executed without a key being pressed, the buffer is empty, and the string variable A$ is assigned to the *null* string given by ""

Because the Commodore 64/VIC 20 will not know when you are going to press a key, and therefore when to execute a GET statement, you must program a special loop whenever you want to GET a character from the keyboard. This loop can be written as follows:

```
10 GET A$: IF A$="" THEN 10
```

In this statement, when **GET A$** is executed, the next character in the keyboard buffer is stored in A$. If the keyboard buffer is empty (that is, if you have not pressed a key), then A$ will be assigned the null string "". If this is the case, line 10 will just branch to itself and execute the GET statement again. This loop will

continue until you press a key which will store a character in the keyboard buffer. At this point the statement following line 10 will be executed.

In order to test the GET statement type in line 10 above followed by the line **20 ? A$;: GOTO 10.** If you run this two-line program, the Commodore 64/VIC 20 should print on the screen any character that you type, as shown in Figure 11.1.

Run this program and try lots of keys. Note that you can print any character, including the graphic symbols. Also note that the RETURN, reverse video, color, and cursor keys work in the normal way.

In the program shown in Figure 11.1, each character you type is stored in the first location of the keyboard buffer. This is because the program is executed so rapidly that as soon as you press a key, the GET statement is immediately executed again and will read the character and therefore empty the buffer. Only if you can type in more than one character before the GET statement is executed a second time will these additional characters be stored in the keyboard buffer. You will have an opportunity to observe this effect in the plane program later in this chapter.

Figure 11.1 This program GETs a character from the keyboard buffer and displays it at the next screen location.

```
10 GET A$: IF A$="" THEN 10
20 ? A$;: GOTO 10
RUN
THIS PROGRAM WILL PRINT ANY CHARACTER
THAT YOU TYPE.  NOTE THAT PRINTING THE
RETURN KEY CAUSES THE CURSOR TO MOVE TO
THE BEGINNING OF THE NEXT LINE; THE
CURSOR IS INVISIBLE BECAUSE WE ARE ONLY
EXECUTING THE PROGRAM SHOWN ABOVE, AND
WE DID NOT PROGRAM IN A CURSOR!
TO STOP THE PROGRAM, JUST PRESS THE
STOP KEY LIKE THIS:
BREAK IN 10
READY.
```

DRAWING PICTURES INTERACTIVELY

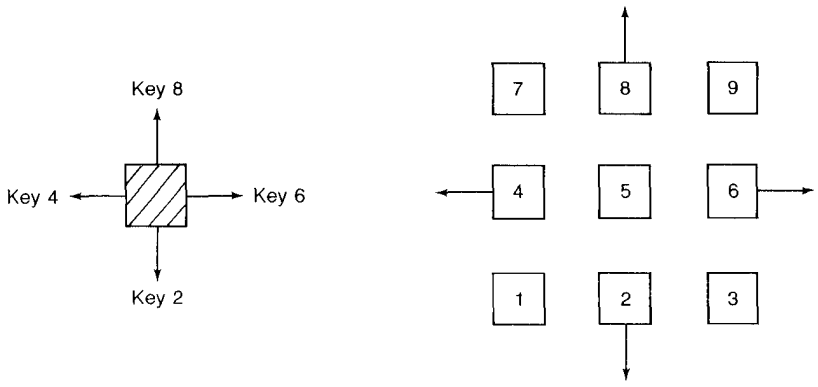
In order to draw a picture interactively, we will write a program that will cause a graphic symbol to be moved left, right, up, or down by typing A, D, W, and X, respectively, as shown in Figure 11.2.

A program to accomplish this is shown in Figure 11.3. Lines 500-530 contain the "Move to X,Y" subroutine described in Chapter 9. Line 5 clears the screen. Lines 10-20 move the cursor to location X=10, Y=10 and print the graphic character LOGO+. Line 30 is the GET loop just described. This statement will loop on itself until a key is pressed. When a key is pressed, lines 40–70 check to see if the key was an A, D, W, or X. If it was A, the value of X is decremented by one, and the program branches to line 20, which will move the cursor one space left and print a new graphic character. Line 30 will then wait for another key to be pressed. Note that moving left corresponds to decrementing X, and moving right corresponds to incrementing X. Moving up corresponds to decrementing Y while moving down corresponds to incrementing Y. If the key pressed is not A, D, W, or X, then line 95 will

Figure 11.3 Program for drawing pictures interactively on the screen.

```
2 REM INTERACTIVE DRAWING
5 PRINT"□"
10 X=10:Y=10:B$="▒"
20 GOSUB 500:PRINT B$
30 GET A$:IF A$="" THEN 30
40 IF A$="A" THEN X=X-1:GOTO 20
50 IF A$="D" THEN X=X+1:GOTO 20
60 IF A$="W" THEN Y=Y-1:GOTO 20
70 IF A$="X" THEN Y=Y+1:GOTO 20
95 GOTO 30
500 REM  MOVE TO X,Y
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

Figure 11.2 Move graphic symbol left, right, up, or down using keys A, D, W, and X.



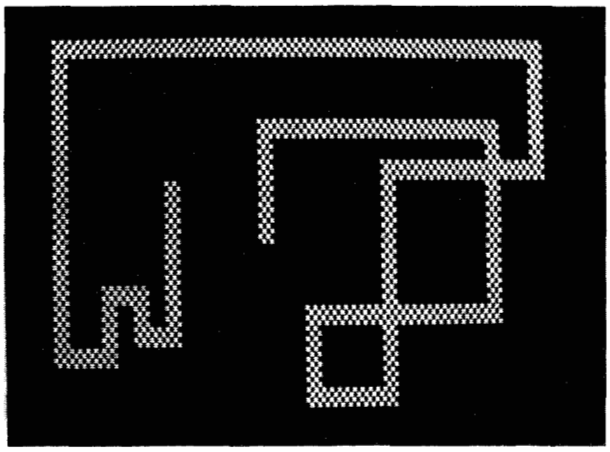


Figure 11.4 Sample run of program shown in Figure 11.3.

branch back to line 30 and wait for another key to be pressed.

Type in this program and run it. A sample run is shown in Figure 11.4.

Making Diagonal Lines

As an exercise you should try to modify the program shown in Figure 11.3 to make it draw diagonal lines using keys Z, C, Q, and E, as shown in Figure 11.5.

Figure 11.5 Move graphic symbol diagonally using keys Z, C, Q, and E.

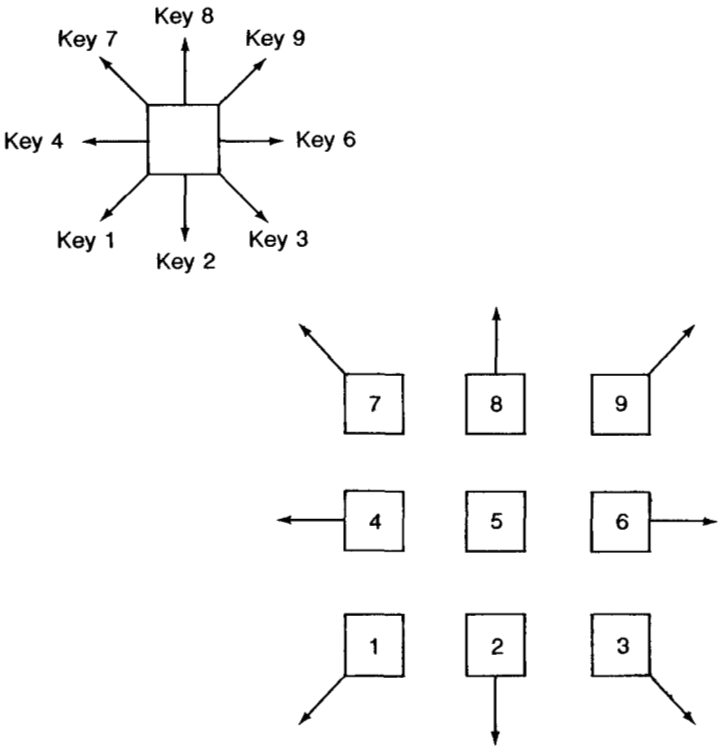


Figure 11.6 Modification of program shown in Figure 11.3 to allow drawing of diagonal lines.

```
2 REM INTERACTIVE DRAWING
5 PRINT"□"
10 X=10:Y=10:B$="▒"
20 GOSUB 500:PRINT B$
30 GET A$:IF A$="" THEN 30
40 IF A$="A" THEN X=X-1:GOTO 20
50 IF A$="D" THEN X=X+1:GOTO 20
60 IF A$="W" THEN Y=Y-1:GOTO 20
70 IF A$="X" THEN Y=Y+1:GOTO 20
80 IF A$="Z" THEN X=X-1:Y=Y+1:GOTO 20
90 IF A$="C" THEN X=X+1:Y=Y+1:GOTO 20
100 IF A$="Q" THEN X=X-1:Y=Y-1:GOTO 20
110 IF A$="E" THEN X=X+1:Y=Y-1:GOTO 20
190 GOTO 30
500 REM  MOVE TO X,Y
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

The listing of a program that will allow drawing of diagonal lines is shown in Figure 11.6. Lines 80-110 have been added to check for the pressing of keys Z, C, Q, or E. For example, if key Z is pressed, you want to move the cursor left 1 and down 1. This is accomplished by decrementing X by 1 (X=X−1) and incrementing Y by 1 (Y=Y+1) as shown in line 80. By branching to line 20 the cursor is moved to this new

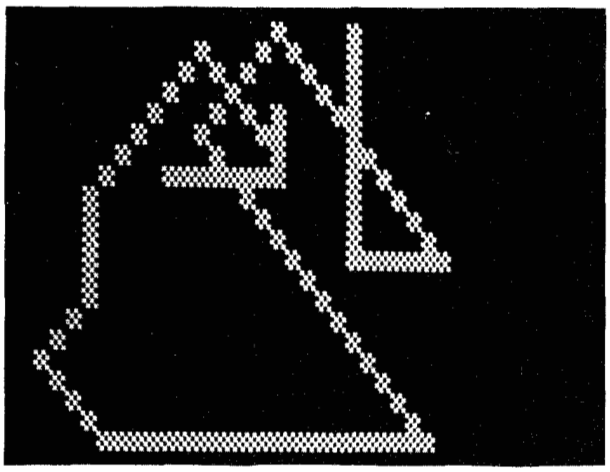


Figure 11.7 Sample run of program shown in Figure 11.6.

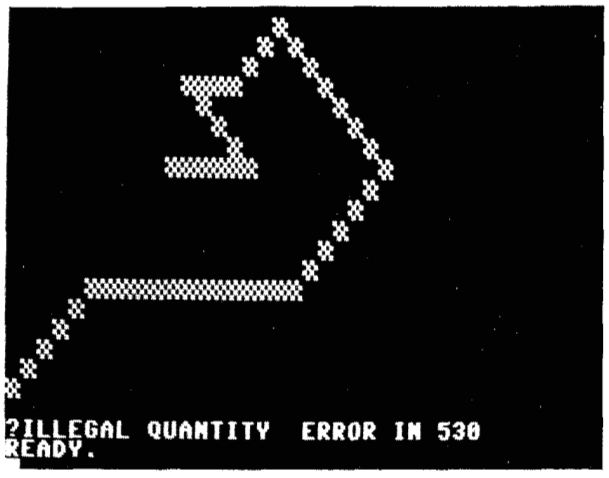
location, and a new graphic symbol is plotted. Note that line 95 in Figure 11.3 had to be deleted and a new **GOTO 30** statement was added as line 190 in Figure 11.6.

A sample run of the program given in Figure 11.6 is shown in Figure 11.7.

Watching the Edge of the Screen

When you run the program shown in Figure 11.6, you will find that if you draw the picture off the left edge of the screen, you will obtain the error message shown in Figure 11.8. Each time you move the cursor left, you subtract one from X. The value of X will be zero at the left edge of the screen. If you try to move one more position to the left, the value of X will become −1. But the argument X in TAB(X) in line 530 must be in the range 0-255. Therefore, when you make X=−1, you obtain the error message **?ILLEGAL QUANTITY ERROR IN 530.**

Figure 11.8 Error message produced because X became negative in TAB(X).



Other strange phenomena will occur if you go off the top, bottom, or right edge of the screen. You should try this. In order to prevent these phenomena from occurring, you can modify the "Move to X,Y" subroutine by adding lines 502-508 shown in Figure 11.9. Figure 11.9(a) shows the modification for the VIC 20 and Figure 11.9(b) shows it for the Commodore 64. Line 502 will replace any negative value of Y with the value of zero. This will prevent the picture from trying to go off the top of the screen. Line 506 will do the same for the value of X. This will prevent the error message shown in Figure 11.8 from occurring. Line 504 will keep the picture from scrolling when you try to move it off the bottom of the screen. Line 508 in Figure 11.9(a) restricts the picture to a maximum of 21 columns, appropriate for the VIC 20. Line 508 in Figure 11.9(b) restricts the picture to a maximum of 39 columns, appropriate for the Commodore 64. (Changing the maximum number of columns to 22 on the VIC 20 or 40 on the Commodore 64 will cause the picture to scroll when the cursor is at the bottom right position of the screen.)

Figure 11.9 Modification (lines 502-508) of program shown in Figure 11.3 to keep picture on the screen. (a) VIC 20. (b) Commodore 64.

(a)

```
2 REM INTERACTIVE DRAWING
5 PRINT"□"
10 X=10:Y=10:B$="▒"
20 GOSUB 500:PRINT B$
30 GET A$:IF A$="" THEN 30
40 IF A$="A" THEN X=X-1:GOTO 20
50 IF A$="D" THEN X=X+1:GOTO 20
60 IF A$="W" THEN Y=Y-1:GOTO 20
70 IF A$="X" THEN Y=Y+1:GOTO 20
80 IF A$="Z" THEN X=X-1:Y=Y+1:GOTO 20
90 IF A$="C" THEN X=X+1:Y=Y+1:GOTO 20
100 IF A$="Q" THEN X=X-1:Y=Y-1:GOTO 20
110 IF A$="E" THEN X=X+1:Y=Y-1:GOTO 20
190 GOTO 30
500 REM  MOVE TO X,Y
502 IF Y<0 THEN Y=0
504 IF Y>21 THEN Y=21
506 IF X<0 THEN X=0
508 IF X>20 THEN X=20
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

(b)

```
2 REM INTERACTIVE DRAWING
5 PRINT"□"
10 X=10:Y=10:B$="▒"
20 GOSUB 500:PRINT B$
30 GET A$:IF A$="" THEN 30
40 IF A$="A" THEN X=X-1:GOTO 20
50 IF A$="D" THEN X=X+1:GOTO 20
60 IF A$="W" THEN Y=Y-1:GOTO 20
70 IF A$="X" THEN Y=Y+1:GOTO 20
80 IF A$="Z" THEN X=X-1:Y=Y+1:GOTO 20
90 IF A$="C" THEN X=X+1:Y=Y+1:GOTO 20
```

(b)
```
100 IF A$="Q" THEN X=X-1:Y=Y-1:GOTO 20
110 IF A$="E" THEN X=X+1:Y=Y-1:GOTO 20
190 GOTO 30
500 REM  MOVE TO X,Y
502 IF Y<0 THEN Y=0
504 IF Y>23 THEN Y=23
506 IF X<0 THEN X=0
508 IF X>38 THEN X=38
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

A sample run of the program given in Figure 11.9 is shown in Figure 11.10. Note that the picture is never allowed to go off the screen. You should add lines 501-508 to your program and try drawing pictures near the edge of the screen.

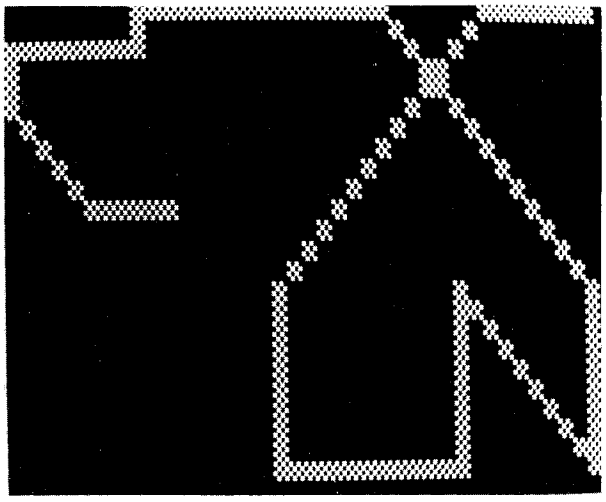


Figure 11.10 The program shown in Figure 11.9 will keep the picture on the screen.

Making a Blinking Cursor

One disadvantage of the program given in Figure 11.9 is that if you backtrack along a line you have already drawn, you cannot easily tell where you are. This deficiency can be corrected by adding a blinking cursor to the program. This can be done by substituting for line 30 the following three lines:

25 FOR I=1 TO 50: NEXT

30 GET A$: IF A$<>""THEN 40

35 GOSUB 500: PRINT " ": FOR I=1 TO 50: NEXT: GOTO 20

The resulting main program is shown in Figure 11.11.

Note that in line 30 A$ is again compared to the null string "", and the program will branch to line 40 if A$ is *not* equal to the null string (that is, if a key has been pressed). If a key has not been pressed, then line 35 will be executed. This line first determines where the cursor is (using **GOSUB 500**) and then prints a *blank* at that position. The FOR...NEXT loop **FOR 1=1 TO 50: NEXT** is used as a *delay*. That is, it uses up the amount of time needed to go through this loop fifty times. If you want a longer delay time (corresponding to a slower blinking cursor), change the 50 to a larger number. For a shorter delay, change the 50 to a smaller number. After this delay, during which a blank is being displayed at the cursor position, the program branches to line 20, where the graphic symbol B$ is again displayed at the cursor position. Line 25 is another delay loop which is used to keep the symbol B$ displayed for a short time. Line 30 then GETs another value for A$ and, if a key has not been pressed, line 35 will display another blank at the cursor position. This loop (lines 20-35) will produce the effect of a blinking cursor.

Figure 11.11 Lines 20-35 produce a blinking cursor.

```
2 REM INTERACTIVE DRAWING
5 PRINT"■"
10 X=10:Y=10:B$="▒"
20 GOSUB 500:PRINT B$
25 FOR I=1 TO 50:NEXT
30 GET A$:IF A$<>"" THEN 40
35 GOSUB 500:PRINT" ":FOR I=1 TO 50:NEXT:GOTO 20
40 IF A$="A" THEN X=X-1:GOTO 20
50 IF A$="D" THEN X=X+1:GOTO 20
60 IF A$="W" THEN Y=Y-1:GOTO 20
70 IF A$="X" THEN Y=Y+1:GOTO 20
80 IF A$="Z" THEN X=X-1:Y=Y+1:GOTO 20
90 IF A$="C" THEN X=X+1:Y=Y+1:GOTO 20
100 IF A$="Q" THEN X=X-1:Y=Y-1:GOTO 20
110 IF A$="E" THEN X=X+1:Y=Y-1:GOTO 20
190 GOTO 30

READY.
```

Make this change and run the modified program. You may be able to observe that the cursor blinks a little faster at the top of the screen than at the bottom of the screen. This is because the time spent in the subroutine "Move to X,Y" (line 500) depends on the value of Y. Larger values of Y, corresponding to the lower part of the screen, will cause a longer time to be spent in subroutine 500 when it is called in lines 20 and 35. Thus, the statement **GOSUB 500** is like a delay that depends on the value of Y. By adding the fixed delay given by **FOR 1=1 TO 50: NEXT** the dependence of the blinking rate on the screen position can be minimized. It will be exaggerated if you change the 50 in the FOR...NEXT delay loop to a smaller number. (Try changing it to 1.) If you change the 50 in the FOR...NEXT loops in lines 25 and 35 to 99, the blinking rate will be reduced and there should be less of a difference in the rate between the top and bottom of the screen. Try it.

Changing a Graphic Symbol

More varied pictures could be drawn if you could interactively change the graphic symbol that is printed in the program in Figure 11.11. Suppose, for example, that when you press the key G (for graphic symbol), the cursor stops blinking and the next key you press will be the new graphic symbol. This modification can be added to the program in Figure 11.11 by deleting line 190 and adding the following four lines:

120 IF A$<>"G" THEN 30
130 GET A$: IF A$="" THEN 130
136 B$=A$
138 GOTO 20

The revised main program listing is shown in Figure 11.12. Line 120 checks to see if the key pressed was G. If it was *not* G, the program branches back to line 30 and waits for another key to be pressed. If it was G, line 130 will be executed. This is just the standard GET loop that waits for a key to be pressed. The next key pressed will be the new graphic symbol. It is assigned to B$ in line 136. The program then branches back to line 20 and displays this new symbol.

Make these changes and run this new program. A sample run is shown in Figure 11.13. You may notice in running this program that the reverse video key appears not to work. That is, you cannot change B$ to a reverse video character. This is because pressing the RVS key after pressing G will assign RVS to A$ and therefore to B$ in line 136. The reverse video will then be turned on when B$ is next printed in line 20. However, the reverse video is turned off by a carriage return. Thus, since the PRINT statement in line 20 does not end with a semicolon, the execution of this PRINT statement will immediately turn off the reverse video. Even if this PRINT statement ended with a semicolon, the reverse video would be turned off by all of the PRINT statements in subroutine 500.

Figure 11.12 Lines 120-140 will change the graphic symbol if key G is pressed.

```
2 REM INTERACTIVE DRAWING
5 PRINT"□"
10 X=10:Y=10:B$="▒"
20 GOSUB 500:PRINT B$
25 FOR I=1 TO 50:NEXT
30 GET A$:IF A$<>"" THEN 40
35 GOSUB 500:PRINT" ":FOR I=1 TO 50:NEXT:GOTO 20
40 IF A$="A" THEN X=X-1:GOTO 20
50 IF A$="D" THEN X=X+1:GOTO 20
60 IF A$="W" THEN Y=Y-1:GOTO 20
70 IF A$="X" THEN Y=Y+1:GOTO 20
80 IF A$="Z" THEN X=X-1:Y=Y+1:GOTO 20
90 IF A$="C" THEN X=X+1:Y=Y+1:GOTO 20
100 IF A$="Q" THEN X=X-1:Y=Y-1:GOTO 20
110 IF A$="E" THEN X=X+1:Y=Y-1:GOTO 20
120 IF A$<>"G" THEN 30
130 GET A$:IF A$="" THEN 130
136 B$=A$
138 GOTO 20

READY.
```

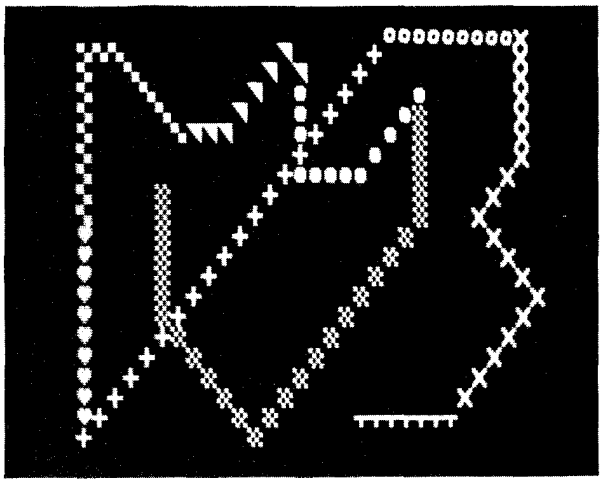


Figure 11.13 Sample run of program shown in Figure 11.12 illustrating changed graphic symbols.

In order to add the reverse video capability to the program in Figure 11.12, it is necessary to turn on (and off) the reverse video each time B$ is printed by line 20. This can be done by adding the following lines:

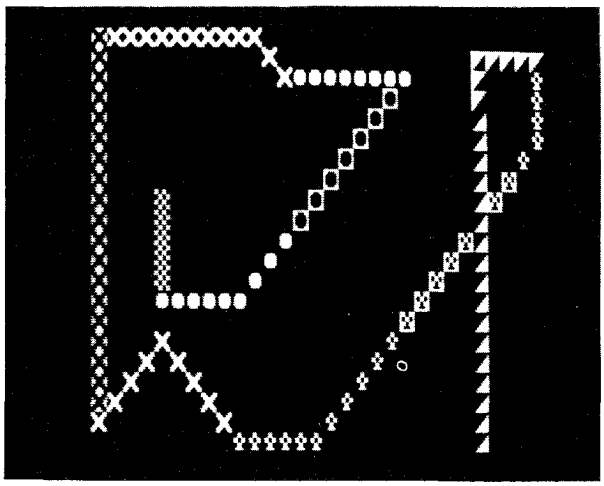
20 GOSUB 500: IF RV=0 THEN PRINT B$:GOTO 25
21 PRINT "RVS" + B$ + "OFF"
132 IF A$ = "RVS" THEN RV=1:GOTO 138
134 IF A$ = "OFF" THEN RV=0:GOTO 138

Figure 11.14 Lines 10-21 and 132-138 will allow the graphic symbol to be changed to a reverse video character.

```
2 REM INTERACTIVE DRAWING
5 PRINT"■"
10 X=10:Y=10:B$="▒"
20 GOSUB 500:IF RV=0 THEN PRINT B$:GOTO 25
21 PRINT "■"+B$+"■"
25 FOR I=1 TO 50:NEXT
30 GET A$:IF A$<>"" THEN 40
35 GOSUB 500:PRINT" ":FOR I=1 TO 50:NEXT:GOTO 20
40 IF A$="A" THEN X=X-1:GOTO 20
50 IF A$="D" THEN X=X+1:GOTO 20
60 IF A$="W" THEN Y=Y-1:GOTO 20
70 IF A$="X" THEN Y=Y+1:GOTO 20
80 IF A$="Z" THEN X=X-1:Y=Y+1:GOTO 20
90 IF A$="C" THEN X=X+1:Y=Y+1:GOTO 20
100 IF A$="Q" THEN X=X-1:Y=Y-1:GOTO 20
110 IF A$="E" THEN X=X+1:Y=Y-1:GOTO 20
120 IF A$<>"G" THEN 30
130 GET A$:IF A$="" THEN 130
132 IF A$="■" THEN RV=1:GOTO 138
134 IF A$="■" THEN RV=0:GOTO 138
136 B$=A$
138 GOTO 20

READY.
```

The resulting program is shown in Figure 11.14. If the key pressed after pressing G is the RVS key, line 132 will set the reverse video flag RV to 1. If the key pressed after pressing G is the OFF key, line 134 will set the flag RV to 0. Anytime a new graphic symbol is typed after pressing G, line 136 will change B$ to the symbol A$ entered. Now, in line 20, after setting the cursor, a check is made as to whether the reverse video flag RV is 0 or 1. If it is 0, B$ is printed as usual. If RV is 1 then "RVS" + B$ + "OFF" is printed by 21. Thus, the character printed is reverse video.

Figure 11.15 Sample run of program shown in Figure 11.14 illustrating changed reverse video graphic symbols.



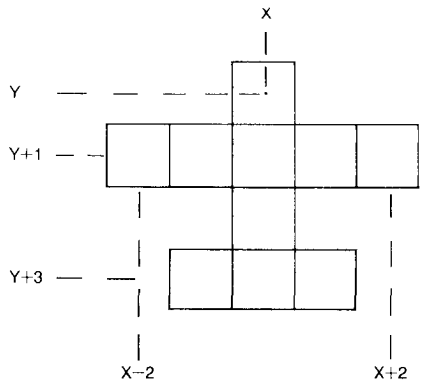
Make the changes shown in Figure 11.14 and run the new program. A sample run is shown in Figure 11.15.

FIGHTER PLANE WITH PHASERS

In the interactive drawing program just described, all plotted points remain on the screen. Sometimes it is desirable to *move* a point (or points) around the screen. In this case old points must be erased as new points are plotted.

As an example of doing this, suppose you want to draw the plane shown in Figure 11.16 and then move it around the screen using the A, D, W, and X keys. To do this you will need to be able to *draw* the plane at any location X,Y and also be able to *erase* the plane at any

Figure 11.16 Geometry of plane to be drawn on screen.



location. Subroutines for doing this are shown in Figure 11.17a.

The subroutines in Figures 11.17b and 11.17c are our usual "Move to X,Y" subroutines with the screen limits changed to keep the wings and tail from going off the screen. Line 110 in Figure 11.17a draws the plane at X,Y and line 160 erases the plane at X,Y. In order to test these subroutines, type in the statements shown in Figure 11.17, and then clear the screen and type these statements in the immediate mode: **X=15** (**10** on the VIC 20)**: Y=5: GOSUB 100.**

You should obtain the result shown in Figure 11.18. In order to test the erasing subroutine, move the cursor to the top of the screen and edit the statements in Figure 11.18 by changing **GOSUB 100** to **GOSUB 150.** When you press RETURN the plane should disappear. Try it.

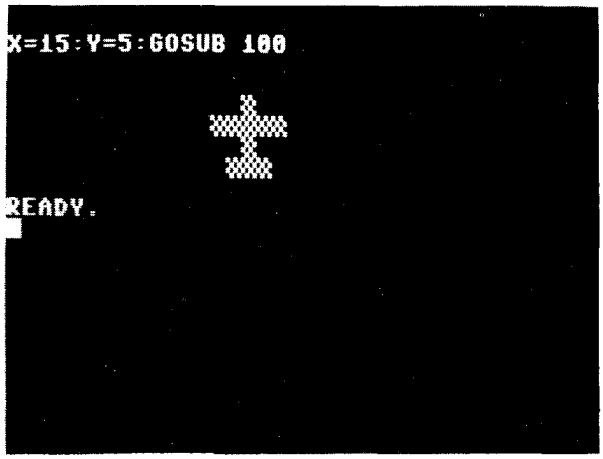


Figure 11.18 Testing subroutine to draw plane.

Main Program with Stubs

The next step is to write a main program that will move the plane left, right, up, and down when you press keys A, D, W, and X, respectively. In addition, we want to fire a phaser from the left wing when key L is pressed and from the right wing when key R is pressed. Even though we have not learned how to do this yet, we can write the complete main program as shown in Figure 11.19. Line 10 clears the screen and sets the initial plane location. Line 20 draws the plane at this initial X,Y location. Line 30 is the standard GET loop that waits for a key to be pressed. Lines 40-70 check to see if the key pressed is a A, D, W, or X. If it is any of these keys, the plane is erased and then redrawn at the appropriate new position.

Figure 11.17 (a) Subroutines to draw and erase the plane. (b) "Move to X,Y" subroutine for the VIC 20. (c) "Move to X,Y" subroutine for the Commodore 64.

(a)
```
100 REM  DRAW PLANE
110 GOSUB 600:PRINT"                    ":RETURN
150 REM  ERASE PLANE
160 GOSUB 600:PRINT"                    ":RETURN

READY.
```

(b)
```
600 REM  MOVE TO X,Y
602 IF Y<0 THEN Y=0
604 IF Y>18 THEN Y=18
606 IF X<2 THEN X=2
608 IF X>18 THEN X=18
610 PRINT"■";:IF Y=0 THEN 630
620 FOR I=1 TO Y:PRINT:NEXT
630 PRINT TAB(X);:RETURN

READY.
```

(c)
```
600 REM  MOVE TO X,Y
602 IF Y<0 THEN Y=0
604 IF Y>20 THEN Y=20
606 IF X<2 THEN X=2
608 IF X>36 THEN X=36
610 PRINT"■";:IF Y=0 THEN 630
620 FOR I=1 TO Y:PRINT:NEXT
630 PRINT TAB(X);:RETURN

READY.
```

Lines 80-90 check to see if the key pressed is an L or an R. If it is an L then the program branches to the subroutine at line 200; if it is an R then the program branches to the subroutine at line 300. When they are written, these subroutines will fire the phasers. For now, just type

200 RETURN
300 RETURN

These are called *stubs,* and we will add phaser-firing subroutines here later. You know that you want to call these subroutines when you press keys L and R. Therefore, you can complete the main program as shown in Figure 11.19 without having written the phaser-firing subroutines at lines 200 and 300. Of course, if you run the program and press key L (or R) nothing will happen, because the program will immediately RETURN and then go back to line 30. However, keys A, D, W, and X should behave properly. Type in the main program shown in Figure 11.19 and run it. Does it behave as you expected?

Phaser Subroutines

Now that the main program is working, you can concentrate on writing the phaser subroutines. There are many different ways to make phasers. One possibility is shown in Figure 11.20. Lines 200-250 fire the left phaser. Lines 300-350 fire the right phaser. The subroutine in lines 400-440 erases either phaser.

Lines 500-530 contain yet another copy of the "Move to" subroutine. However, this time we make it "Move to U,V". We can't use the existing "Move to X,Y" subroutine because (1) using it to fire phasers will require us to alter X and Y from the plane coordinates and (2) the existing subroutine contains statements in lines 602-608 that won't allow X and Y to reach the edges of the screen. In the case of the phasers, we have to be able to fire phasers right up to the edges of the screen. Hence, we introduce U and V as new coordinate variables for that purpose and a new copy of the "Move to" subroutine just for them.

The FOR...NEXT loop in lines 220-240 plots a string of dots from the tip of the left wing to the top of the screen. Line 250 then erases this string (using the subroutine at line 400). This is one phaser firing. The program then RETURNs to the main program (in line 80) and checks for another key-pressing. The loop in lines 320-340 plots a similar string of circles from the tip of the right wing to the top of the screen, and line 350 erases this string. The strings of dots and circles are erased by plotting a string of blanks over them, using the loop in lines 410-430.

Add the subroutines shown in Figure 11.20 to the rest of the program (given in Figures 11.19 and 11.17), and run the program. Pressing keys L and R should produce the phasers shown in Figure 11.21.

Figure 11.19 Main program for controlling a fighter plane with phasers.

```
5 REM FIGHTER PLANE WITH PHASERS
10 PRINT"■":X=10:Y=10
20 GOSUB 600:GOSUB 100
30 GET A$:IF A$="" THEN 30
40 IF A$="A" THEN GOSUB 150:X=X-1:GOSUB 100:GOTO 30
50 IF A$="D" THEN GOSUB 150:X=X+1:GOSUB 100:GOTO 30
60 IF A$="W" THEN GOSUB 150:Y=Y-1:GOSUB 100:GOTO 30
70 IF A$="X" THEN GOSUB 150:Y=Y+1:GOSUB 100:GOTO 30
80 IF A$="L" THEN GOSUB 200:GOTO 30
90 IF A$="R" THEN GOSUB 300:GOTO 30
95 GOTO30

READY.
```

Figure 11.20 Subroutines for firing phasers.

```
200 REM FIRE LEFT PHASER
210 U=X-2
220 FOR V=Y TO 0 STEP -1
230 GOSUB 500:PRINT"●"
240 NEXT
250 GOSUB 400:RETURN
300 REM  FIRE RIGHT PHASER
310 U=X+2
320 FOR V=Y TO 0 STEP -1
330 GOSUB 500:PRINT"o"
340 NEXT
350 GOSUB 400:RETURN
400 REM  ERASE PHASER
410 FOR V=Y TO 0 STEP -1
420 GOSUB 500:PRINT" "
430 NEXT
440 RETURN
500 REM  MOVE TO U,V
510 PRINT"■";:IF V=0 THEN 530
520 FOR I=1 TO V:PRINT:NEXT
530 PRINT TAB(U);:RETURN

READY.
```

(a)

(b)

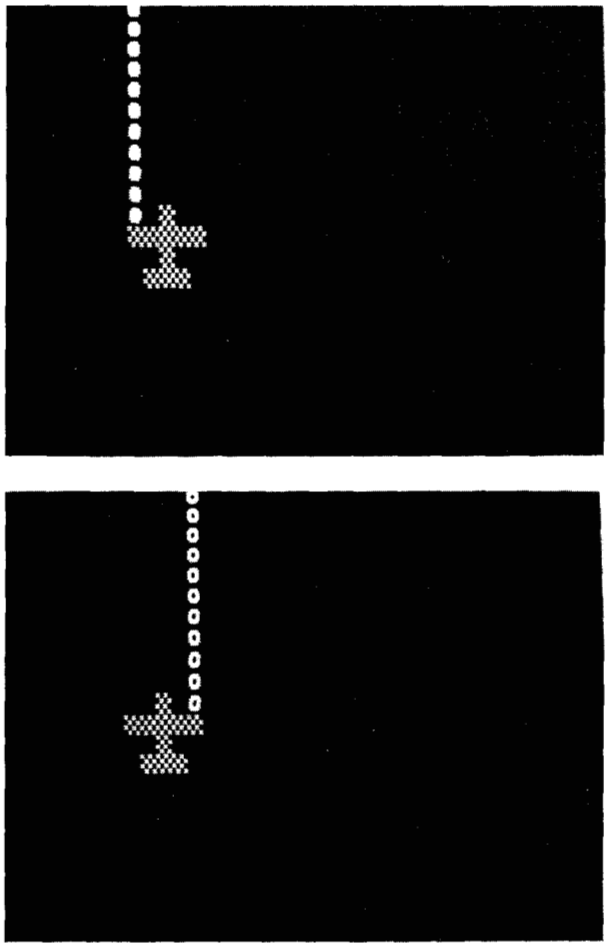


Figure 11.21 (a) Phaser produced by pressing key L. (b) Phaser produced by pressing key R.

Keyboard Buffer

Run the "plane with phasers" program, and press key L followed quickly by key R. The left phaser should be fired, and then the right phaser should be fired. Although the left phaser had not finished firing when you pressed the R the Commodore 64/VIC 20 remembers that the R has been pressed and fires the right phaser after finishing the left phaser firing.

Press the keys L, R, L, R in order *quickly*. Observe that the Commodore 64/VIC 20 remembers all four key pressings.

Press the key combination L, R quickly *five* times (a total of ten key presses). Note that the Commodore 64/VIC 20 remembers all of these and fires a total of ten phasers.

Now press the key combination L, R quickly *six* times (a total of twelve key presses). Do this fast enough that all twelve keys are pressed before the left phaser has finished its *first* firing. You can move the plane to the bottom part of the screen to give yourself more time. If you press both keys six times before the left phaser has finished its first firing, you will find that the Commodore 64/VIC 20 appears to have forgotten your last key press.

The reason for this apparently strange behavior has to do with the keyboard buffer that was described at the beginning of this chapter. This buffer contains ten memory locations which are used to temporarily store the characters corresponding to the pressed keys until the GET statement can remove them from the buffer. When we enter L, R six times very rapidly (while the first firing is occurring in response to the first L) there are 11 key presses to store before a GET occurs. Thus, the last R key press is lost.

EXERCISE 11-1
Modify the program shown in Figure 7.4 by substituting a PRINT statement and a GET statement for line number 120. The modified program should continue as soon as the user presses key Y.

EXERCISE 11-2
Modify the interactive drawing program of Figure 11.14 so that the cursor may be changed in color from the keyboard.

EXERCISE 11-3
Modify the phaser-firing subroutines shown in Figure 11.20 so that only a single "bullet" is fired from each wing.

EXERCISE 11-4
Write a program that will move a graphic symbol left, right, up, or down using keys A, D, W, and X as shown in Figure 11.2. However, instead of moving the graphic symbol only one space for each key pressed, let the graphic symbol continue to move in the same direction, tracing out a line, until another key is pressed to change its direction.

EXERCISE 11-5
Modify the program in Exercise 11-4 so that the graphic symbol moves across the screen without tracing out a line. The graphic symbol should change direction when you press keys A, D, W, and X.

EXERCISE 11-6
Modify the program with the "plane and phasers" so the plane is purple, the left phaser is yellow, and the right phaser is green.

12 LEARNING TO USE ARRAYS

You have learned that numerical values are stored in memory cells with names like A and B3. Similarly, strings are stored in memory cells with names like A$ and B3$. Sometimes it is desirable to identify a collection of memory cells by one name. Such a collection of memory cells is called an *array,* and the individual memory cells within the array are identified by means of *subscripts.*

In this chapter you will learn:

1. how to represent arrays in BASIC
2. the difference between one-dimensional and multi-dimensional arrays
3. how to use the DIM statement
4. how to use arrays when plotting bar graphs
5. how to sort data stored in a one-dimensional array
6. how to use the ON...GOSUB statement
7. how to write an interactive program to display three-dimensional letters.

ARRAYS

You will often encounter data that are related in some way. For example, Table 10.1 in Chapter 10 lists the six New England states and their populations. In the program in Figure 10.6 we read each state name into the memory cell S$ and each population figure into the memory cell P. Only one state name and one population figure were in these memory cells at any one time. We printed the state name and plotted a bar with a length proportional to the population. Then we read another state name and population figure into S$ and P.

Some programs, however, would require that all of the state names and populations be stored in different memory cells at the same time. We would therefore need twelve memory cells, six for the state names and six for the populations. It is convenient to store all the state names in an *array* called S$ and all the populations in an *array* called P. The individual memory cells within the array are distinguished by a subscript I. An individual element within the array is sometimes called a *subscripted variable* for example, P(I) or S$(I). The arrays S$ and P are shown in Figure 12.1.

Note that in BASIC, subscripts are enclosed in parentheses. The variable name P(2), for example, is the name of the memory cell that contains the value 511456. You can use these subscripted variables in the same way as simple variable names. For example, type these statements as shown in Figure 12.2:

```
S$(2)="VT"
?S$(2)
P(2)=511456
?P(2)
```

Figure 12.1 (a) The six subscripted variables S$(I), I=0,5 contain the state names. (b) The six subscripted variables P(I), I=0,5 contain the state population figures.

S$(0)	ME
S$(1)	NH
S$(2)	VT
S$(3)	MA
S$(4)	CT
S$(5)	RI

(a)

P(0)	993663
P(1)	737681
P(2)	444732
P(3)	5689170
P(4)	3032217
P(5)	949723

(b)

```
S$(2)="VT"

READY.
?S$(2)
VT

READY.
P(2)=511456

READY.
?P(2)
 511456

READY.
```

Figure 12.2 Subscripted variables can be used just like simple variables.

Although the memory cells S$(2) and P(2) are elements of an array, individually they can be treated like any other memory cell names.

Having a memory cell name contain a subscript, however, can be very useful. For example, type the following two statements as shown in Figure 12.3:

FOR I=0 TO 10: A(I)=2*I: NEXT
FOR I=0 TO 10: ?A(I);: NEXT

The first statement fills each of the eleven memory cells A(I), I=0, 10 with the value 2*I. The second statement prints these eleven values.

Try changing the first statement to **FOR I=0 TO 11: A(I)=2 *I: NEXT.** You will obtain the error message **?BAD SUBSCRIPT ERROR.** The reason for this error will be explained in the next section.

The DIM Statement

Whenever a BASIC program first encounters a subscripted variable, such as A(3), it automatically assigns eleven memory locations to the array. These memory locations are assigned the names A(0)-A(10).

```
FOR I=0 TO 10:A(I)=2*I:NEXT

READY.
FOR I=0 TO 10:?A(I);:NEXT
 0  2  4  6  8  10  12  14  16  18  20
READY.
```

Figure 12.3 Filling the array A(I) with the value 2*I.

If you try to use the name A(11), as in the example just given, you get the error message **?BAD SUBSCRIPT ERROR** because this name is not reserved in the computer.

If you want to use an array with more than eleven memory cells, you must explicitly define the array with a DIM (for DIMension) statement. For example, to assign sixteen memory cells to the array B you would type **DIM B(15).** You could then use the sixteen memory cells B(0)-B(15). The constant 15 in this DIM statement defines the upper subscript limit of the array. (The upper limit can also be a variable or an expression.) The lower subscript limit is always assumed to be zero.

You can define more than one array with a single DIM statement. For example, you can write **DIM B(15),A(3),C(24)** to define three arrays containing sixteen, four, and twenty-five memory cells, respectively.

Although it is not necessary to use a DIM statement when the array contains eleven or fewer memory cells, it is usually a good idea to use one. The DIM statement gives you a convenient place to define what the array means (by following the DIM statement with a REM statement). In addition, if you need fewer than eleven memory cells in an array, the DIM statement will save memory by reserving only the number you define. This saving is bigger than it may appear, because the Commodore 64/VIC 20 uses five bytes of memory for *each* numerical memory cell.

There is another way to save a considerable amount of memory when using a large array that contains only *integers* (whole numbers). If you add a percent sign, %, to an array name, the Commodore 64/VIC 20 will consider all array elements to be integers and will use only *two* bytes of memory (rather than five) to store each integer element. For example **DIM C%(99)** would define an integer array containing 100 elements and would save 300 bytes of memory when executing the program. You would, of course, use the sub-

scripted variable **C%(I)** in the program. Reserving more memory locations than you use in the program will not cause problems; it will simply use extra memory.

The DIM statement may occur anywhere in the program, but it must occur *before* you refer to the corresponding subscripted variable. An array can be *dimensioned* only once in a program. If you try to dimension it more than once you will obtain the error message **?REDIM'D ARRAY ERROR.** Remember that if a subscripted variable such as A(3) is encountered *before* a DIM statement has been executed, the Commodore 64/VIC 20 will automatically dimension the array equivalent to **DIM A(10).** If it then encounters a DIM statement for A, the above error message will be displayed.

For example, try typing **FOR I=0 TO 11: A(I)=2*I: NEXT.** If you have not dimensioned the array A, you will obtain the error message **?BAD SUBSCRIPT ERROR** as shown in Figure 12.4. The problem, of course, is that you tried to refer to A(11), and the Commodore 64/VIC 20 has automatically dimensioned the array as A(10). But if you try to correct this by typing **DIM A(11)** you will obtain the error message **?REDIM'D ARRAY ERROR** as shown in Figure 12.4, because the array is already dimensioned. To correct this, type **CLR** as shown in Figure 12.4. This *clears,* or reinitializes, the system variables and has the effect of wiping out any dimensioning information. You can then type the following statements without errors, as shown in Figure 12.4:

DIM A(11)
FOR I=0 TO 11: A(I)=2*I: NEXT
FOR I=0 TO 11: ?A(I);: NEXT

Figure 12.4 Examples of using the DIM and CLR statements.

```
FOR I=0 TO 11: A(I)=2*I: NEXT

?BAD SUBSCRIPT  ERROR
READY.
DIM A(11)

?REDIM'D ARRAY  ERROR
READY.
CLR

READY.
DIM A(11)

READY.
FOR I=0 TO 11: A(I)=2*I: NEXT

READY.
FOR I=0 TO 11:?A(I);: NEXT
 0   2   4   6   8   10  12  14  16  18  20
22
READY.
```

The maximum number of elements in an array will be limited by the amount of memory in your Commodore 64/VIC 20. If the total amount of memory used by your program, variables, and arrays exceeds the amount of memory in your Commodore 64/VIC 20, you will obtain the error message **?OUT OF MEMORY ERROR.**

Any time you want to know how many bytes of free memory you have left, type **?FRE(1).** (Numbers over 32768, which are possible on the Commodore 64, appear as negative values. To get the corresponding positive value, add 65536.) For example, Figure 12.5 shows that an array containing 100 elements uses 507 bytes of memory (five for each of the 100 elements in the array and seven for an array header). Before the DIM A(99) statement, there are 65536−26627= 38909 bytes free and after there are 65536−27134= 38402 bytes free so 38909−38402=507 bytes have been used up.

```
? FRE(1)
-26627

READY.
DIM A(99)

READY.
? FRE(1)
-27134

READY.
```

Figure 12.5 The statement ?FRE(1) will print the number of free bytes of memory left.

Multi-dimensional Arrays

An array that contains a single subscript is called a one-dimensional array. An array that contains more than one subscript is called a multi-dimensional array. For example, the DIM statement **DIM A(2,3)** defines a two-dimensional array containing twelve elements. It can be thought of as a two-dimensional *matrix* containing three rows and four columns as shown in Figure 12.6. In the array A(I,J), the first subscript, I, is the *row* number and the second subscript, J, is the column number. Thus, for example, in Figure 12.6 the value of A(1,2) is 8 and the value of A(2,1) is 12.

A three dimensional array containing a total of 7×9×3 = 189 elements can be defined by the DIM statement **DIM F(6,8,2).** Examples of using two- and three-dimensional arrays in a program will be given in

Chapters 13 and 14. In the remainder of this chapter we will write some programs using one-dimensional arrays.

Figure 12.6 The array A(I,J) containing twelve elements.

	J			
	0	1	2	3
0	11	7	0	13
1	3	15	8	4
2	5	12	9	1

BAR GRAPHS USING ARRAYS

The program shown in Figure 10.5 plots four bars of lengths 12, 21, 5, and 17. Review that program and make sure you understand how it works. Line 400 plots a bar of length L using the graphic symbol G$. In this section we will modify this program to plot bars of the same lengths, using a different graphic symbol for each bar.

The modified program and its execution are shown in Figure 12.7. Line 15 is a new DATA statement that contains the four graphic symbols that will be used for the four bars. Line 25 is the DIM statement **25 DIM G$(4),L(4).** This statement defines an array G$(I) that will contain the four graphic symbols and an array L(I) that will contain the four lengths. Although this DIM statement defines *five* elements for each array, we will use only the elements G$(1)-G$(4) and L(1)-L(4) and ignore G$(0) and L(0).

Figure 12.7 Bar graph example using arrays.

```
10 REM  BAR GRAPH EXAMPLE
15 DATA "%","●","o","■"
20 DATA 12,21,5,17
25 DIM G$(4),L(4)
30 FOR I=1 TO 4:READ G$(I):NEXT
40 FOR I=1 TO 4:READ L(I):NEXT
50 FOR J=1 TO 4
60 L=L(J):G$=G$(J):GOSUB 400
70 NEXTJ
90 END
400 FOR I=1 TO L:PRINT G$;:NEXT:PRINT:PR
INT:RETURN
READY.
RUN
%%%%%%%%%%%%

●●●●●●●●●●●●●●●●●●●●

ooooo

READY.
```

Line 30 reads the four graphic symbols in the DATA statement on line 15 into the four subscripted variables G$(1)-G$(4). Line 40 reads the four values 12, 21, 5, and 17 from line 20 into the four subscripted variables L(1)-L(4) respectively.

The loop defined by lines 50-70 plots the four bars. Each time through the loop, a new length L(J) and a new graphic symbol G$(J) are assigned to L and G$ to be plotted by line 400. Note how the subscript J is incremented from 1 to 4 by successive passes through the loop. Also note that the subscripted variables L(J) and G$(J) and the simple variables L and G$ are not confused by the Commodore 64/VIC 20 and are treated as separate memory cells.

The four bars in Figure 12.7 can be plotted adjacent to each other by eliminating one of the PRINT statements in line 400, as shown in Figure 12.8.

Suppose you would like to plot the bars shown in Figure 12.8 in order of *increasing* length—that is, the shortest bar first, the next-to-shortest second, and so on. You can do this if you rearrange the array L(I) so that the elements are in increasing, or *ascending,* order. One simple method of sorting an array in ascending order will now be described.

Figure 12.8 Plotting the bars adjacent to each other.

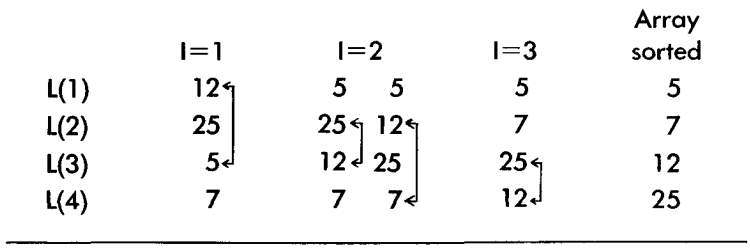
```
LIST
10 REM  BAR GRAPH EXAMPLE
15 DATA "%","●","o","■"
20 DATA 12,21,5,17
25 DIM G$(4),L(4)
30 FOR I=1 TO 4:READ G$(I):NEXT
40 FOR I=1 TO 4:READ L(I):NEXT
50 FOR J=1 TO 4
60 L=L(J):G$=G$(J):GOSUB 400
70 NEXTJ
90 END
400 FOR I=1 TO L:PRINT G$;:NEXT:PRINT:RE
TURN
READY.
RUN
%%%%%%%%%%%%
●●●●●●●●●●●●●●●●●●●●
ooooo

READY.
```

Sorting an Array in Ascending Order

Many algorithms have been devised for sorting an array of elements in ascending order. Some are more efficient than others, that is, they are executed faster. Some (not necessarily the same ones) are easier to understand than others. The method of sorting the array L illustrated in Figure 12.9 is fairly easy to understand (but not very efficient).

The method begins by comparing the first element in the array (I=1) with all succeeding elements.

Figure 12.9 Sorting an array by moving the smallest succeeding value to location I, I=1 to N − 1 (N= number of elements in array).



Whenever a succeeding element is found to be smaller than the first element, it is interchanged with the first element. Thus, after the first element (whose value may have changed a few times) is compared with *all* succeeding values, we will have moved the *smallest* value to the first position in the array.

If we repeat this procedure starting with the *second* element (I=2), then after comparing the second element with all succeeding elements and interchanging the values if any succeeding element is smaller than the second one, the next-to-smallest value will be in the second position in the array.

This process continues until we have compared the next-to-last element with the last element in the array. At this point the array is sorted in ascending order as shown in Figure 12.9. The algorithm for this procedure is shown in pseudocode in Figure 12.10. Compare Figures 12.9 and 12.10 and make sure you understand how this sorting algorithm works.

The algorithm shown in Figure 12.10 looks fairly easy to write in BASIC. The only problem is how to interchange the contents of L(I) and L(J). Note that the two statements

L(I)=L(J)
L(J)=L(I)

will not work because the original value in L(I) will be destroyed when the value of L(J) is put in L(I) in the first statement. This means that the second statement will really be assigning the value in L(J) to itself, and L(I) and L(J) will have the same value. It requires *three* statements to interchange the values of L(I) and L(J), as shown in Figure 12.11.

Figure 12.10 Pseudocode representation of sorting algorithm shown in Figure 12.9.

```
for I=1 TO N-1
    for J=I+1 TO N
        if L(I)<=L(J)
        then do nothing
        else interchange L(I) and L(J)
    nextJ
nextI
```

Figure 12.11 Three statements are required to interchange L(1) and (J).

L(I)=L(J)
L(J)=L(I)
} will *not* interchange L(I) and L(J)

T=L(I)
L(I)=L(J)
L(J)=T
} *will* interchange L(I) and L(J)

The value in L(I) must be stored temporarily in another memory cell T before the value in L(J) is put in L(I). Then the value in T (formerly in L(I)) can be put in L(J).

The sorting algorithm shown in Figure 12.10 is written as a BASIC subroutine in Figure 12.12. Line 2040 interchanges the values in L(I) and L(J). Add this subroutine to the program shown in Figure 12.8.

Figure 12.12 BASIC subroutine to sort array L(I) containing N elements in ascending order.

```
2000 REM  SORT L(I)
2010 FOR I=1 TO N-1
2020 FOR J=I+1 TO N
2030 IF L(I)<=L(J) THEN 2050
2040 T=L(I):L(I)=L(J):L(J)=T
2050 NEXTJ:NEXTI
2060 RETURN

READY.
```

Then add the statement **45 N=4: GOSUB 2000: REM SORT L(I)** to the main program as shown in Figure 12.13. The result of running this program is also shown in Figure 12.13. Line 45 sets the number of elements in the array L to 4 and then sorts this array by calling the subroutine shown in Figure 12.12.

If you compare Figure 12.13 with Figure 12.8 you will notice that the four graphic symbols are plotted in the same relative order. That is, they did not get sorted as the data did. However, it probably makes more

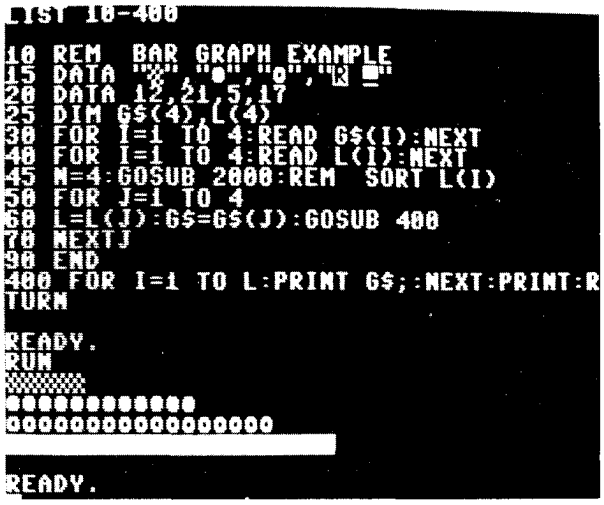


Figure 12.13 Plotting bar graphs in ascending order using the subroutine in Figure 12.11.

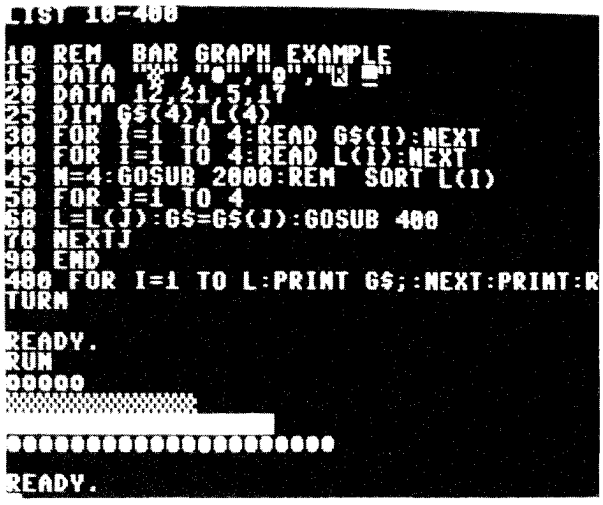


Figure 12.15 Plotting bar graphs using the subroutine in Figure 12.14.

sense to associate a particular graphic symbol with a particular data value (such as inflation, growth, etc.) so that if the data is rearranged (sorted), the corresponding graphic symbols will also be rearranged. We can do this by adding the statement **2045 T$=G$(I): G$(I)=G$(J): G$(J)=T$** to the subroutine given in Figure 12.12 as shown in Figure 12.14. This statement will cause G$(I) and G$(J) to be interchanged each time L(I) and L(J) are interchanged. This will cause a given data value to "keep" its particular graphic symbol as shown in Figure 12.15.

Figure 12.14 Sorting subroutine that interchanges G$(I) and G$(J) each time L(I) and L(J) are interchanged.

```
2000 REM  SORT L(I)
2010 FOR I=1 TO N-1
2020 FOR J=I+1 TO N
2030 IF L(I)<=L(J) THEN 2050
2040 T=L(I):L(I)=L(J):L(J)=T
2045 T$=G$(I):G$(I)=G$(J):G$(J)=T$
2050 NEXTJ:NEXTI
2060 RETURN

READY.
```

Sorting an Array in Descending Order

The subroutine in Figure 12.14 can easily be modified to sort the array L(I) in *descending* order rather than ascending order by changing line 2030 to **2030 IF L(I)>=L(J) THEN 2050** as shown in Figure 12.16. Running the main program with this subroutine will produce the result shown in Figure 12.17.

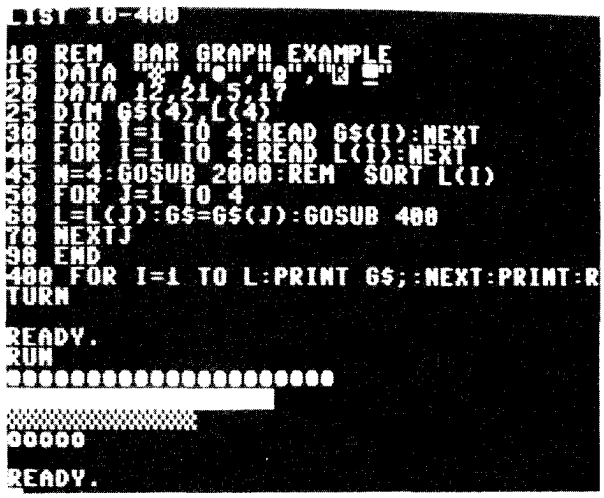
THE ON...GOSUB STATEMENT

Sometimes it is convenient to be able to branch to one of several possible subroutines, depending upon the

Figure 12.16 Subroutine to sort array L(I) containing N elements in descending order.

```
2000 REM  SORT L(I)
2010 FOR I=1 TO N-1
2020 FOR J=I+1 TO N
2030 IF L(I)>=L(J) THEN 2050
2040 T=L(I):L(I)=L(J):L(J)=T
2045 T$=G$(I):G$(I)=G$(J):G$(J)=T$
2050 NEXTJ:NEXTI
2060 RETURN

READY.
```

Figure 12.17 Plotting bar graphs in descending order using the subroutine in Figure 12.16.



value of some index variable I. This can be done by using the ON...GOSUB statement. As an example, if the statement **30 ON I GOSUB 100,200,300** is executed, then if I=1, the program branches to the subroutine at line 100; if I=2, the program branches to the subroutine at line 200; if I=3, the pro-

gram branches to the subroutine at line 300; if I is any other value (it must be between 0 and 255), then no branch occurs and the statement following on the ON...GOSUB statement is executed. Each subroutine must end with a RETURN statement. When one of these RETURN statements is executed, the program will branch back to the statement following the ON...GOSUB statement.

The index I in the ON...GOSUB statement just shown can be any variable or expression with a value in the range 0-255. There can be any number of subroutine line numbers following GOSUB, as long as the entire statement fits on four screen lines on the VIC 20 or two screen lines on the Commodore 64. The ON...GOSUB statement branches to the subroutine at the line number whose position following the word GOSUB corresponds to the value of the index I. The use of the ON...GOSUB statement is illustrated in the next example.

Displaying 3-D Letters

In Chapter 9 you learned how to make 3-D letters. The main program shown in Figure 9.27 displays the 3-D name JEFF. It uses the graphic primitives shown in Figure 9.15 and the subroutines shown in Figures 9.18, 9.21, 9.24, and 9.26. For the program in this section we will add the two subroutines shown in Figures 12.18 and 12.19 for displaying a 3-D P and a 3-D T, respectively (see Exercise 9-2).

We will rewrite the main program starting at line 1000 to do the following:

1. Display the message **ENTER J, F, E, P, OR T:**,
2. Display the 3-D letter corresponding to the key pressed,
3. Display the message **DO YOU WISH TO ENTER ANOTHER LETTER?** and repeat steps 1-3 if key Y is pressed.

Figure 12.18 Subroutine to display 3-D P.

```
600 REM  3-D P
610 GOSUB 200:PRINT "TTTTTTTT■■■ ■";B2$;:GOSUB 220
620 RETURN

READY.
```

Figure 12.19 Subroutine to display 3-D T.

```
700 REM  E-D T
710 PRINT D5$;B5$;C4$;B5$;E4$;B4$;
720 FOR I=1 TO 7:PRINT A1$;C1$;B2$;"■■";:NEXT:PRINT A1$;E1$;
730 RETURN

READY.
```

The basic strategy of the program will be to use the ON...GOSUB statement. The subroutines to display the 3-D letters are located at the following line numbers:

Line Number	*Subroutine*	*Shown in Figure*
100	Display 3-D J	9.18
200	Display 3-D F	9.21
300	Display 3-D E	9.24
600	Display 3-D P	12.18
700	Display 3-D T	12.19

Therefore, the statement **ON I GOSUB 100,200,300, 600,700** will cause a J, F, E, P, or T to be displayed depending upon whether I is equal to 1, 2, 3, 4, or 5. We need some way of making the value of I (1-5) correspond to the key pressed (J, F, E, P, or T) whose value is stored in a string variable A$.

To accomplish this we will store all possible letters in an array B$(I) using the two statements:

1020 DATA J, F, E, P, T
1030 FOR I=1 TO 5: READ B$(I): NEXT

This will result in the array shown in Figure 12.20. Note that the value of the subscript I is the value needed in the ON...GOSUB statement to cause a branch to the proper subroutine (for example, I=4 corresponds to P and will cause a branch to line 600).

The next problem is how to determine which index value I corresponds to the letter stored in A$ (the key pressed). This can be done by comparing A$ with B$(I) as I increases from 1 to 5. When a match occurs, the

Figure 12.20 The value of the subscript 1 identifies a particular letter.

	B$(I)	*I*
B$(1)	J	1
B$(2)	F	2
B$(3)	E	3
B$(4)	P	4
B$(5)	T	5

value of I is the one to use in the ON...GOSUB statement. In pseudocode the match can be found as follows:

```
I=1
do until A$=B$(I) or I>5
    I=I+1
enddo
```

In other words we simply increment I by one until there is a match between A$ and B$(I) or until we have gone through the entire list without a match. (This would occur if you pressed a key other than J, F, E, P, or T.) This algorithm can be written in BASIC as follows:

```
1050 I=1
1060 IF A$=B$(I) OR I>5 THEN 1075
1070 I=I+1: GOTO 1060
1075 -------
```

Note that for this algorithm to work, the array B$(I) must be dimensioned to at least B$(6) because I will equal 6 if no match is found.

The entire main program for displaying a 3-D letter is shown in Figure 12.21. Line 1005 clears the screen. Line 1010 dimensions the array B$(I). Lines 1020-1030 fill the array B$(I) as shown in Figure 12.20. Line 1040 displays the message **ENTER J, F, E, P, OR T:**. Line 1045 branches to a subroutine at line 800 that will produce a blinking cursor and wait for a letter to be entered. This subroutine uses the GET statement and is shown in Figure 12.22. Compare Figure 12.22 with lines 20-35 in Figure 11.11. The use of the subroutine in Figure 12.22 instead of the INPUT statement in Figure 12.21 will make it unnecessary to press the RETURN key after pressing the letter to be displayed.

Figure 12.21 Main program to display 3-D letters.

```
1000 REM  LETTER DISPLAY PROGRAM
1005 PRINT"■"
1010 DIM B$(6)
1020 DATA J,F,E,P,T
1030 FOR I=1 TO 5:READ B$(I):NEXT
1040 PRINT "ENTER J,F,E,P, OR T:"
1045 GOSUB 800:REM GET A$
1050 I=1
1060 IF A$=B$(I) OR I>5 THEN 1075
1070 I=I+1:GOTO 1060
1075 PRINT:PRINT
1080 ON I GOSUB 100,200,300,600,700
1085 PRINT:PRINT"■■■■■"
1090 PRINT "DO YOU WISH TO ENTER ANOTHER LETTER? ";
1095 GOSUB 800:REM GET A$
1100 IF A$="Y" THEN PRINT:GOTO 1040
1200 END

READY.
```

Figure 12.22 Subroutine to GET A$ from a blinking cursor.

```
800 REM GET A$ FROM BLINKING CURSOR
810 PRINT "■ ■";:FOR I=1 TO 200:NEXT
820 GET A$:IF A$<>"" THEN 840
830 PRINT" ■";:FOR I=1 TO 200:NEXT:GOTO 810
840 :PRINT A$;"■";:RETURN

READY.
```

Lines 1050-1070 in Figure 12.21 determine the value of I corresponding to the key pressed. Line 1075 moves the cursor to the beginning of the next line and then skips a line. Line 1080 is the ON...GOSUB statement that will branch to the appropriate subroutine to display the 3-D letter. After the 3-D letter has been displayed, or if a key other than J, F, E, P, or T has been pressed (resulting in a value of I=6 in line 1080), line 1085 will be executed. Line 1085 moves the cursor to the beginning of the next line and down four lines. (The cursor goes to different locations for the different 3-D letters.) Line 1090 prints the message **DO YOU WISH TO ENTER ANOTHER LETTER?** and line 1095 waits for a key to be pressed by calling the blinking cursor subroutine at line 800. If key Y is pressed, line 1100 will cause the program to branch back to line 1040. The PRINT statement in line 1100 is required so that the message in line 1040 will begin at the beginning of the next line. (Otherwise, it will begin at the location of the blinking cursor where the Y had been printed in line 840.) If any key other than Y is pressed in line 1095, the program ends at line 1200.

You can enter this program by typing in the statements shown in Figures 9.15, 9.18, 9.21, 9.24, 9.26, 12.18, 12.19, 12.22, and 12.21. A sample run of the program is shown in Figure 12.23.

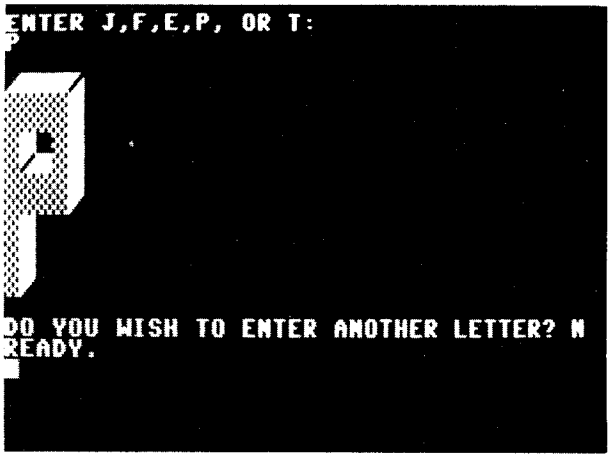


Figure 12.23 Sample run of 3-D letter display program.

EXERCISE 12-1
Write a program that:

a. stores the names and populations of the six New England states in the arrays S$(I) and P(I) as shown in Figure 12.1

b. plots a bar graph of the populations using a different graphic symbol for each state

c. sorts the populations in ascending order

d. plots a second bar graph (after pressing key S) with the populations in ascending order.

EXERCISE 12-2
Write a program that:

a. stores N test scores in DATA statements, with the value of N stored as the first entry of the first data statement

b. reads the test scores into an array S(I)

c. computes and prints out the average of the N test scores.

EXERCISE 12-3
If AV is the average of the N test scores stored in the array S(I), then the *standard deviation* is defined as

$$SD = \sqrt{\frac{1}{N} \sum_{I=1}^{N} (S(I) - AV)^2}$$

where the notation $\sum_{I=1}^{N}$ means the sum from I=1 to N. Modify the program in Exercise 12-2 to compute and print out the standard deviation of the test scores. Run the program for the following test scores:

Test Scores							
75	36	60	92	80	72	68	48
65	82	88	72	76	85	72	98
48	57	73	66	76	88	73	82
44	90	70	56	81	75	87	90

EXERCISE 12-4
The weights of a group of males and females are shown below. Write a program that will compute and print out the averages and standard deviations of the two groups of weights. Modify the program to compute and print the average and standard deviation of all weights (both male and female) (see Exercise 12-3).

Male		*Female*
200	138	103
185	205	105
185	159	112
125	230	102
140	150	160
195	140	120
190	170	115
155	145	130
185	169	140
140	215	118

EXERCISE 12-5
A person makes the following monthly deposits in a savings account paying 5% interest, compounded monthly.

Month	1	2	3	4	5	6	7	8	9	10	11	12
Deposit (dollars)	25	20	30	15	25	40	20	30	35	35	35	25

The identical pattern of deposits is repeated for a second and a third year. Write a program that will compute the amount of money the person has deposited and the total amount in the account at the end of 6, 12, 18, 24, 30, and 36 months. Read in the monthly deposits as an array D(I). (Note: If R is the annual interest rate, and it is compounded monthly, then each month the added interest is equal to R/12 times the amount in the account.)

EXERCISE 12-6
The polynomial:

$$P(x) = a_1x^4 + a_2x^3 + a_3x^2 + a_4x + a_5$$

can be written in the following nested form:

$$P(x) = a_5 + x(a_4 + x(a_3 + x(a_2 + x(a_1))))$$

If the coefficients a_i are stored as subscripted variables A(I), then the polynomial P can be evaluated, using the nested form, by the algorithm:

```
P = A(1)
for I=2 to 5
   P = A(I) + X * P
next I
```

Write a program that will use a similar nesting algorithm to evaluate the polynomial

$$P(x) = 3x^5 + 4x^4 - 2x^3 + 5x - 7$$

for values of x between -2 and $+2$ in steps of 0.2. Print out a table of x and P(x). Make your program general so that the coefficients are stored in DATA statements and the program can handle a polynomial of any order.

13 $$$ WITH STRINGS ATTACHED: LEARNING ABOUT LEFT$, RIGHT$, AND MID$

You have learned earlier in this book that memory cells with names like A$ and C3$ contain *strings*. The dollar sign, $, is used in BASIC to identify string-related names. CBM BASIC has a number of special functions that make it easy for you to manipulate strings. Learning how to use these functions will permit you to write many interesting programs. In this chapter you will learn:

1. to use the string functions LEFT$, RIGHT$, MID$, and LEN
2. to use the numeric/string functions STR$ and VAL
3. to use the ASCII code functions ASC and CHR$
4. how to display dollars and cents on the screen
5. how to write a program to shuffle and display a deck of playing cards
6. how to write a program to deal a hand of playing cards.

THE STRING FUNCTIONS LEFT$, RIGHT$, MID$, AND LEN

The string functions LEFT$, RIGHT$, and MID$ are used to extract some portion of a string. The function LEN is used to determine the length of a string.

LEFT$

The function **LEFT$(A$,I)** is a string that contains the left-most I characters of the string A$. For example, if A$="ABCDE", then **LEFT$(A$,2)** will be the string "AB". To verify this, type these statements, as shown in Figure 13.1:

```
A$="ABCDE"
?LEFT$(A$,2)
```

RIGHT$

The function **RIGHT$(A$,I)** is a string that contains the right-most I characters of the string A$. For example, if A$="ABCDE", then **RIGHT$(A$,2)** will be the string "DE". To verify this type **?RIGHT$(A$,2)** as shown in Figure 13.1.

MID$

The function **MID$(A$,I,J)** is a string that contains the J characters of A$ that start at position I (the first character of A$ is position 1). For example, if A$="ABCDE", then **MID$(A$,3,2)** will be the string "CD". To verify this, type **?MID$(A$,3,2)** as shown in Figure 13.1.

The function MID$ can also be written with only two arguments. In this case **MID$(A$,I)** is a string that contains the characters of A$ starting at position I and continuing to the end of the string A$. For example, if A$="ABCDE", then **MID$(A$,2)** will be the string "BCDE". To verify this, type **?MID$(A$,2)** as shown in Figure 13.1. Note carefully the difference between **MID$(A$,2)** and **RIGHT$(A$,2).** The former is a string containing the right-most characters starting at position 2 of A$, while the latter is a string containing the right-most two characters of A$.

```
A$="ABCDE"

READY.
?LEFT$(A$,2)
AB

READY.
?RIGHT$(A$,2)
DE

READY.
?MID$(A$,3,2)
CD

READY.
?MID$(A$,2)
BCDE

READY.
?LEN(A$)
 5

READY.
```

Figure 13.1 Examples of using the string functions LEFT$, RIGHT$, MID$, and LEN.

LEN

The function **LEN(A$)** is equal to the length of the string A$. It is a *numerical* value, not a string. For example, If A$="ABCDE", then the value of **LEN(A$)** is 5. To verify this, type **?LEN(A$)** as shown in Figure 13.1.

THE NUMERIC/STRING FUNCTIONS VAL AND STR$

It is important to understand the difference between a numerical value such as 456 and the string "456". It is like the difference between BOSTON and "BOSTON". BOSTON is a city in Massachusetts; "BOSTON" is a six-letter word that is the name of the city. Similarly, 456 is a number that you can add to other numbers. "456" is just the three characters 4,5, and 6. Sometimes you will need to convert a string like "456" to its corresponding numerical value 456. The function VAL will do this. You may also need to convert a numerical value such as 456 to its corresponding string "456". The function STR$ will do this.

VAL

The function **VAL(A$)** is equal to the numerical equivalent of the string A$. If A$ does not have a numerical equivalent, then **VAL(A$)** is equal to zero.

As an example of using the VAL function, clear the screen and type these statements, as shown in Figure 13.2.

A$="456"
?A$
?VAL(A$)

Note that **?A$** prints the string 456 starting in column 0, while **?VAL(A$)** prints the number 456 with the usual leading blank.

```
A$="456"

READY.
?A$
456

READY.
?VAL(A$)
 456

READY.
?VAL(A$)+10
 466

READY.
?A$+10

?TYPE MISMATCH  ERROR
READY.
?VAL("K")
 0

READY.
```

Figure 13.2 Examples of using the numeric/string function VAL.

In order to clarify the difference between VAL(A$) and A$, type these statements, as shown in Figure 13.2:

?VAL(A$)+10
?A$+10

Note that the number VAL(A$) can be added to 10, whereas trying to add the string A$ to the number 10 will produce the error message **?TYPE MISMATCH ERROR.** Finally, type **?VAL("K")** as shown in Figure 13.2. This shows that the value of the function VAL is zero if the string does not have a numerical equivalent.

STR$

The function **STR$(A)** is the equivalent of the numerical value A. As an example of using the STR$ function, clear the screen and type these statements, as shown in Figure 13.3:

A=456
?A
?STR$(A)

Note that both print statements print the number 456 with the leading blank. This means that the string STR$(A) actually contains the leading blank; it is the string " 456". To verify this type **?LEN(STR$(A))** as shown in Figure 13.3. This shows that STR$(A) really does contain four characters (blank, 4, 5, and 6).

```
A=456

READY.
?A
 456

READY.
?STR$(A)
 456

READY.
?LEN(STR$(A))
 4

READY.
?"$"+STR$(A)+".00"
$ 456.00

READY.
?"$"+MID$(STR$(A),2)+".00"
$456.00

READY.
```

Figure 13.3 Examples of using the numeric/string function STR$.

Strings can be put together (concatenated) to form longer strings. For example, type **?"$"+STR$(A)+".00"** as shown in Figure 13.3. The strings "$", " 456", and ".00" form the string "$ 456.00".

Suppose you wanted the total string to look like "$456.00" *without a blank between the dollar sign and* the first digit. You want to include all but the first character in the string STR$(A). Try typing **?"$"+MID$(STR$(A),2)+".00"** as shown in Figure 13.3. Do you see how it works? Remember that **MID$(STR$(A),2)** is the string containing all of the characters in STR$(A) after the first (that is, starting at position 2). We will again look at how to display dollars and cents later in this chapter.

The functions STR$ and VAL are reciprocal functions because they satisfy the following relations:

string = STR$(VAL(string))
value = VAL(STR$(value))

Verify this by typing the following statements, as shown in Figure 13.4:

?STR$(VAL("246"))
?VAL(STR$(246))

```
?STR$(VAL("246"))
 246

READY.
?VAL(STR$(246))
 246

READY.
```

Figure 13.4 STR$ and VAL are reciprocal functions.

THE ASCII CODE FUNCTIONS ASC AND CHR$

ASCII stands for "American Standard Code for Information Interchange." In this standard code a certain number is associated with each character (letter, digit, or special character). This code is used extensively throughout the computer industry for sending information from one computer to another and for sending data between terminals and computers. The Commodore 64/VIC 20 has special graphic symbols as well as special cursor and color keys that are not part of the ASCII standard. Therefore, the Commodore 64/VIC 20 uses an enhanced ASCII code to represent all of its characters and special functions. The Commodore 64/VIC 20 function ASC can be used to find the ASCII number associated with any character, and the function CHR$ can be used to find the character associated with any ASCII number.

ASC

The function **ASC(A$)** is equal to the ASCII code of the first character in the string A$. To find some ASCII codes, clear the screen and type these statements, as shown in Figure 13.5:

?ASC("A")
?ASC("SHIFT S")
?ASC("ABC")
?ASC("HOME")
?ASC("7")

Letters, graphic symbols, digits, and special function keys all have ASCII numbers. Note that the ASCII

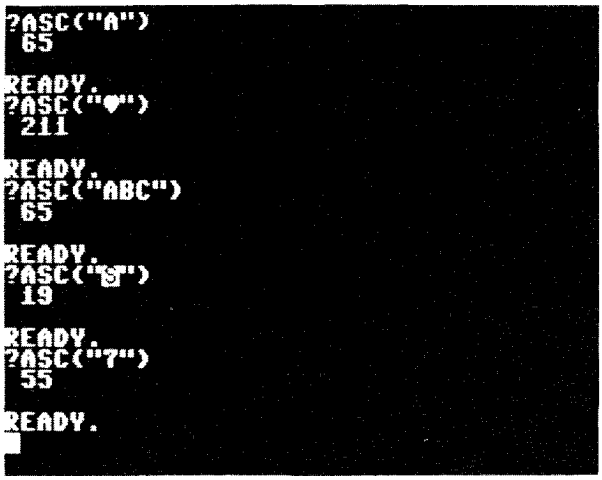


Figure 13.5 Examples of ASCII codes of some characters.

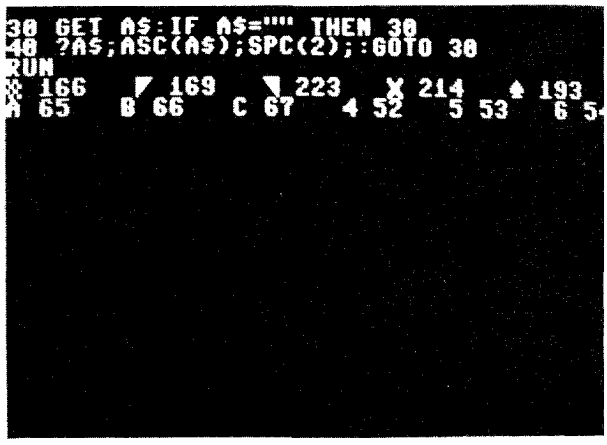
number for a digit is different from the digit itself (for example, 55 is the ASCII code for 7). Also note that the function ASC("ABC") is the ASCII code of the first character, A, in the string "ABC".

If you want to see what some other ASCII numbers are, try typing this two-line program:

```
30 GET A$: IF A$= "" THEN 30
40 ?A$ ; ASC(A$); SPC(2);: GOTO 30
```

This program will print any character you type followed by its ASCII code. A sample run is shown in Figure 13.6. The ASCII codes for all of the characters are given in Appendix B.

Figure 13.6 Program to find the ASCII codes of each key pressed.



CHR$

If you know the ASCII code of a character you can generate the string of that character using the function **CHR$(A)** where A is the ASCII code of the character.

Using the results in Figure 13.5, try typing the following statements, as shown in Figure 13.7:

```
?CHR$(65)
?CHR$(211)
?CHR$(55)
```

Now type **?CHR$(19).** The cursor moves HOME, a line is skipped (because the PRINT statement does not end with a semicolon), and then the READY message is printed. Thus, the function CHR$ can be used in a program to produce cursor movements equivalent to including cursor keystrokes within a string. The ASCII codes and CHR$ functions associated with the various cursor movements, color keys, and reverse video functions are given in Appendix B.

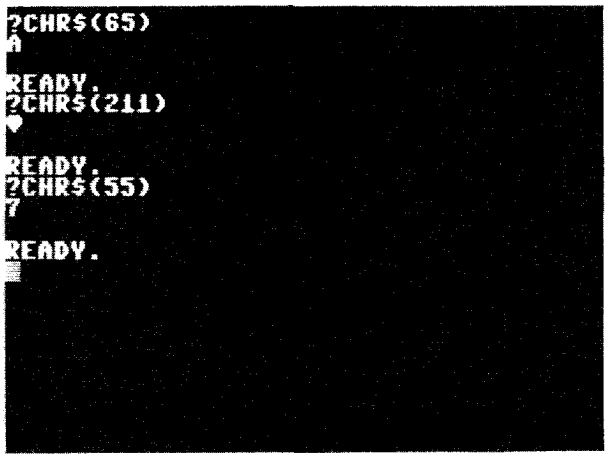


Figure 13.7 Examples of printing characters using their ASCII codes.

PRINTING DOLLARS AND CENTS

Lots of practical programs involve money and require you to display dollars and cents on the screen. This is not so easy as it may seem. First of all, if you compute some monetary value such as interest on a savings account, you will want to round to the nearest cent. You can do this by adding 0.005 to the value and then displaying only two places after the decimal point. In order to try this, type these statements, as shown in Figure 13.8:

```
A=208.4978
A1=A+.005
?A1
?INT(A1*100)/100
```

Note that although this method rounded the amount to the nearest cent, the Commodore 64/VIC 20 does not display trailing zeros. Therefore, fifty cents (.50) is printed as .5.

```
A=208.4978

READY.
A1=A+.005

READY.
?A1
 208.5028

READY.
?INT(A1*100)/100
 208.5

READY.
```

Figure 13.8 Rounding a monetary value to the nearest cent.

One way to print the .5 as .50 is to convert the dollars and cents separately to their string equivalents and then display these strings. To investigate this possibility, type these statements, as shown in Figure 13.9:

A2=INT(A1)

?A2

A2$=MID$(STR$(A2),2)

?A2$

Note that A2 is the dollar value and A2$ is the string representation of this value, with the leading blank eliminated.

Figure 13.9 A2$ is a string representation of the dollar amount.

```
A2=INT(A1)

READY.
?A2
 208

READY.
A2$=MID$(STR$(A2),2)

READY.
?A2$
208

READY.
```

In order to obtain a string representation of the cents value, type these statements, as shown in Figure 13.10:

A3=A1−A2

?A3

A3$=MID$(STR$(A3),3,2)

?A3$

Note that the cents value A3 is found by subtracting the dollar value from the total rounded amount. A string representing the cents amount consists of the third and fourth characters in the string STR$(A3) (the first character is a blank).

```
A3=A1-A2

READY.
?A3
 .502799988

READY.
A3$=MID$(STR$(A3),3,2)

READY.
?A3$
50

READY.
?"$";A2$;".";A3$
$208.50

READY.
```

Figure 13.10 A3$ is a string representation of the cents amount.

The total dollars and cents can be displayed by typing **?"$";A2$;".";A3$** which will display

$208.50

as shown at the bottom of Figure 13.10.

The statements shown in Figures 13.8, 13.9, and 13.10 can be combined to form the subroutine shown in Figure 13.11. This subroutine should print the value of A in the form $XX.YY. For the first value of A shown in Figure 13.11 the subroutine works well. However, for a value of A=159.996 the subroutine prints $160..9. The problem can be found by looking at the values of A1 and A3 as shown in Figure 13.11. If the fractional part of A1 (the cents value, A3) is less than 0.01 then A3 will be stored in scientific notation. Now the third and fourth characters in STR$(A3) are ".9" rather than "00". The subroutine shown in Figure 13.11 can be fixed by adding the statement **925 IF A3<.01 THEN A3$="00": GOTO 940** as shown in Figure 13.12. Note that this modified subroutine prints the correct dollars and cents values for all of the examples shown.

The last example shown in Figure 13.12 rounds 999999.999 to $1000000.00. When writing a check for this value (or any value over $1,000.00) it would look better and make the value easier to read if you included commas in the dollar amount.

```
LIST

900 REM  PRINT A AS $XX.YY
910 A1=A+.005:A2=INT(A1):A3=A1-A2
920 A2$=MID$(STR$(A2),2)
930 A3$=MID$(STR$(A3),3,2)
940 PRINT"$";A2$;".";A3$;
950 RETURN
READY.
A=208.4978:GOSUB 900
$208.50
READY.
A=159.996:GOSUB 900
$160..9
READY.
?A1
 160.001

READY.
?A3
 9.99987125E-04

READY.
```

Figure 13.11 This subroutine for displaying dollars and cents will not work for cents values less than 0.01.

```
LIST

900 REM  PRINT A AS $XX.YY
910 A1=A+.005:A2=INT(A1):A3=A1-A2
920 A2$=MID$(STR$(A2),2)
925 IF A3<.01 THEN A3$="00":GOTO 940
930 A3$=MID$(STR$(A3),3,2)
940 PRINT"$";A2$;".";A3$;
950 RETURN
READY.

A=208.4978:GOSUB 900
$208.50
READY.
A=159.996:GOSUB 900
$160.00
READY.
A=999999.999:GOSUB 900
$1000000.00
READY.
```

Figure 13.12 Modified subroutine that displays correct dollars and cents values.

Adding Commas to the Dollar Amount

Suppose that you want to add commas to the value

$$\underbrace{\$\ \underbrace{2}_{\text{A6\$}},\underbrace{357}_{\text{A5\$}},\underbrace{829}_{\text{A4\$}}.\underbrace{49}_{\text{A3\$}}}$$

That is, if L=LEN(A2$), then:

$$\text{A4\$}=\begin{cases}\text{A2\$} & (\text{L}<=3)\\ \text{MID\$(A2\$,L-2,3)} & (\text{L}>3)\end{cases}$$

$$\text{A5\$}=\begin{cases}\text{LEFT\$(A2\$,L-3)} & (\text{L}<=6)\\ \text{MID\$(A2\$,L-5,3)} & (\text{L}>6)\end{cases}$$

$$\text{A6\$}=\ \text{LEFT\$(A2\$,L-6)}\qquad (\text{L}>6)$$

The algorithm for adding the commas will then be:

if L<=3
then print $A4$.A3$
else if L<=6
  then print $A5$,A4$.A3$
  else print $A6$,A5$,A4$.A3$

Figure 13.13 shows how this algorithm can be added to the subroutine shown in Figure 13.12. Lines 940-975 implement the algorithm described above. Two examples of using this subroutine are also shown in Figure 13.13.

Figure 13.13 Subroutine that includes commas when displaying dollars and cents.

```
900 REM  PRINT A AS $ XX.XXX.YY
910 A1=A+.005:A2=INT(A1):A3=A1-A2
920 A2$=MID$(STR$(A2),2)
925 IF A3<0.01 THEN A3$="00":GOTO 940
930 A3$=MID$(STR$(A3),3,2)
940 L=LEN(A2$)
945 PRINT"$";
950 IF L<=3 THEN A4$=A2$:GOTO 975
955 IF L<=6 THEN A5$=LEFT$(A2$,L-3):GOTO
 967
960 A6$=LEFT$(A2$,L-6):PRINT A6$;",";
965 A5$=MID$(A2$,L-5,3)
967 PRINT A5$;",";
970 A4$=MID$(A2$,L-2,3)
975 PRINT A4$;".";A3$;
980 RETURN
READY.
A=999999.999:GOSUB 900
$1,000,000.00
READY.
A=32541.658:GOSUB 900
$32,541.66
READY.
```

PLAYING CARDS

As another example of using string functions, we will now develop some subroutines that will be useful in card game programs. The first thing to decide is how to represent a deck of cards within the computer. It is convenient to associate a number between 1 and 52 with each card in the deck. We will use the numbering system shown in Figure 13.14. For example, the seven of hearts is number 33 and the jack of diamonds is number 24.

The value of a card (ace through king) has a value number V, and each suit has a suit number S, as defined in Figure 13.14.

Figure 13.14 Each card in the deck is associated with a number between 1 and 52.

	Club	Diamond	Heart	Spade	Value No. V
A	1	14	27	40	1
2	2	15	28	41	2
3	3	16	29	42	3
4	4	17	30	43	4
5	5	18	31	44	5
6	6	19	32	45	6
7	7	20	33	46	7
8	8	21	34	47	8
9	9	22	35	48	9
T	10	23	36	49	10
J	11	24	37	50	11
Q	12	25	38	51	12
K	13	26	39	52	13
Suit S No.	1	2	3	4	

It is usually easier to use a card number, C, as much as possible to distinguish cards and then to use C to find the value and suit of the card when needed. Given a card number C, the corresponding suit number S is given by S=INT((C−1)/13)+1. Verify this by trying some examples from Figure 13.14. For example, if C=26 (king of diamonds):

```
S=INT(25/13)+1
 =1+1=2
```

Once you know S, the value number V can be determined from the equation V=C−(S−1)*13. For example, if C=26, then S=2 and V=26−(2−1)*13=13.

It is convenient to store all of the card numbers in an array C%(I) (we will use an integer array to save memory). This array can be initialized with these statements:

```
DIM C%(52)
FOR I=1 TO 52:  C%(I)=I:  NEXT
```

Thus, for example, C%(47)=47 and represents the eight of spades.

Suppose you want to display the nineteenth card in the deck. The card number is C%(19)=19. The suit number is

```
S=INT((C%(19)−1)/13)+1
 =INT(18/13)+1
 =2
```

and the value number is:

```
V=C%(19)−(S−1)*13
 =19−1*13
 =6
```

Therefore, as defined in 13.14, the card is the six of diamonds. To display this value, define the two strings V$ and S$ shown in Figure 13.15. Note that the position of each value character in V$ corresponds to the appropriate value number V in Figure 13.14. Therefore, the single value character V1$ corresponding to the value number V is given by **V1$=MID$(V$,V,1).** Similarly, the position of each suit's graphic symbol in S$ corresponds to the appropriate suit number S in Figure 13.14. Therefore, the single suit character S1$ corresponding to the suit number S is given by **S1$=MID$(S$,S,1).**

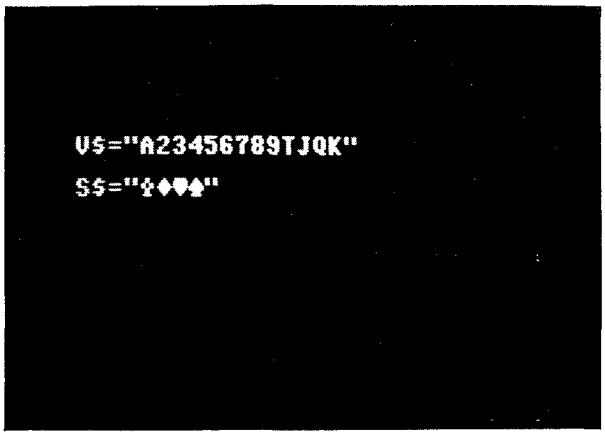


Figure 13.15 Definition of the value string V$ and the suit string S$.

These ideas are incorporated in the two subroutines shown in Figure 13.16. The subroutine given by lines 3000-3050 sets up the deck by dimensioning and initializing C%(I) and defining V$ and S$. This subroutine should be called once at the beginning of any program involving playing cards.

The subroutine in lines 3100-3150 in Figure 13.16 will find the value string V1$ and the suit string S1$ of the card located at position P in the array C%, that is, the card with number C%(P). Lines 3110-3120 define the suit number S and value number V using the formulas given above. Lines 3130-3140 find the single-character strings V1$ and S1$.

In order to test these subroutines, type these statements, as shown in Figure 13.17:

```
GOSUB 3000
P=33: GOSUB 3100: ?V1$;S1$
P=52: GOSUB 3100: ?V1$;S1$
```

Note that card number 33 is the seven of hearts, and card number 52 is the king of spades, as shown in Figure 13.14.

Figure 13.16 Subroutines to set up the deck (line 3000) and pick a card at location P (line 3100).

```
3000 REM  PLAYING CARD SETUP
3010 DIM C%(52)
3020 FOR I=1 TO 52:C%(I)=I:NEXT
3030 V$="A23456789TJQK"
3040 S$="♠♥♦♣"
3050 RETURN
3100 REM  PICK CARD AT LOCATION P
3110 S=INT((C%(P)-1)/13)+1
3120 V=C%(P)-(S-1)*13
3130 V1$=MID$(V$,V,1)
3140 S1$=MID$(S$,S,1)
3150 RETURN

READY.
```

```
GOSUB 3000

READY.
P=33:GOSUB 3100:?V1$;S1$
7♥

READY.
P=52:GOSUB 3100:?V1$;S1$
K♠

READY.
```

Figure 13.17 Testing the subroutines given in Figure 13.16.

You can display the entire deck by adding the main program shown in Figure 13.18. Line 20 sets up the deck, and line 25 skips a line and initializes the tabulation variable N. The FOR...NEXT loop in lines 30-60 increments P from 1 to 52, finds V1$ and S1$ for the card at position P (line 40), and prints these value and suit characters at tab N (line 50). Line 55 increases the tabulation variable N by 6 to form a new column and, if the value is greater than 19 (column 4), executes a PRINT command to reset the cursor to the next line and resets N to 1. The result of running this program is shown in Figure 13.19. Note that the cards are printed in the order shown in Figure 13.14. In order to print them in a random order, you must first shuffle the deck.

Shuffling a Deck of Cards

To shuffle a deck of cards, all you have to do is to scramble the order of the card numbers stored in the

Figure 13.18 Program to display an entire deck of cards.

```
10 REM  DISPLAY DECK
20 GOSUB 3000:REM  SETUP DECK
25 PRINT:N=1
30 FOR P=1 TO 52
40 GOSUB 3100:REM  GET NEXT CARD
50 PRINT TAB(N);V1$;S1$;:REM  DISPLAY CARD
55 N=N+6:IF N>19 THEN PRINT:N=1
60 NEXTP
100 END

READY.
```

```
RUN
A♠   2♠   3♠   4♠
5♠   6♠   7♠   8♠
9♠   T♠   J♠   Q♠
K♠   A♦   2♦   3♦
4♦   5♦   6♦   7♦
8♦   9♦   T♦   J♦
Q♦   K♦   A♥   2♥
3♥   4♥   5♥   6♥
7♥   8♥   9♥   T♥
J♥   Q♥   K♥   A♣
2♣   3♣   4♣   5♣
6♣   7♣   8♣   9♣
T♣   J♣   Q♣   K♣

READY.
```

Figure 13.19 Result of running program shown in Figure 13.18.

card array C%(I). The following simple algorithm will do this:

for I=1 TO 52
  find random number J between 1 and 52
  interchange C%(I) and C%(J)
next I

This algorithm interchanges each element in C%(I) with another element selected at random.

RND(1) is a random number with a value greater than 0 and less than 1; therefore, **J=INT(52*RND(1)+1)** will be a random integer between 1 and 52.

A subroutine that will shuffle the deck while displaying a blinking version of the word "SHUFFLING" is shown in Figure 13.20. The FOR...NEXT loop in lines 3220-3280 corresponds to the *for...next* loop in the algorithm. Line 3260 interchanges C%(I) and C%(J). Each time through the loop "**S H U F F L I N G**" is printed in line 3240. The cursor is backspaced seventeen positions in line 3270, using the cursor string B7$ defined in line 3210. This will cause the word "**S H U F F L I N G**" to be printed at the same location each time the loop is executed. However, line 3230 turns on the reverse video every

other time through the loop. The reverse video is turned off each time through the loop in line 3270 (half the time it is already off). This will produce a blinking effect during the shuffling of the deck. Type in this subroutine and try it.

Figure 13.20 Subroutine to shuffle a deck of cards.

```
3200 REM  SHUFFLE DECK
3210 B7$="■■■■■■■■■■■■■■■■■■■■■"
3220 FOR I=1 TO 52
3230 IF I-INT(I/2)*2=0 THEN PRINT"■";
3240 PRINT "S H U F F L I N G";
3250 J=INT(52*RND(1)+1)
3260 T=C%(I):C%(I)=C%(J):C%(J)=T
3270 PRINT B7$;"■";
3280 NEXTI:RETURN

READY.
```

In order to produce a "random" seed for the random number generator (so that you will not deal the same hands each time you power up your Commodore 64/VIC 20—see Chapter 4) you should add the statement **I=RND(−TI)** to the card set up subroutine as shown in line 3015 of Figure 13.21.

Add lines 27 and 28 shown in Figure 13.22 to the main program given in Figure 13.18. This new program will shuffle the deck and then display all the cards. A sample run is shown in Figure 13.23.

Figure 13.21 Add line 3015 to the card setup subroutine when using the random function RND.

```
3000 REM  PLAYING CARD SETUP
3010 DIM C%(52)
3015 I=RND(-TI)
3020 FOR I=1 TO 52:C%(I)=I:NEXT
3030 V$="A23456789TJQK"
3040 S$="♠♥♦♣"
3050 RETURN
3100 REM  PICK CARD AT LOCATION P
3110 S=INT((C%(P)-1)/13)+1
3120 V=C%(P)-(S-1)*13
3130 V1$=MID$(V$,V,1)
3140 S1$=MID$(S$,S,1)
3150 RETURN

READY.
```

Figure 13.22 Main program to display shuffled deck of cards.

```
10 REM  DISPLAY DECK
20 GOSUB 3000:REM  SETUP DECK
25 PRINT:N=1
27 GOSUB 3200:REM  SHUFFLE DECK
28 PRINT:PRINT
30 FOR P=1 TO 52
40 GOSUB 3100:REM  GET NEXT CARD
50 PRINT TAB(N);V1$;S1$;:REM  DISPLAY CARD
55 N=N+6:IF N>19 THEN PRINT:N=1
60 NEXTP
100 END

READY.
```

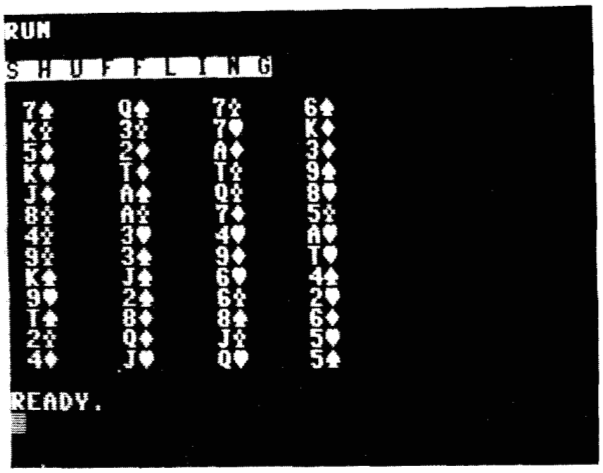


Figure 13.23 Sample run of program shown in Figure 13.22.

Dealing a Hand of Cards

The program shown in Figure 13.22 can easily be modified to deal a hand of cards. All you have to do is divide the cards among a number of players and limit the number of cards dealt to the desired number.

Let NP=number of players and NC=number of cards per hand.

The first card dealt to each player can be displayed on a single line with the statements:

```
50 P=1:PRINT
70 FOR J=1 TO NP
80 GOSUB 3100: REM DEAL NEXT CARD
90 PRINT V1$;S1$;SPC(2);
100 P=P+1
110 NEXTJ
```

Note that P points to the next card in the deck, and subroutine 3100 finds the card at C%(P).

To deal NC cards to each player and display them on succeeding lines, add these statements, as shown in Figure 13.24:

```
60 FOR I=1 TO NC
120 PRINT
130 NEXTI
```

In this program lines 30-40 allow the user to type in the values NP and NC. Line 50 points to the top card of the deck and skips a line. Lines 60-130 are the outer FOR...NEXT loop that prints NC rows of cards. Lines 70-110 are the inner FOR...NEXT loop that deals NP cards and displays them on one line. Line 100 points to the next card in the deck after each one is dealt. The PRINT statement in line 120 is necessary to move the cursor to the beginning of the next line after each round of cards is dealt. Two sample runs of this program are shown in Figure 13.25.

Figure 13.24 Program to deal a hand of cards.

```
10 REM  DEAL HAND OF CARDS
20 GOSUB 3000:REM  SETUP DECK
25 PRINT
27 GOSUB 3200:REM  SHUFFLE DECK
28 PRINT:PRINT
30 INPUT"ENTER NUMBER OF PLAYERS ";NP
40 INPUT"ENTER NUMBER OF CARDS PER HAND ";NC
50 P=1:PRINT
60 FOR I=1 TO NC
70 FOR J=1 TO NP
80 GOSUB 3100:REM  DEAL NEXT CARD
90 PRINT V1$;S1$;SPC(2);
100 P=P+1
110 NEXTJ
120 PRINT
130 NEXTI
140 END

READY.
```

Figure 13.25 Sample runs of program shown in Figure 13.24.

```
RUN
S H U F F L I N G

ENTER NUMBER OF PLAYERS ? 2
ENTER NUMBER OF CARDS PER HAND ? 6

3♤  A♦
4♠  6♥
3♦  K♤
6♠  K♥
T♥  3♥
J♦  3♠

READY.
```

```
RUN
S H U F F L I N G

ENTER NUMBER OF PLAYERS ? 4
ENTER NUMBER OF CARDS PER HAND ? 13

A♤  3♤  2♥  5♠
T♦  A♦  4♠  9♦
2♤  T♥  J♤  Q♥
K♤  K♥  2♠  7♥
5♦  4♥  5♦  A♠
8♥  K♦  J♠  3♦
A♥  9♠  3♠  Q♤
J♥  3♥  6♥  Q♠
6♤  9♤  T♦  T♤
7♠  6♠  T♠  9♥
J♦  4♦  8♦  4♤
7♤  8♠  Q♦  2♦
K♠  5♤  5♥  8♤

READY.
```

It would be nice if you could sort each hand by suit. This is easier to do than you may think.

Sorting a Hand by Suit

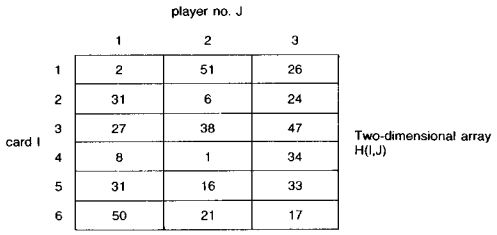
Suppose that a hand contains the cards shown in Figure 13.26a, where the card number for each card is also given (see Figure 13.14). If the card numbers are sorted in ascending order, the cards will be sorted in ascending order by suit, as shown in Figure 13.26b. This illustrates the advantage of using card numbers to represent playing cards inside the computer.

Figure 13.26 A hand of cards can be sorted by suit by sorting the card numbers in ascending order.

Card	Card No.	Card No.	Card
6H	32	2	2C
4D	17	3	3C
8D	21	6	6C
4S	43	17	4D
3C	3	21	8D
JS	50	32	6H
6C	6	35	9H
9H	35	43	4S
2C	2	50	JS
(a)		(b)	

In order to sort a hand, we will need to store the card numbers for each card in the hand. We can store these in an array. For convenience we will use a *two-dimensional* array, H(I,J), in which each column will contain the card numbers for a different player, as shown in Figure 13.27. To sort all hands, we will need to sort each column in ascending order.

Figure 13.27 Each column of the two-dimensional array H(I,J) contains the card numbers for one player.



The array H(I,J) must contain NC rows (number of cards per hand) and NP columns (number of players). Since we do not know what these values are until lines 30-40 in Figure 13.24 are executed, we will add the following dimension statement at line 45: **45 DIM H(NC,NP).** Every time a card is dealt, we need to add the card number to the array H(I,J) by adding the statement **75 H(I,J)=C%(P)** as shown in Figure 13.28. Note that this statement is inside the two nested FOR...NEXT loops and that the array H(I,J) will be filled one row at a time. In Figure 13.28 we have added the one additional statement **135 GOSUB 200: REM DISPLAY SORTED HAND** where we will put everything that we do not yet know how to do.

The subroutine at line 200 will have to sort each column in H(I,J) in ascending order and then display

Figure 13.28 Main program to deal a hand of cards and then display the sorted hand.

```
10 REM  DEAL HAND OF CARDS
20 GOSUB 3000:REM  SETUP DECK
25 PRINT
27 GOSUB 3200:REM  SHUFFLE DECK
28 PRINT:PRINT
30 INPUT"ENTER NUMBER OF PLAYERS ";NP
40 INPUT"ENTER NUMBER OF CARDS PER HAND ";NC
45 DIM H(NC,NP)
50 P=1:PRINT
60 FOR I=1 TO NC
70 FOR J=1 TO NP
75 H(I,J)=C%(P)
80 GOSUB 3100:REM  DEAL NEXT CARD
90 PRINT V1$;S1$;SPC(2);
100 P=P+1
110 NEXTJ
120 PRINT
130 NEXTI
135 GOSUB 200:REM  DISPLAY SORTED HAND
140 END

READY.
```

the corresponding cards. This subroutine is shown in Figure 13.29. Line 205 prints the word "**SORTING**", so that if it takes a little time (it will) the user will know what is going on. Line 210 will go to a subroutine that will sort each column in H(I,J). The nested FOR...NEXT loops in lines 220-260 are similar to the ones in lines 60-130 of Figure 13.28 that displayed the original hand. The subroutine at line 3100 will find the card at position P in the array C%; that is, the card with card number C%(P). This was useful in line 80 in Figure 13.28 where we were incrementing P each time through the loop. In Figure 13.29, however, we do not know P, but we do know the card number—it is H(I,J). Therefore, we would like to use the subroutine at line 3100 to find the value of the card with card number H(I,J). We must make C%(P) contain the value H(I,J). Because the array element C%(0) is not normally used but is available, we will use this location to store H(I,J), as shown in line 240 in Figure 13.29. Note that we must set P=0 so that the subroutine at line 3100 shown in Figure 13.21 will use C%(0) as the equivalent of H(I,J).

Figure 13.29 Subroutine to display the sorted hands of cards.

```
200 REM  DISPLAY SORTED HAND
205 PRINT:PRINT "SORTING":PRINT
210 GOSUB 2000:REM  SORT COLUMNS OF H
220 FOR I=1 TO NC
230 FOR J=1 TO NP
240 P=0:C%(0)=H(I,J):GOSUB 3100
250 PRINT V1$;S1$;SPC(2);
260 NEXTJ:PRINT:NEXTI:RETURN

READY.
```

We must now sort the columns of H(I,J) in ascending order. If you study the sorting algorithm that we developed in the last chapter (see Figure 12.10), you will note that all we have to do is apply this same algorithm to each column of H(I,J). The resulting algorithm is given in Figure 13.30. The BASIC implementation of this algorithm is written as a subroutine in Figure 13.31.

```
for J=1 TO NP
  for I=1 TO NC-1
    for K=I+1 TO NC
      if H(I,J)<=H(K,J)
      then do nothing
      else interchange H(I,J) and H(K,J)
    nextK
  nextI
nextJ
```

Figure 13.30 Algorithm for sorting each column of H(I,J) in ascending order.

Figure 13.31 Subroutine to sort each column of the array H(I,J).

```
2000 REM SORT EACH COLUMN OF H(NC,NP)
2010 FOR J=1 TO NP
2020 FOR I=1 TO NC-1
2030 FOR K=I+1 TO NC
2040 IF H(I,J)<=H(K,J) THEN 2060
2050 T=H(I,J):H(I,J)=H(K,J):H(K,J)=T
2060 NEXTK:NEXTI:NEXTJ:RETURN

READY.
```

We have now written all of the subroutines needed to run the program shown in Figure 13.28. A sample run is shown in Figure 13.32. Note that each hand is sorted by suit with the suits displayed in order (clubs, diamonds, hearts, and spades).

If you were sorting a bridge hand, you would want the ace to be high rather than low, and thus to be placed after the king when displaying each suit. You can do this in Exercise 13-4.

EXERCISE 13-1

Write a program that will accept a string A$ and a substring B$ from the keyboard, and then search for the first occurrence of the substring B$ in A$. If a match is found, the value of P should be set to the position in A$ of the first character of B$. (P=1 corresponds to the first character in A$.) If no match is found, set P=0.

```
S H U F F L I N G

ENTER NUMBER OF PLAYERS ? 6
ENTER NUMBER OF CARDS PER HAND ? 7

K♦  A♣  Q♦  Q♣  6♣  2♣
3♥  T♣  K♥  9♥  6♦  5♥
9♦  7♥  Q♥  8♦  4♥  T♠
K♠  A♦  T♥  4♠  J♥  A♥
9♠  2♦  J♣  6♠  8♣  5♠
2♦  4♣  6♥  2♥  T♦  7♣
8♠  3♣  J♦  5♣  4♦  A♠

SORTING

2♦  A♣  J♣  5♣  6♣  2♣
9♦  3♣  J♦  Q♣  8♣  7♣
K♦  4♣  Q♦  8♦  4♦  A♥
3♥  T♣  6♥  2♥  6♦  5♥
8♠  A♦  T♥  9♥  T♦  A♠
9♠  7♥  Q♥  4♠  4♥  5♠
K♠  2♦  K♥  6♠  J♥  T♠

READY.
```

Figure 13.32 Sample run of the program shown in Figure 13.28.

EXERCISE 13-2
Modify the program in Exercise 13-1 to find all occurrences of B$ in A$. Store the locations of all matches in the array P(I). A value of P(I)=0 will indicate that there are no more matches in the string.

EXERCISE 13-3
Write a program that will replace all occurrences of the substring B$ in A$ with the substring C$. The program should accept the string A$ and the two substrings B$ and C$ from the keyboard.

EXERCISE 13-4
Write a program that will shuffle a deck of cards and deal four bridge hands. (Each hand contains 13 cards.) The four hands are to be sorted with the ace as the high card in each suit.

EXERCISE 13-5
Some card games require the players to cut for the deal, with either the high card or low card winning the deal. Write a program that will allow NP players to cut for the deal and will assign the deal to the player cutting the highest card.

EXERCISE 13-6
Write a program that will print a check. The program should allow the user to type in a name, address, date, and check amount. Print the check in the form shown below. Check to make sure that the amount is in the range 0–9,999,999.00 and print leading asterisks in the amount box on the check.

Exercise 13-6

PAY TO THE ORDER OF

JOHN DOE
1234 VIC DRIVE
ROCHESTER, MI 48063

DATE

12/13/83

PAY THIS AMOUNT

******1,250.41

14 LEARNING TO PEEK AND POKE

As you have learned, the Commodore 64/VIC 20 contains a large number of memory locations that are used to store the program and data. Some of this memory is read/write memory (RAM), some is read-only memory (ROM), and some is special input/output memory that allows communication with the outside world. This communication includes getting data from the keyboard and writing and reading data to and from a cassette tape and disk drive.

When writing a program in BASIC, you refer to a memory cell by its name, such as A$ or C3. You do not know exactly which memory location within the Commodore 64/VIC 20 contains the data in C3. The BASIC interpreter automatically takes care of assigning these locations. However, in order to use the full power of the Commodore 64/VIC 20, you must sometimes read and write data to *specific* memory locations within the computer. In order to do this with maximum flexibility and speed you must write the program in assembly language.

You can, however, read and write data to specific memory locations with BASIC by using the PEEK and POKE statements. In this chapter you will learn how:

1. data is stored in memory locations in the Commodore 64/VIC 20
2. to use the PEEK and POKE statements
3. to control the border, background, and character colors of the screen
4. to use the alternate character set of the Commodore 64/VIC 20, including some new graphic symbols
5. to POKE graphic pictures directly on the Commodore 64/VIC 20 screen
6. to control the sounds produced by the Commodore 64/VIC 20
7. to define your own graphic characters on the Commodore 64/VIC 20
8. to create Sprite graphics figures on the Commodore 64

THE PEEK AND POKE STATEMENTS

The 6510/6502 microprocessor, the "brain" of the Commodore 64/VIC 20 (see Figure 3.1), is an integrated circuit chip that contains forty pins. It can address a total of 65,536 memory locations (with addresses between 0-65535), because sixteen of the forty pins are used to form an address, and each of the address pins can be either high or low (1 or 0). Thus an address might be represented by the sixteen bits 0011010111000001. This *binary* number is equivalent to the *decimal* number 13761, so this memory location would have an address of 13761. Since each of the

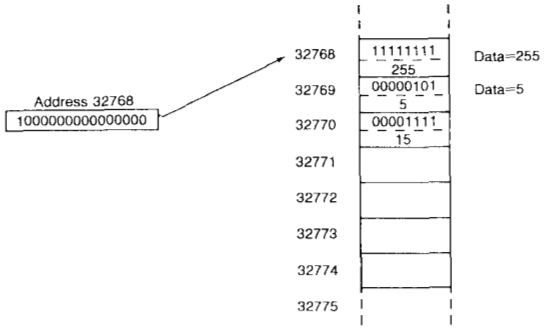
sixteen bits in the address can be either a 1 or a 0, the total number of possible addresses is $2^{16} = 65536$. A VIC 20 actually contains considerably less than this maximum amount of memory, since it comes with only 5K bytes and is ordinarily expanded in memory by adding either 8K, 16K, or 24K RAM. The Commodore 64 on the other hand actually has 65,536 RAM locations (64K). This is achieved by having RAM and ROM share some locations and RAM, ROM, and input/output devices share others. At any time, each shared address is either RAM or ROM or possibly an input/output device but which it is depends upon the contents of certain other nonshared locations which are always RAM or ROM.

It is sometimes convenient to represent binary numbers as *hexadecimal* numbers. This is not necessary when using BASIC, because the PEEK and POKE statements use only *decimal* numbers. However, if you want to program in assembly language, then the use of hexadecimal numbers is essential. Although you do not need to know about them for anything in this book, a brief discussion of hexadecimal numbers is given in Appendix D.

In addition to the sixteen address pins, the 6510/6502 microprocessor uses eight pins to transmit and receive data. These pins are connected to all of the memory chips in the Commodore 64/VIC 20. Thus, data is moved between memory locations in groups of eight bits called *bytes*. The total number of different values that a data byte can have is $2^8 = 256$. Thus, data in a memory location in the Commodore 64/VIC 20 can have a value between 0-255. This relationship between addresses and data is shown in Figure 14.1. In this figure memory location 32768 contains a data value of 255 (eight 1s), and memory location 32769 contains a data value of 5.

You can find the data value stored in a particular memory location by using the PEEK statement. You can store a particular data value in a given memory location (provided it contains a RAM cell) by using the POKE statement.

Figure 14.1 Each address in the range 0-65535 points to a memory location containing data in the range 0-255.



PEEK

The function **PEEK**(*addr*) returns the data value stored in the memory location with an address *addr*. The value of *addr* must be in the range 0-65535. Try printing some value of PEEK to see what you get. For example, try these statements, as shown in Figure 14.2:

?PEEK(792)
?PEEK(770)

(These locations in your Commodore 64/VIC 20 will probably contain data values different from those shown in Figure 14.2.)

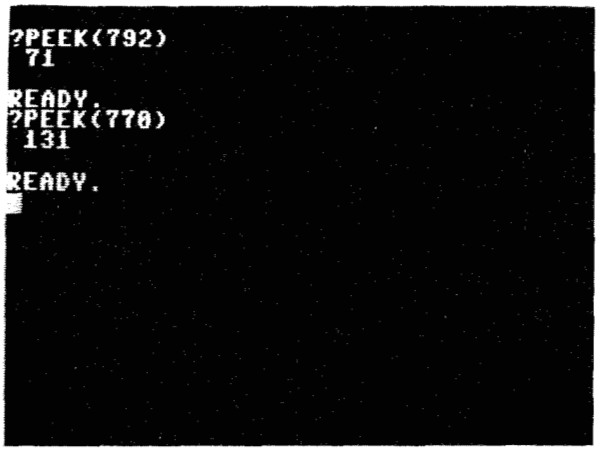


Figure 14.2 Memory location 792 contains the value 71 and memory location 770 contains the value 131.

Certain memory locations have particular meanings to the Commodore 64/VIC 20. For example, memory location 197 will have a value of 64 if no key is being pressed. If a key is pressed, then memory location 197 will have a value between 0 and 64 depending upon the position of the key on the keyboard.

To see how this works, type and run this one-line program: **10 ? PEEK(197) : GOTO 10.** The data value in location 197 will continue to be displayed and scroll off the screen. Press several keys while the program is running and watch the numbers change. Note that when you are not pressing any key, the value displayed is 64.

This fact can be used in programs if you want to wait while a key is being pressed or until a key is pressed. For example, the program shown in Figure 14.3 will cause a ball to bounce back and forth across the screen as long as any key is pressed. When you release the key, the ball will stop. Each time a new ball is printed by lines 10 and 20, the subroutine in line 50 is called. Line 50 will loop on itself if **PEEK(197)=64,** that is, if no key is being pressed. As soon as you press

a key, the program returns from the subroutine and prints another ball. Study lines 10 and 20 and make sure you understand how the ball moves. Line 10 moves the ball from left to right, and line 20 moves the ball from right to left. Type in this program and run it. On the VIC 20, change statement 9 to **9 WIDTH=22.**

```
LIST
2 REM  BOUNCING BALL
5 PRINT"QQQ";
9 WIDTH=40
10 FOR I=1 TO WIDTH-1:PRINT" ●";:GOSU
50:NEXT
20 FOR I=1 TO WIDTH-1:PRINT" ●";:GO
8 50:NEXT
30 GOTO 10
50 IF PEEK(197)=64 THEN 50
60 RETURN
READY.

RUN
```

Figure 14.3 PEEK(197) will be equal to 64 if no key is being pressed.

POKE

Whereas PEEK allows you to read the data value in a particular memory location, the statement **POKE** *addr,data* allows you to store the value "*data*" in the memory location "*addr*". For example, type these statements as shown in Figure 14.4:

POKE 990,75
?PEEK(990)

Note that **PEEK(990)** verifies that you actually stored the value 75 in memory location 990.

Figure 14.4 POKE 990,75 stores the value 75 in memory location 990.

```
POKE 990,75

READY.
?PEEK(990)
 75

READY.
```

On a VIC 20 as it comes from the factory, the 506 memory locations 7680-8185 are special locations called the TV RAM. These 506 locations contain the special codes corresponding to the 506 characters (23×22) that are displayed on the VIC's video screen. (A blank is a character with its own special code.) If you have a VIC 20 with a memory expansion, then the TV RAM is automatically changed to the 506 locations 4096-4601. On the Commodore 64, the TV RAM is contained in the 1000 locations from 1024 to 2023. These 1000 locations contain the special codes corresponding to the 1000 characters (25×40) that are displayed on the Commodore 64's video screen.

On an unmodified VIC 20, memory location 7680 corresponds to the upper left-hand corner of the screen, and the memory addresses increase along each screen row. On a VIC 20 with a memory expansion, the situation is the same except that memory location 4096 corresponds to the upper left-hand corner of the screen. Similarly, on a Commodore 64, memory location 1024 corresponds to the upper left-hand corner of the screen, and the memory addresses increase along each screen row.

In each case, if you POKE a value to one of the addresses in the TV RAM, the character corresponding to this value will appear on the screen. To see this, use your cursor controls to scroll the screen until a nonblank character can be seen in the upper left-hand corner. Then move the cursor to the middle of the screen and type whichever of the following is appropriate:

POKE 7680,83 (if you have an unmodified VIC 20)
POKE 4096,83 (if you have a VIC 20 with a memory expansion
POKE 1024,83 (if you have a Commodore 64)

You should now see the heart graphic character in the upper left-hand corner of the screen. This is because 83 is the special code for a heart, and you POKEd it into the first location of the TV RAM. We will see how this technique can be used to draw complete pictures later in this chapter.

The reason we began the above by moving a nonblank character to the upper left-hand corner of the screen is to make certain the heart character would appear when we POKEd its special code there. If the upper left-hand corner had been blank to start with, POKEing the code for a heart would not have changed anything. This is because the way the Commodore 64/VIC 20 makes a screen cell blank is to change the character color for that cell to be the color of the background of the screen. Thus, changing only the code of the character has no effect. To see this, try poking a heart character into the upper left-hand corner when it is initially blank.

On a VIC 20 without a memory expansion, the 506 memory locations 38400-38905 are special locations called the COLOR RAM. These 506 locations contain special codes that determine the color of the corresponding 506 characters that are displayed on the VIC's video screen. Location 38400 corresponds to the upper left-hand corner of the screen, and the memory addresses increase along each screen row. On a VIC 20 with a memory expansion, the COLOR RAM is automatically moved to the 506 memory locations 37888-38393. In this case, location 37888 corresponds to the upper left-hand corner of the screen. On the Commodore 64, the COLOR RAM is contained in the 1000 locations from 55296 to 56295. In this case, location 55296 corresponds to the upper left-hand corner of the screen, and the memory addresses increase along each screen row.

On a VIC 20, each character on the screen will be one of eight colors depending upon the code stored in the corresponding cell of the COLOR RAM. The colors and their codes are shown in Figure 14.5a. On a Commodore 64, each character on the screen will be one of sixteen colors depending upon the code stored in the corresponding cell of the COLOR RAM. The colors and their codes for the Commodore 64 are shown in Figure 14.5b.

Figure 14.5a Commodore Standard Eight-Color Codes.

COLOR	POKE CODE
BLACK	0
WHITE	1
RED	2
CYAN	3
PURPLE	4
GREEN	5
BLUE	6
YELLOW	7

Figure 14.5b *Commodore Standard Sixteen-Color* Codes.

COLOR	POKE CODE
BLACK	0
WHITE	1
RED	2
CYAN	3
PURPLE	4
GREEN	5
BLUE	6
YELLOW	7
ORANGE	8
BROWN	9
LT. RED	10
GRAY 1	11
GRAY 2	12
LT. GREEN	13
LT. BLUE	14
GRAY 3	15

The color of any character on the screen can be changed by POKEing the corresponding location in the COLOR RAM with a color code. To see this, use your cursor controls to scroll the screen until a nonblank character can be seen in the upper left-hand corner. Then move the cursor to the middle of the screen and type whichever of the following is appropriate:

POKE 38400,5 (if you have an unmodified VIC 20)
POKE 37888,5 (if you have a VIC 20 with a memory expansion)
POKE 55296,5 (if you have a Commodore 64)

The color of the character in the upper left-hand corner of the screen should now be green. If you have a VIC 20, repeat this exercise and POKE each number between 0 to 7 into either location 38400 or 37888, whichever is appropriate, and record the results. If you have a Commodore 64, POKE each number between 0 and 15 into locations 55296 and record the results. In each case, you will make the character in the upper left-hand corner turn all possible colors.

It is also possible to modify both the border and background colors of the screen by POKEing values into appropriate memory locations. In the VIC 20 the memory location 36879 controls both the border and background colors. In this case, the border color can be any of the eight colors shown in Figure 14.5a and the color codes shown there apply. The background color, however, can be any of the sixteen colors shown in Figure 14.5b and the color codes shown in Figure 14.5b apply. To choose a combination of border and background colors, let BO be the appropriate color code 0-7 for the border as shown in Figure 14.5a and let BK be the appropriate color code 0-15 for the background as shown in Figure 14.5b. We choose the combination of colors with the statement

POKE 36879,16*BK+BO+8

To see this, type in the following program and run it, entering appropriate color codes from the keyboard.

```
10 INPUT BK,BO
20 POKE 36879,16*BK+BO+8
30 GOTO 10
```

On the Commodore 64, memory location 53280 controls the border color and memory location 53281 controls the background color of the screen. Either color may be any of the sixteen colors shown in Figure 14.5b and the color codes shown there apply. To see

this, type in the following program and run it, entering appropriate color codes from the keyboard.

```
10 INPUT BK,BO
20 POKE 53280,BO:POKE 53281,BK
30 GOTO 10
```

EXERCISE 14-1
Write a program for your computer that will blank the screen and then display in sequence all possible combinations of border and background colors. Introduce a suitable viewing delay using a FOR...NEXT loop.

EXERCISE 14-2
Write a program for your computer that will fill the screen with graphic hearts and display them with a randomly-chosen combination of border, background, and character colors (using the same color for all the hearts).

EXERCISE 14-3
To the program of Exercise 14-2, add a loop that 50 times will randomly choose a combination of border, background, and character colors and display them.

ALTERNATE CHARACTER SET

When you press a letter key, the Commodore 64/VIC 20 will display a capital letter. When you press the same key while holding the SHIFT key down, the Commodore 64/VIC 20 will display one of the graphic symbols. This is the standard character set that is available when you power up the Commodore 64/VIC 20.

The Commodore 64/VIC 20 also contains an alternate character set that produces lower case letters instead of graphic symbols. You can change to this alternate character set from the keyboard by pressing the LOGO and SHIFT keys together. Pressing the LOGO and SHIFT keys together again will change back to the standard character set. Start with a screen containing lots of characters and try this.

You can also change to the alternate character set by typing

POKE 36869,242 (if you have an unmodified VIC 20)

POKE 36869,194 (if you have a VIC 20 with a memory expansion)

POKE 53272,23 (if you have a Commodore 64)

To return to the standard character set, type

POKE 36869,240 (if you have an unmodified VIC 20)

POKE 36869,192 (if you have a VIC 20 with a memory expansion)

POKE 53272,21 (if you have a Commodore 64)

In order to illustrate this, clear the screen and if you have an unmodified VIC 20 type

```
POKE 36869,242
? "this is lower case"
```

and if you have a VIC 20 with a memory expansion, type:

```
POKE 36869,194
? "this is lower case"
```

If you have a Commodore 64, clear the screen and type:

```
POKE 53272,23
? "this is lower case"
```

The result in the case of the Commodore 64 is shown in Figure 14.6.

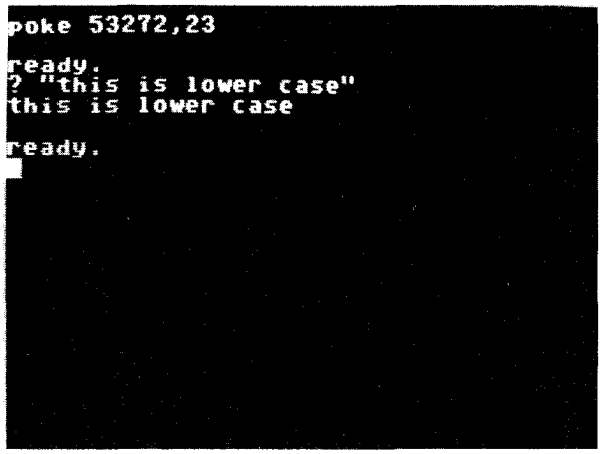


Figure 14.6 POKE 53272,23 changes a Commodore 64 to the alternate character set (lower case).

Now, type whichever of the following is appropriate:

POKE 36869,240 (if you have an unmodified VIC 20)

POKE 36869,192 (if you have a VIC 20 with a memory expansion)

POKE 53272,21 (if you have a Commodore 64)

Figure 14.7 shows what happens on the Commodore 64. Note that this changes back to the standard character set, and the lower case letters that you typed are changed to upper case. This shows that the same screen codes are interpreted differently depending upon whether the computer is in graphics or lower case

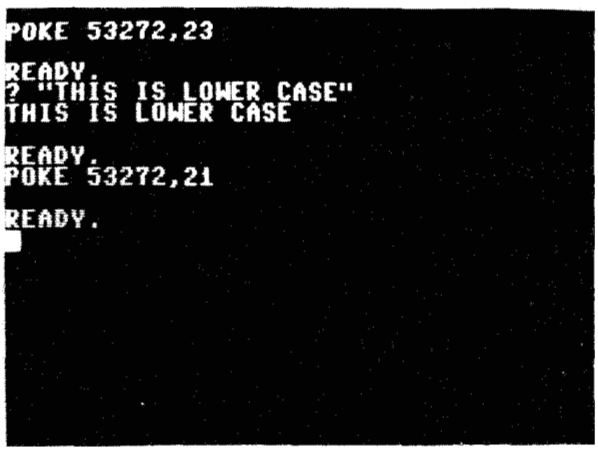


Figure 14.7 POKE 53272,21 changes a Commodore 64 back to the standard character set (graphics).

character mode. The same result happens on the VIC 20.

The Commodore 64/VIC 20 has two other character sets that you are already familiar with. To see this, make sure the screen has lots of characters and type whichever of the following is appropriate:

POKE 36869,241 (if you have an unmodified VIC 20)

POKE 36869,193 (if you have a VIC 20 with a memory expansion)

POKE 53272,22 (if you have a Commodore 64)

What character set do you see now? You should see the standard graphics characters in reverse video. Next, type whichever of the following is appropriate:

POKE 36869,243 (if you have an unmodified VIC 20)

POKE 36869,195 (if you have a VIC 20 with a memory expansion)

POKE 53272,24 (if you have a Commodore 64)

What character set do you see now?

New Graphic Symbols

When you use the alternate character set, the graphic symbols on the special character keys and the left graphic symbols on the letter keys are still available. Most of them are the same as they were before, but some new ones have been added. The four new graphic symbols are shown in Figure 14.8 and can be displayed by typing:

SHIFT π

LOGO *

SHIFT £

SHIFT @

Try these. Remember you must first enter LOGO and SHIFT together or type **POKE 36869,242** (**POKE 36869,194** if you have a memory expansion) on the VIC 20 or **POKE 53272,23** on the Commodore 64 in order to see these.

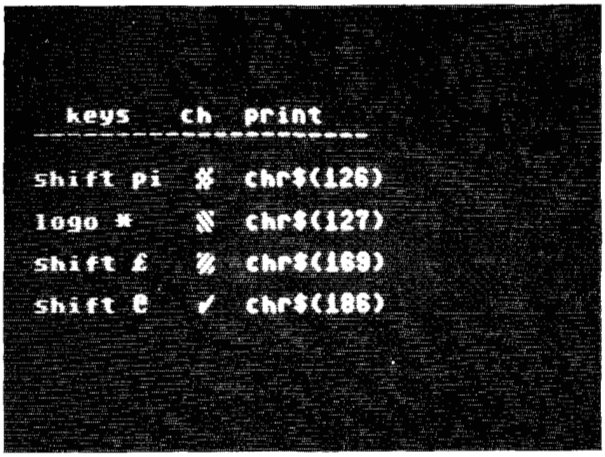
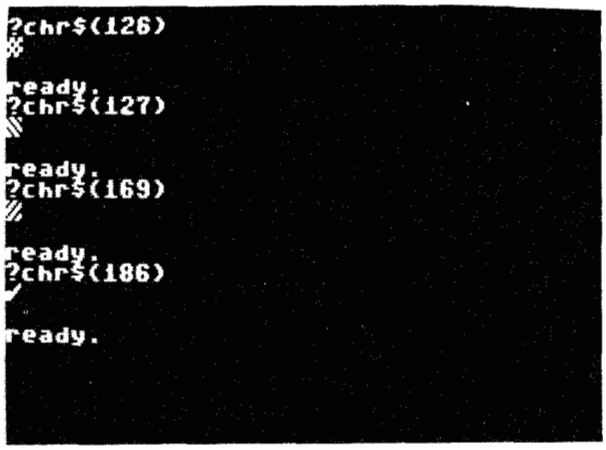


Figure 14.8 Four new graphic symbols are available in the alternate character set.

The ASCII codes for these four graphic symbols are 126, 127, 169, and 186. The CHR$ function can also be used to display these (or any other) graphic symbols, as shown in Figure 14.9.

Figure 14.9 The function CHR$(A) can be used to display a character with ASCII code A.



The new graphic symbols with ASCII codes 126, 127, and 169 are particularly useful for making bar graphs. We will use these three symbols for inflation, unemployment, and growth in the "economy" bar graph shown in Figure 10.30 for the Commodore 64 and Figure 10.31 for the VIC 20. We can make this change by changing the definitions of the graphic symbols in lines 15 and 16 of the program in Figure 10.27, as shown in Figure 14.10a for the VIC 20 and

Figure 14.10a Main program for plotting economy bar graph on the VIC 20 using new graphic symbols.

```
10 REM  THE ECONOMY
11 PRINT"□"
13 POKE 36869,242
15 I1$=CHR$(126):U1$=CHR$(169):G1$=CHR$(127):M1$="▒ ■"
16 I2$=I1$:M2$=M1$:M3$="▒■"
20 DATA 4.7,7.8,4.8,2.8
30 DATA 6.8,7,5.8,4.5
40 DATA 9,6,4.9,3.4
50 DATA 13.3,5.8,0.8,-.8
65 X=0
70 FOR J=1 TO 4
80 GOSUB 200:REM  PLOT 4 BARS
90 NEXTJ

READY.
```

Figure 14.10b Main program for plotting economy bar graph on the Commodore 64 using new graphic symbols.

```
10 REM  THE ECONOMY
11 PRINT"□"
13 POKE 53272,23
15 I1$=CHR$(126):U1$=CHR$(169):G1$=CHR$(127):M1$="▒ ■"
16 I2$=I1$:U2$=U1$:G2$=G1$:M2$="■":M3$="▒■"
20 DATA 4.7,7.8,4.8,2.8
30 DATA 6.8,7,5.8,4.5
40 DATA 9,6,4.9,3.4
50 DATA 13.3,5.8,0.8,-.8
60 DATA 18.2,6.2,1.7,-.5
65 X=0
70 FOR J=1 TO 5
80 GOSUB 200:REM  PLOT 4 BARS
90 NEXTJ
100 GOSUB 600:REM  PRINT HEADING & SCALE
150 GOTO 150

READY.
```

Figure 14.10b for the Commodore 64. Note that we also had to add the line numbered 13 in each case in order to change to the alternate character set. If you have a VIC 20 with a memory expansion, change this line to read **13 POKE 36869,194.**

We must also change lines 650-680 in Figure 10.29 as shown in Figure 14.11 so as to remove the character color symbols in each of these statements. The result of running these new programs is shown in Figure 14.12 for the VIC 20 and Figure 14.13 for the Commodore 64. Note that the letters in the caption are lower case even though they are upper case in the program listings. Again, this is due to the fact that we have changed to the lower case character set and the same character codes are interpreted differently in graphics and lower case modes. In particular, the screen codes for upper case letters in graphics mode become the screen codes for lower case letters in lower case mode.

Figure 14.11 Lines 650-670 have been changed so as to remove the color character symbols.

```
630 PRINT"■■THE ECONOMY"
640 PRINT
650 PRINT I1$;I1$;" INFLATION"
660 PRINT U1$;U1$;" UNEMPLOYMENT"
670 PRINT G1$;G1$;" GROWTH IN GNP"
680 PRINT M1$;M1$;" PERSONAL INCOME"

READY.
```

POKING GRAPHIC PICTURES

Recall that the Commodore 64 screen has 1000 print locations organized as a 25×40 grid (see Figure 9.4). As you learned earlier, each of these 1000 print locations is associated with a particular memory address in the Commodore 64's TV RAM. The

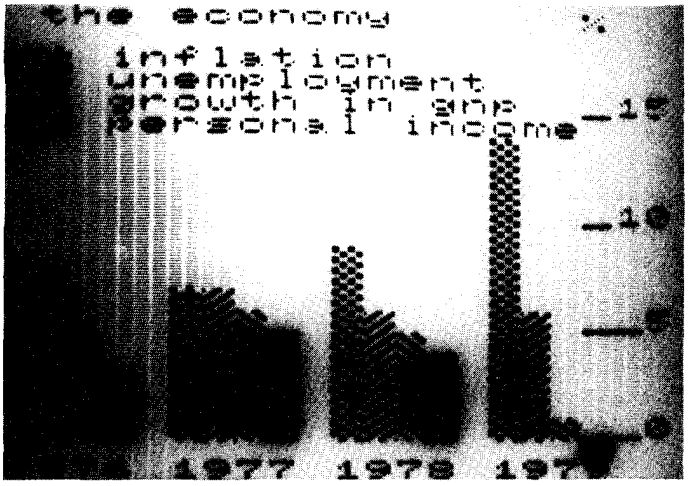


Figure 14.12 Bar graph of economic data on the VIC 20 using the new graphic symbols.

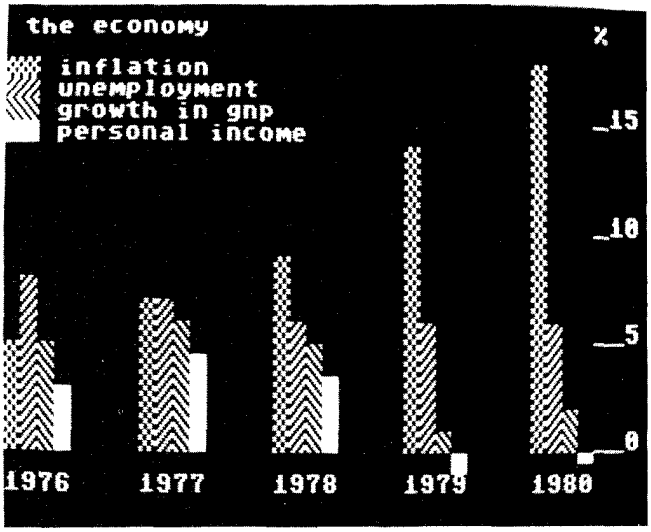


Figure 14.13 Bar graph of economic data on the Commodore 64 using the new graphic symbols.

memory address contains the code for the character displayed at the corresponding screen location. The upper left-hand corner of the screen is address 1024, and the addresses increase by one along each row as shown in Figure 14.14.

Since each row on the screen contains forty print locations, the difference between the address of one screen location and that of the location directly below it will be 40. By numbering the rows Y from 0 to 24 and the columns X from 0 to 39, the address of the screen location X,Y can be calculated from the equation

$$\text{Address} = 1024 + 40 \times Y + X$$

For example, as shown in Figure 14.14, the screen location at X=25, Y=5 has the address

$$\begin{aligned}\text{Address} &= 1024 + 40 \times 5 + 25 \\ &= 1024 \times 200 + 25 \\ &= 1249\end{aligned}$$

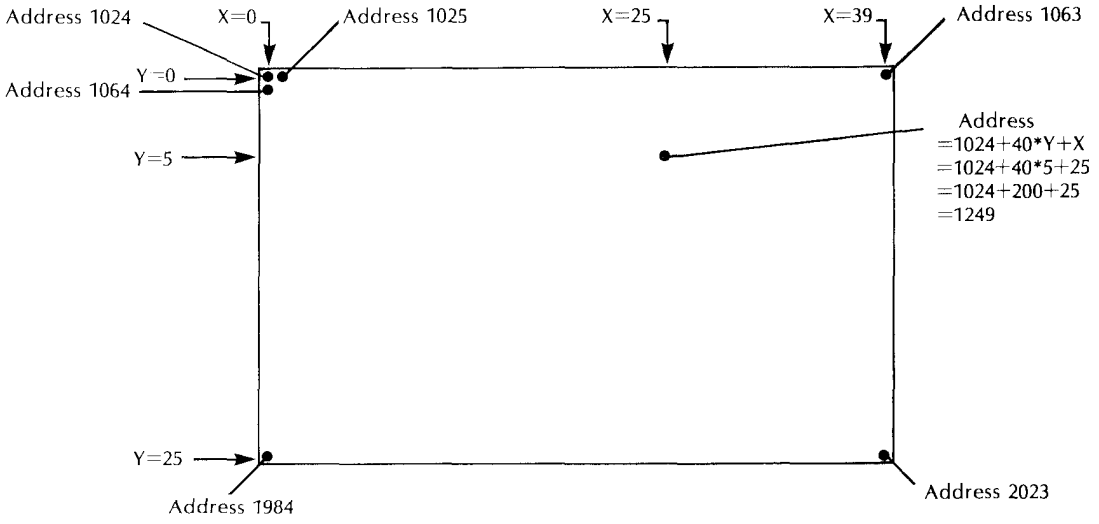
Recall that the VIC 20 screen has 506 print locations organized as a 23 × 22 grid. Each of the 506 print locations has an associated memory address in the VIC 20's TV RAM. This memory address contains the code for the character to be displayed at the corresponding screen location. The upper left-hand corner of the screen is address 7680 or 4096 depending upon whether or not the VIC 20 has a memory expansion. Since each row on the screen contains twenty-two print positions, the difference between the address of one screen location and the location directly below it is 22. By numbering the rows Y from 0 to 22 and the columns X from 0 to 21, the address of the screen location can be calculated from the equation

$$\text{Address} = 7680 + 22 \times Y + X$$

for an unmodified VIC 20 or

$$\text{Address} = 4096 + 22 \times Y + X$$

Figure 14.14 Each print location on the Commodore 64 screen has a unique memory address associated with it.



if the VIC 20 has a memory expansion. These formulas and that for the Commodore 64 can be combined to read

$$\text{Address} = \text{BASE} + \text{WIDTH} \times Y + X$$

where

$$\text{BASE} = \begin{cases} 7680 \text{ for an unmodified VIC 20} \\ 4096 \text{ for a VIC 20 with a memory expansion} \\ 1024 \text{ for a Commodore 64} \end{cases}$$

and

$$\text{WIDTH} = \begin{cases} 22 \text{ for a VIC 20} \\ 40 \text{ for a Commodore 64} \end{cases}$$

PEEK/POKE Codes

If you know the X and Y coordinates of a location where you want to print a character, you can calculate the address using the above formula and then POKE the special character code at this address. However, the special character code you must POKE is not the ASCII code, but a different PEEK/POKE code. To see this, clear the screen and type a heart graphic followed by a spade graphic in the upper left-hand corner of the screen using the keyboard. Then press RETURN and type these statements, as shown in Figure 14.15a.

BASE=address given above for your computer
?ASC("SHIFT S")
?PEEK(BASE)
POKE BASE+1,83

The ASCII code for the heart is 211, while the PEEK/POKE code is 83. POKEing an 83 to location BASE+1 will change the spade graphic to a second heart as shown in Figure 14.15b.

To find out what the PEEK/POKE codes are for all of the keys, you can write a program that will print whichever key you press at a known screen location and then PEEK that address. A program to do this is shown in Figure 14.16. The values of BASE and WIDTH shown in line 12 are for a Commodore 64. The values must be changed as described above for an unmodified VIC 20 or a VIC 20 with a memory expansion.

For the given values of X and Y, line 40 moves the cursor to that screen position using the usual "Move to X,Y" subroutine and calculates this screen address, SA. Line 50 is the usual GET loop waiting for a key to be pressed. Line 60 prints the character that was pressed and then prints the PEEK/POKE code by PEEKing the character that was just displayed. The nested FOR...NEXT loops in lines 20-70 will cause

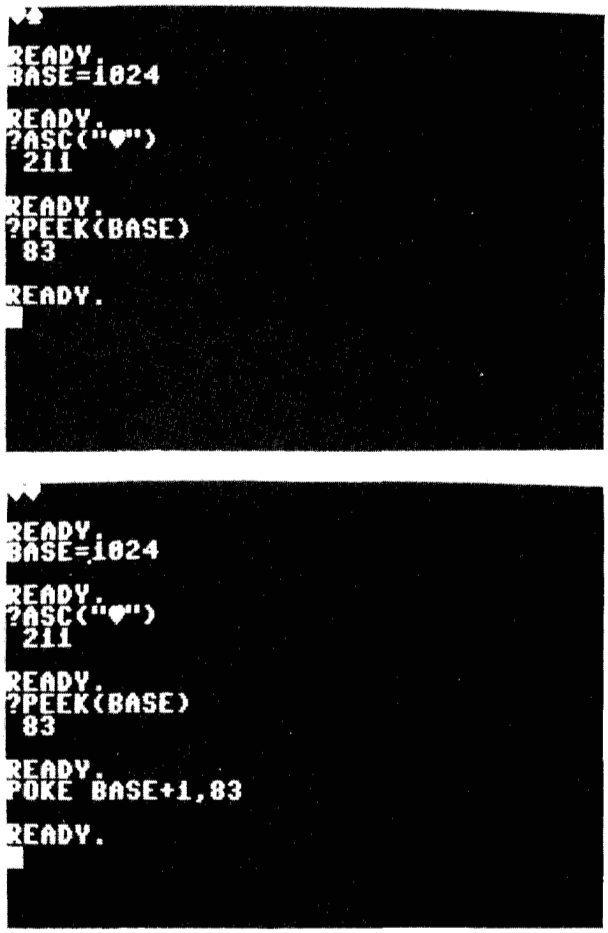


Figure 14.15 The PEEK/POKE codes are not the same as the ASCII codes.

three values to be printed on a VIC 20 or five values on a Commodore 64 on each of eleven rows before the program stops. A sample run of this program is shown in Figure 14.17. A complete list of all PEEK/POKE codes is given in Appendix C.

Figure 14.16 Program to print the PEEK/POKE code of whichever key is pressed.

```
10 REM  PRINT POKE CODES ON SCREEN
12 BASE=1024:WIDTH=40
15 PRINT"■";
20 FOR Y=1 TO 21 STEP 2
30 FOR X=0 TO WIDTH-1 STEP 8
40 GOSUB 500:SA=BASE+WIDTH*Y+X
50 GET A$:IF A$="" THEN 50
60 PRINT A$;PEEK(SA);
70 NEXTX:NEXTY
80 GOTO 80
500 REM  MOVE TO X,Y
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT SPC(X);:RETURN

READY.
```

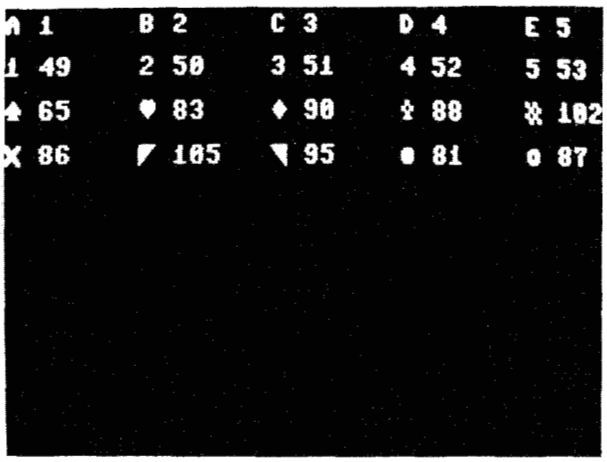


Figure 14.17 Some PEEK/POKE codes found by running the program of Figure 14.16 (see complete list of PEEK/POKE codes in Appendix C).

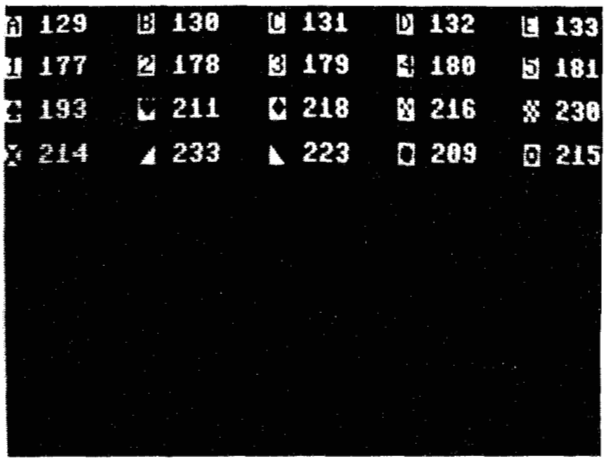


Figure 14.19 Some reverse video PEEK/POKE codes found by running the program of Figure 14.18 (see complete list of PEEK/POKE codes in Appendix C).

One difference between the ASCII codes and the PEEK/POKE codes involves reverse video characters. The RVS and OFF keys have ASCII codes 18 and 146, respectively. Therefore, CHR$(18) and CHR$(146) can be used in a PRINT statement to turn the reverse video on and off. However, all reverse video characters have unique PEEK/POKE codes and can therefore be POKEd directly on the screen. Reverse video characters do not have unique ASCII codes and can be PRINTed using the standard ASCII codes only after the reverse video has been turned on.

To find the reverse video PEEK/POKE codes, change line 60 in Figure 14.16 to turn the reverse video on before A$ is printed and off after it is printed, as shown in Figure 14.18. It turns out that the reverse video PEEK/POKE code is always the normal PEEK/POKE code plus 128. This can be seen in the sample run of the program shown in Figure 14.19.

Figure 14.18 Program to print the reverse video PEEK/POKE code of whichever key is pressed.

```
10 REM  PRINT POKE CODES ON SCREEN
12 BASE=1024:WIDTH=40
15 PRINT"■";
20 FOR Y=1 TO 21 STEP 2
30 FOR X=0 TO WIDTH-1 STEP 8
40 GOSUB 500:SA=BASE+WIDTH*Y+X
50 GET A$:IF A$="" THEN 50
60 PRINT "■";A$;"■";PEEK(SA);
70 NEXTX:NEXTY
80 GOTO 80
500 REM  MOVE TO X,Y
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT SPC(X);:RETURN

READY.
```

How to POKE a Picture

In order to POKE a picture on the screen, you must POKE each graphic symbol into the appropriate screen location in the TV RAM and POKE the desired color into the corresponding location of the COLOR RAM. Suppose that you want to draw a rectangular picture that is H rows high and W columns wide, as shown in Figure 14.20. The PEEK/POKE codes for each element in the picture are stored in a two-dimensional integer array P%(I,J) where I has values between 0 and H-1 and J has values between 0 and W-1.

The upper left-hand corner of the picture is to be located at position X,Y on the screen. Therefore, the base address BS of this element in the picture is given by **BS = BASE + WIDTH * Y + X,** where

$$\text{BASE} = \begin{cases} 7680 \text{ for an unmodified VIC 20} \\ 4096 \text{ for a VIC 20 with a memory expansion} \\ 1024 \text{ for a Commodore 64} \end{cases}$$

$$\text{WIDTH} = \begin{cases} 22 \text{ for a VIC 20} \\ 40 \text{ for a Commodore 64} \end{cases}$$

The screen address A of any element in the picture can be calculated relative to this base address by using the formula **A = BS + WIDTH*I + J** as shown in Figure 14.20. Drawing the picture then involves POKEing all values of P%(I,J) into the appropriate screen addresses.

However, we still need to POKE the correct colors for the picture. To do so, we note that each location of the TV RAM has an associated location in the COLOR RAM. The code in the former determines the character displayed at the corresponding location on the screen and the code in the latter determines its

color. Let CBASE be the starting address in the COLOR RAM. That is, let

$$\text{CBASE} = \begin{cases} 38400 \text{ for an unmodified VIC 20} \\ 37888 \text{ for a VIC 20 with a memory expansion} \\ 55296 \text{ for a Commodore 64} \end{cases}$$

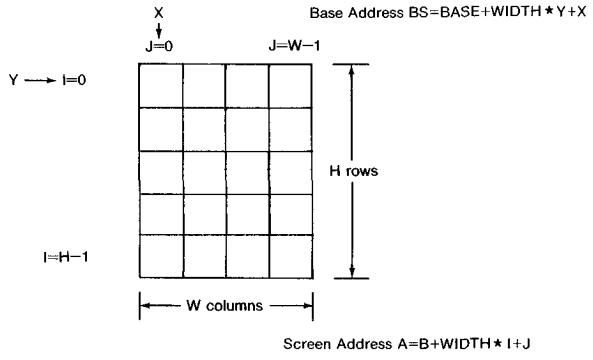
If A is a location in the TV RAM and B is the corresponding location in the COLOR RAM, we see that

$$\begin{aligned} B - \text{CBASE} &= \text{offset from start of COLOR RAM} \\ &= \text{offset from start of TV RAM} \\ &= A - \text{BASE} \end{aligned}$$

In other words, B = A + CBASE − BASE. Therefore, when we POKE a screen code into address A of the TV RAM, we can control the color of the same screen location by POKEing the desired color code into location B = A + CBASE − BASE of the COLOR RAM.

The subroutine of Figure 14.21 POKEs an H × W picture whose screen codes are stored in an integer array P%(I,J) into the TV RAM and, at the same time, POKES a fixed color number C into every picture location in the COLOR RAM. Note that the picture can be moved anywhere on the screen by changing the values of X and Y. The values shown for BASE and CBASE are for a Commodore 64. They must be changed as discussed above for an unmodified VIC 20 or a VIC 20 with a memory expansion. The color number C=5 corresponds to green.

Figure 14.20 The array P%(I,J) contains the PEEK/POKE codes for a picture of width W and height H.



As an example of using this subroutine we will POKE the king shown in Figure 4.23 in Chapter 4. The first step is to look up the PEEK/POKE codes for each element in the picture from the tables in Appendix C. Enter these codes into a matrix in the shape of the picture, as shown in Figure 14.22.

Each row in Figure 14.22 can then be stored in a DATA statement and read into the array P%(I,J) as shown in lines 110-130 of the program in Figure 14.23. The DATA statements are shown in Fig. 14.26a. Line 140 draws this picture at location X=10, Y=5, by calling the subroutine shown in Figure 14.21. The result of running this program is shown in Figure 14.24.

Figure 14.21 Subroutine that POKEs an H × W picture P%(I,J) at location X,Y.

```
4000 REM POKE H X W PICTURE P%(I,J) AT X,Y
4005 BASE=1024:WIDTH=40:CBASE=55296:C=5
4010 BS=BASE+WIDTH*Y+X
4020 FOR I=0 TO H-1
4030 FOR J=0 TO W-1
4040 A=BS+WIDTH*I+J:POKE A,P%(I,J)
4050 B=A+CBASE-BASE:POKE B,C
4060 NEXTJ:NEXTI:RETURN

READY.
```

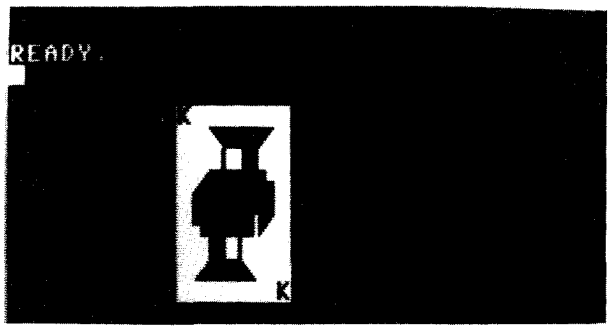
139	160	160	160	160	160	160
160	160	223	32	32	223	160
160	160	231	160	32	160	160
160	105	32	32	32	124	160
160	32	32	32	32	32	160
160	123	32	32	103	233	160
160	160	32	231	160	160	160
160	105	32	32	95	160	160
160	160	160	160	160	160	139

Figure 14.22 The PEEK/POKE codes for the king.

Figure 14.23 Main program to POKE a king.

```
100 REM POKE A KING
105 PRINT"J"
107 H=9:W=7:DIM P%(H-1,W-1)
110 FOR I=0 TO H-1
120 FOR J=0 TO W-1
130 READ P%(I,J):NEXTJ:NEXTI
140 X=10:Y=5:GOSUB 4000
150 END

READY.
```

Figure 14.24 Result of running the program in Figure 14.23.



Multiple pictures can be stored in a *three-dimensional* array with one picture being stored in each plane of the array, as shown in Figure 14.25. The PEEK/POKE codes used to draw a king, queen, and jack are shown in Figure 14.26.

The subroutine shown in Figure 14.21 must be modified to POKE the values in P%(I,J,K) as shown in Figure 14.27. The main program shown in Figure 14.28 reads all of the data in Figure 14.26 and then draws the jack (K=2) in line 140, the queen (K=1) in line 150, and the king (K=0) in line 160. Line 105 prints the message **"I'M THINKING!"** because it takes a few seconds to read all of the DATA statements before any picture is displayed. The result of running the program in Figure 14.28 is shown in Figure 14.29.

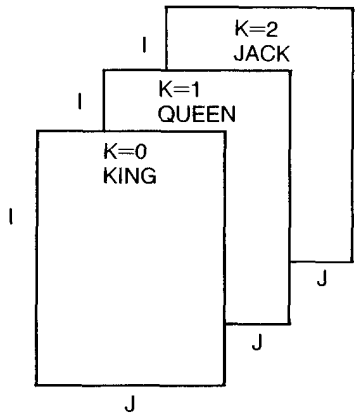


Figure 14.25 The three-dimensional array P%(I,J,K) can store a different picture for each value of K.

Figure 14.26 (a) PEEK/POKE codes for the king stored in P%(I,J,0). (b) PEEK/POKE codes for the queen stored in P%(I,J,1). (c) PEEK/POKE codes for the jack stored in P%(I,J,2).

(a)
```
200 DATA 139,160,160,160,160,160,160
210 DATA 160,160,223,32,32,233,160
220 DATA 160,160,231,160,32,160,160
230 DATA 160,105,32,32,32,124,160
240 DATA 160,32,32,32,32,32,160
250 DATA 160,123,32,32,103,233,160
260 DATA 160,160,32,231,160,160,160
270 DATA 160,105,32,32,95,160,160
280 DATA 160,160,160,160,160,160,139

READY.
```

(b)
```
300 DATA 145,160,160,160,160,160,160
310 DATA 160,160,233,208,160,160,160
320 DATA 160,105,221,160,244,160,160
330 DATA 160,32,32,32,32,251,160
340 DATA 160,32,32,32,32,32,160
350 DATA 160,252,32,32,32,32,160
360 DATA 160,160,246,229,212,233,160
370 DATA 160,160,231,204,105,160,160
380 DATA 160,160,160,160,160,160,145

READY.
```

```
400 DATA 138,160,160,160,160,160,160
410 DATA 160,160,223,32,233,160,160
420 DATA 160,229,246,231,160,160,160
430 DATA 160,229,32,32,32,231,160
440 DATA 160,229,32,32,32,231,160
450 DATA 160,229,32,32,32,231,160
460 DATA 160,160,160,229,225,231,160
470 DATA 160,160,105,32,95,160,160
480 DATA 160,160,160,160,160,160,138

READY.
```

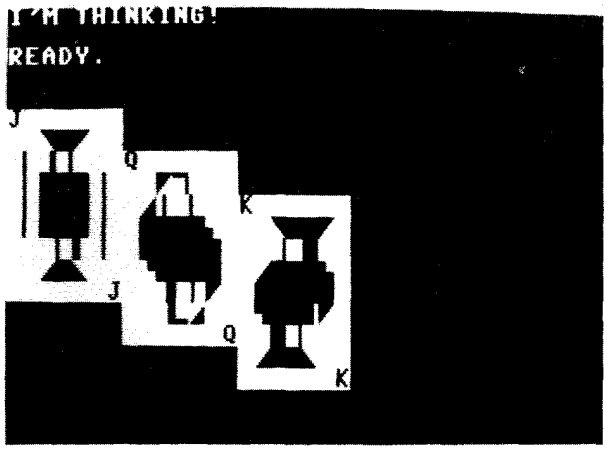
Figure 14.27 Subroutine to POKE picture K in the array P%(I,J,K).

```
4000 REM POKE HXW PICTURE P%(I,J,K) AT X,Y
4005 BASE=1024:WIDTH=40:CBASE=55296:C=5
4010 BS=BASE+WIDTH*Y+X
4020 FOR I=0 TO H-1
4030 FOR J=0 TO W-1
4040 A=BS+WIDTH*I+J:POKE A,P%(I,J,K)
4050 B=A+CBASE-BASE:POKE B,C
4060 NEXTJ:NEXTI:RETURN

READY.
```

Figure 14.28 Main program that displays a jack, queen, and king.

```
100 REM  POKE A JACK,QUEEN,& KING
105 PRINT"■I'M THINKING!"
107 H=9:W=7: DIM P%(H-1,W-1,2)
108 FOR K=0 TO 2
110 FOR I=0 TO H-1
120 FOR J=0 TO W-1
130 READ P%(I,J,K):NEXTJ:NEXTI:NEXTK
140 X=0:Y=5:K=2:GOSUB 4000
150 X=7:Y=7:K=1:GOSUB 4000
160 X=14:Y=9:K=0:GOSUB 4000
180 END

READY.
```

Figure 14.29 Result of running the program in Figure 14.28.



MAKING SOUNDS WITH THE VIC 20

The effectiveness of your programs can be greatly enhanced by adding sound effects. Both the VIC 20 and Commodore 64 can produce a variety of sounds through the speaker of the television set used for display or an auxiliary sound system. The sounds are controlled by PEEKing and POKEing addresses in the memory. However, the sound systems of the two computers are quite different. In fact, the sound capability of the Commodore 64 is much more sophisticated than that of the VIC 20. Consequently, we will discuss making sounds on the VIC 20 in this section and take up making sounds on the Commodore 64 in the next section. Commodore 64 users can skip this section and VIC 20 users can skip the next.

The VIC 20 has four voices, each having approximately a three octave range. The waveform of the voices 0, 1, and 2 is a squarewave (pulsing on and off) which produces a woodwind/organ sound. Voice 3 has a white noise waveform which makes a hissing sound that is useful for percussion sounds and sound effects.

The pitch of voices 0, 1, 2, and 3 is determined by POKEing a value in the approximate range 130-250 into addresses 36874, 36875, 36876, and 36877, respectively, as shown in 14.30. Voices 0, 1, and 2 are staggered in pitch with each successive voice being one octave higher than the previous one. Thus, they can be thought of as tenor, alto, and soprano voices, respectively. By switching voices, it is possible to make music sounds over a five octave range. Only voice 0 can produce the lowest octave and only voice 2 can produce the highest.

The volume of all four voices is determined by POKEing a value between 0 and 15 into address 36878. For convenience, we have assigned the voice number 4 to volume in Figure 14.30 even though it is not really another voice.

Figure 14.30 POKE Addresses for Making Sounds on the VIC 20.

VOICE	SOUND	POKE LOC.
0	TENOR	36874
1	ALTO	36875
2	SOPRANO	36876
3	NOISE	36877
4	VOLUME	36878

Producing a Tone

Figure 14.31 contains a program for producing a series of single notes on the VIC 20. The subroutine at line 500 POKEs pitch P into voice V for a duration D. The values for V, P, and D are specified outside the subroutine. Line 15 calls this subroutine with V=4, P=15, and D=0. This serves to set the volume of all voices to the maximum possible value, 15. Line 20 allows the user to select values for V, P, and D from the keyboard. After playing the specified note, the main program loops back to line 20 so that another note can be selected.

In order to try out some notes, turn up the volume on your TV set and type in the program of Figure 14.31. In general, when choosing the sound parameters, V should be between 0 and 3, P between 130 and 250, and D between 1 and 5000. Choosing P=0 and D=0 will turn off voice V. Try producing pitches 130, 190, and 250 for duration 1000 on each voice in turn. Be sure to try voice 3.

Figure 14.31 Program to produce tones by voice V on pitch P for duration D.

```
10 REM MAKING SOUNDS
15 V=4:P=15:D=0:GOSUB 500
20 INPUT "VOICE, PITCH, AND DURATION";V,P,D
25 GOSUB 500
30 P=0:D=0:GOSUB 500
40 GOTO 20
500 REM MAKE NOTE V=VOICE P=PITCH D=DURATION
505 V1=36874+V:POKE V1,P
510 FOR I=1 TO D:NEXT
515 RETURN

READY.
```

Making Scales

The musical quality of the VIC 20 can better be judged by playing some scales with the three squarewave voices. The program of Figure 14.32 is an adaptation of the above tone-producing program for the purpose of playing musical scales. Note the subroutine for producing tones at line 500 is the same as the previous program.

Line 110 contains the values recommended by Commodore for an ascending C major scale in the middle octave of each voice. Line 120 contains the same values in reverse order to form a descending version of the same scale, followed by the value −1. The latter serves to mark the end of the song. Line 15 turns up the volume and line 20 allows the user to specify the voice V to be used and the duration DU of each note. The loop in lines 25-40 plays the scale using the data in lines 110 and 120.

Make the necessary modifications to the previous program to obtain the program of Figure 14.32. Then, execute the program with various values of V 0-2 and duration DU. In particular, try voices 0, 1, and 2 with duration 200. Also, just for fun, try voice 3. If you are musically trained, you may be disappointed by the accuracy of the pitches produced by the VIC 20. This is more or less an inherent problem with the VIC 20 since the range of audio frequencies possible is subdivided into only approximately 250-130 = 120 distinct values, i.e. the values we POKE. On the Commodore 64, this problem is overcome by subdividing the range of audio frequencies much finer. The result is that one has to POKE two values into two addresses in order to choose a pitch.

Figure 14.32 Program for playing musical scales on the VIC 20.

```
10 REM MAKING SCALES
15 V=4:P=15:D=0:GOSUB 500
20 INPUT "VOICE AND DURATION";V,DU
25 READ P:IF P<0 THEN RESTORE:GOTO 20
30 D=DU:GOSUB 500
35 P=0:D=0:GOSUB 500
40 GOTO 25
100 REM C MAJOR SCALE 2ND OCTAVE
110 DATA 195,201,207,209,215,219,223,225
120 DATA 225,223,219,215,209,207,201,195,-1
500 REM MAKE NOTE V=VOICE P=PITCH D=DURATION
505 V1=36874+V:POKE V1,P
510 FOR I=1 TO D:NEXT
515 RETURN

READY.
```

Producing Multiple Clicks

Let us now consider producing some simple sound effects using the same subroutine to "play pitch P on voice V for duration D." Figure 14.33 contains a program to produce N clicks with a time separation S at pitch P. Make the modifications necessary to create this program and try it with various pitches P, time separations S, and number of clicks N. In particular, try P=240, S=1, and N=60. This should give a beep every second for one minute. Check the beeps against your watch.

Figure 14.33 Program to produce N clicks with separation S and pitch P.

```
10 REM PRODUCE N CLICKS WITH SPACING S AND PITCH P
20 INPUT "ENTER  PITCH";PI
25 INPUT "ENTER SEPARATION (SEC)";S
30 INPUT "ENTER NUMBER OF CLICKS";N
35 V=4:P=15:D=0:GOSUB 500
40 V=2:D=1
45 FOR J=1 TO N
50 P=PI:GOSUB 500
60 P=0:GOSUB 500
65 FOR I=1 TO 730*S:NEXT I
70 NEXT J
75 GOTO 20
500 REM MAKE NOTE V=VOICE P=PITCH D=DURATION
505 V1=36874+V:POKE V1,P
510 FOR I=1 TO D:NEXT
515 RETURN

READY.
```

Producing a Phaser Noise

If you repeatedly call the tone subroutine of Figure 14.31 with different pitch values P, you can produce a variety of effects. For example, the program shown in Figure 14.34 produces "phaser" noises consisting of NC cycles of a sound in which the pitch varies from P1 to P2 in steps of DP.

To hear what this noise sounds like, execute the program and enter from the keyboard NC = 10, P1 =

Figure 14.34 Program to make a phaser noise.

```
10 REM PHASER NOISE
15 INPUT "ENTER NO. CYCLES";NC
20 INPUT "ENTER STARTING PITCH";P1
25 INPUT "ENTER ENDING PITCH";P2
30 INPUT "ENTER PITCH INCREMENT";DP
35 V=4:P=15:D=0:GOSUB 500
40 V=2:D=1
45 FOR J=1 TO NC
50 FOR P=P1 TO P2 STEP DP
55 GOSUB 500
60 NEXT P
65 NEXT J
70 P=0:GOSUB 500
75 GOTO 15
500 REM MAKE NOTE V=VOICE P=PITCH D=DURATION
505 V1=36874+V:POKE V1,P
510 FOR I=1 TO D:NEXT
515 RETURN

READY.
```

200, P2 = 130, and DP = −6. Note that the pitch varies from high to low each cycle. Try a variety of different values for NC, P1, P2, and DP. In particular, try values where P1 is less than P2 and DP is positive.

Producing a Siren Sound

A siren noise can be produced by repeatedly calling the tone subroutine, first with increasing values of P, and then with decreasing values of P. The program is shown in Figure 14.35.

In lines 15–35, the user is asked to choose values for the number of cycles NC, the starting pitch P1, the ending pitch P2, the pitch increment DP, and the holding time T, which controls the amount of time each pitch is held. The volume is turned up in line 40 and voice 2 and the duration or holding time is set in line 45. The inner loop in lines 55–65 creates a wail in ascending pitch and that in lines 70-80 produces one in descending pitch. The outer loop in lines 50 and 85 counts the cycles. Line 90 turns the voice off and line 95 loops to allow a new choice of parameters.

Run this program for the following values: NC = 5, P1 = 130, P2 = 180, DP = .3 and T = 1. This sounds like an old-fashioned ambulance siren. Try varying parameter values including the holding time T.

Adding Sound Effects to the "Plane with Phasers" Program

In Chapter 11 we developed a program that fired phasers from a fighter plane. We will add some sound effects to that program. The original program of Chapter 11 will be unchanged except for the subroutines for firing the left and right phasers.

The phaser-firing subroutines will be modified to fire a single "bullet" that makes a noise each time it moves. These modified subroutines are shown in Figure 14.36. Lines 235 and 335 produce a phaser noise by calling subroutine 1000. Lines 240 and 340 erase the previously plotted "bullet". The old subroutine at 400 that was used to erase the phaser is no longer needed.

Figure 14.35 Program to produce a siren sound.

```
10 REM SIREN NOISE
15 INPUT "ENTER NO. CYCLES";NC
20 INPUT "ENTER STARTING PITCH";P1
25 INPUT "ENTER ENDING PITCH";P2
30 INPUT "ENTER PITCH INCREMENT";DP
35 INPUT "ENTER HOLDING TIME";T
40 V=4:P=15:D=0:GOSUB 500
45 V=2:D=T
50 FOR J=1 TO NC
55 FOR P=P1 TO P2 STEP DP
60 GOSUB 500
65 NEXT P
70 FOR P=P2 TO P1 STEP -DP
75 GOSUB 500
80 NEXT P
85 NEXT J
90 P=0:GOSUB 500
95 PRINT:GOTO 15
500 REM MAKE NOTE V=VOICE P=PITCH D=DURATION
505 V1=36874+V:POKE V1,P
510 FOR I=1 TO D:NEXT
515 RETURN

READY.
```

Figure 14.36 Phaser-firing subroutines modified to produce noise.

```
200 REM FIRE LEFT PHASER
210 U=X-2
220 FOR V=Y TO 0 STEP -1
230 GOSUB 500:PRINT"#"
235 GOSUB 1000
240 GOSUB 500:PRINT" "
245 NEXT
250 RETURN
300 REM  FIRE RIGHT PHASER
310 U=X+2
320 FOR V=Y TO 0 STEP -1
330 GOSUB 500:PRINT"O"
335 GOSUB 1000
340 GOSUB 500:PRINT" "
345 NEXT
350 RETURN

READY.
```

The subroutine to make the phaser noise is shown in Figure 14.37. The program in Figure 14.34 has been modified to fit the plane program. Lines 1100-1110 turn on voice V at pitch P. Line 1000 turns up the volume and 1010 selects voice V = 2 at duration D = 1. The loop in lines 1020-1040 varies the pitch from 135 to 200 in steps of 20.

When this modified plane program is run and a phaser is fired, a single bullet is released, as shown in Figure 14.38, and it makes a noise as it moves along.

The discussion of the various sound subroutines for the VIC 20 presented in this section should enable you to add interesting sound effects to your program. How to incorporate music into a VIC 20 program is described in Chapter 15.

Figure 14.37 Subroutine to produce phaser sound for fighter plane.

```
1000 V0=4:P=15:D=0:GOSUB 1100
1010 V0=2:D=1
1020 FOR P=135 TO 200 STEP 20
1030 GOSUB 1100:NEXT P
1040 P=0:GOSUB 1100:RETURN
1100 V1=36874+V0:POKE V1,P
1110 RETURN

READY.
```

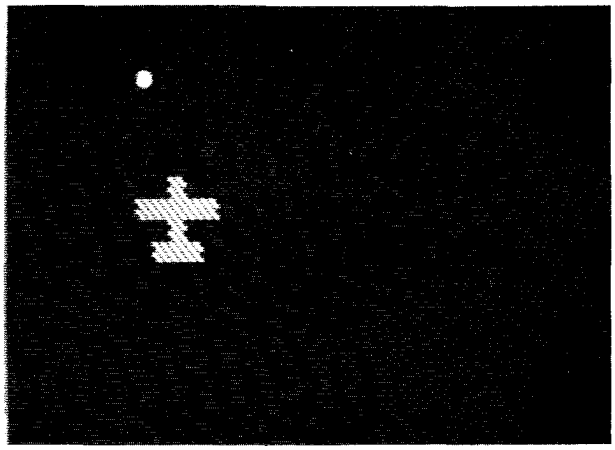


Figure 14.38 A single bullet is fired from the plane and makes noise as it moves.

MAKING SOUNDS WITH THE COMMODORE 64

The Commodore 64 has a controllable sound synthesizer capable of producing the sounds of certain musical instruments with considerable fidelity. It is also capable of a wide variety of sound effects. The sound can be produced through the television set used for display or through a separate sound system. The latter approach is far better because the audio channel of a television set is quite limited.

The Commodore 64 has three independent but otherwise identical voices, each capable of the same nine octave range. The sound of each voice has many programmable parameters. Each voice can independently have one of four different waveforms: triangle, sawtooth, pulse (square/on-off), or noise. In the pulsed waveform, the pulsewidth (relative "on" time versus "off" time) is controllable.

The average amplitude/volume of a note varies with time according to an "envelope" with four controllable parameters called attack, decay, sustain, and release (ADSR for short). In Figure 14.39, the average amplitude/volume envelope of a note is shown including the four phases corresponding to ADSR. Attack is the rate at which the note/sound reaches its maximum amplitude (volume). Decay is the rate the note/sound falls from the maximum amplitude to a lower "sustained" level. Release is the rate the amplitude falls from the sustained level back to nothing.

The parameters of sound are all controlled by POKEing values into the range of addresses 54272-54296. If we let RA = 54272 be the sound reference address, the first seven addresses starting with RA control voice 0, the next seven control voice 1, and the next seven control voice 2 as shown in Figure 14.40. Normally, the volume of all voices, the waveforms, the pulsewidths (where the pulsed waveforms are specified), the attack/decay rates, and the sustain level/release rate are all chosen once at the beginning of a program. The pitches/frequencies of notes are varied as each note is played. The on/off addresses (which happen also to specify the waveforms) are POKEd to turn the notes on and off.

The ranges of POKE values of various voice parameters are also shown in Figure 14.40. Note that the values of attack, decay, sustain, release, volume, and the high byte of pulsewidth vary only from 0 to 15. The waveform values 17, 33, 65, and 129 serve to turn on the triangle, sawtooth, pulse, and noise waveforms, respectively. The values 16, 32, 64, and 128 turn the same respective waveforms off.

Producing a Tone

Figure 14.41 contains a program for producing a series of single notes on voice 0 of the Commodore 64. The subroutine at line 500 POKEs pitch P on for duration

Figure 14.39 ADSR parameters of a note.

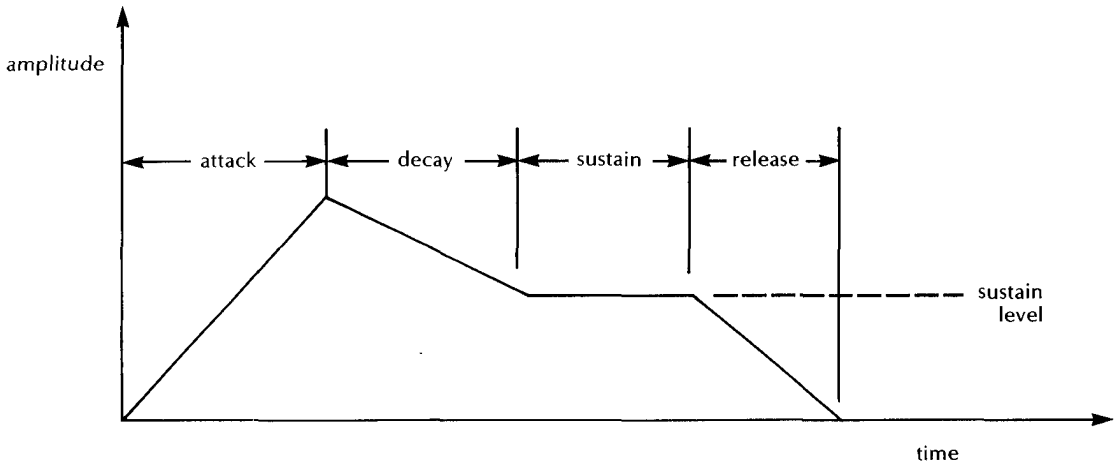


Figure 14.40 POKE addresses for producing sounds on the Commodore 64.

Addresses	Voices	POKE Values	Parameter
RA/RA+7/RA+14	0/1/2	0-255	low byte of pitch
RA+1/RA+8/RA+15	0/1/2	0-255	high byte of pitch
RA+2/RA+9/RA+16	0/1/2	0-255	low byte of pulsewidth
RA+3/RA+10/RA+17	0/1/2	0-15	high byte of pulsewidth
RA+4/RA+11/RA+18	0/1/2	16/17 triangle 32/33 sawtooth 64/65 pulse 128/129 noise	off/on and waveform
RA+5/RA+12/RA+19	0/1/2	0-15/0-15	attack/decay
RA+6/RA+13/RA+20	0/1/2	0-15/0-15	sustain/release
RA+24		0-15	volume of all voices

(RA = 54272)

D with waveform WF+1, where WF is either 16, 32, 64, or 128. The values for P, D, and WF are specified outside the subroutine. For convenience, P is specified as a single value between 0 and 65535. Lines 510-520 break P into high and low bytes and POKE them into the proper locations. Line 530 turns on the note and line 550 turns it off. Line 540 introduces a delay of duration D. Line 560 introduces a second short delay to allow time for the "sustain" prolongation of the note.

Line 20 calls a subroutine at line 300 that turns all voices off and turns the volume up. Line 25 allows the entering of attack, decay, sustain, and release parameters from the keyboard. Line 30 allows a single value 0-4096 to be entered for the pulsewidth in the case that the pulse waveform is chosen. Line 40 calls a subroutine at 400 that sets the sound parameters. If the pulse waveform is chosen, lines 420-430 break the pulsewidth into high and low bytes and POKE them into the proper locations. Line 440 combines attack with decay and sustain with release and POKEs them into the proper locations.

Finally, line 50 allows values for pitch and duration to be entered from the keyboard. Line 60 then calls the tone subroutine at line 500 and the note is played.

Figure 14.41 Program to play single notes on voice 0 of the Commodore 64.

```
5 REM MAKING SOUNDS
10 RA=54272:REM MUSIC REGISTERS REFERENCE ADDRESS
20 GOSUB 300:REM TURN VOICES OFF AND VOLUME UP
25 INPUT "ENTER WAVEFORM, ATTACK, DECAY, SUSTAIN, AND RELEASE";WF,AT,DE,SU,RE
30 IF WF=64 THEN INPUT "ENTER PULSEWIDTH";W
40 GOSUB 400:REM SET SOUND PARAMETERS
50 INPUT "ENTER PITCH AND DURATION";P,D
60 GOSUB 500:REM MAKE TONE
70 GOTO 50
300 REM TURN VOICES OFF AND VOLUME UP
310 FOR I=0 TO 23:POKE RA+I,0:NEXT I
320 POKE RA+24,15
330 RETURN
400 REM SET SOUND PARAMETERS AT=ATTACK DE=DECAY SU=SUSTAIN RE=RELEASE
401 REM WF=WAVEFORM W=PULSEWIDTH (FOR WF=64 ONLY)
410 IF WF<>64 THEN 440
420 WH=INT(W/256):WL=W-256*WH
430 POKE RA+2,WL:POKE RA+3,WH
440 POKE RA+5,16*AT+DE:POKE RA+6,16*SU+RE
450 RETURN
500 REM MAKE TONE P=PITCH D=DURATION
510 PH=INT(P/256):PL=P-256*PH
520 POKE RA,PL:POKE RA+1,PH
530 POKE RA+4,WF+1
540 FOR I=1 TO D:NEXT I
550 POKE RA+4,WF
560 FOR I=1 TO 50:NEXT I:REM SUSTAIN TIME
570 RETURN

READY.
```

After playing the specified note, the main program loops back to line 50 so that another note with a different pitch and duration (but same waveform and ADSR parameters) can be selected.

In order to try out some notes, turn up the volume on your TV set and type in the program of Figure 14.41. At the start, choose your pitches in the range 4000-8000 and your durations in the range 200-2000. Some waveform and ADSR values to try are shown in Figure 14.42. These values approximate the sounds of musical instruments as indicated.

Figure 14.42 Parameter values approximating the sound of musical instruments.

WF	AT	DE	SU	RE	W	Instrument
16	4	15	0	0	-	clarinet
16	0	9	0	0	-	ukelele
32	0	9	0	0	-	banjo
64	0	9	0	0	400	banjo
64	1	15	0	0	400	organ
32	1	15	15	0	-	organ
16	1	15	15	0	-	organ
64	0	9	0	0	256	guitar
64	0	9	0	0	2000	ukelele
32	5	8	12	3	-	violin-like
64	9	0	15	3	2000	synthesizer

Making Scales

The musical quality of the Commodore 64 can better be judged by playing some scales. The program of Figure 14.43 is an adaptation of the above tone-producing program for the purpose of playing musical scales. Notice the subroutines for (1) turning the voices off and the volume up (2) setting the sound parameters, and (3) producing tones at lines 300, 400, and 500, respectively, are unchanged.

Line 110 contains the values recommended by Commodore for an ascending C major scale. Line 120 contains the same values in reverse order to form a descending version of the same scale, following by the value −1. The latter serves to mark the end of the song. Lines 25-30 allow the user to specify the waveform, pulsewidth (if appropriate), and ADSR parameters. Line 45 chooses the duration D for each note. The loop in lines 50–60 plays the scale using the data in lines 110 and 120. The program then loops back to line 25 so new parameters can be chosen.

Make the necessary modifications to the previous program to obtain the program of Figure 14.43. Then, execute the program with various parameter values such as those suggested in Figure 14.42. Just for fun, choose WF=128 at least once.

Figure 14.43 Program for playing musical scales on the Commodore 64.

```
5 REM MAKING SCALES
10 RA=54272:REM MUSIC REGISTERS REFERENCE ADDRESS
20 GOSUB 300:REM TURN VOICES OFF AND VOLUME UP
25 INPUT "ENTER WAVEFORM, ATTACK, DECAY, SUSTAIN, AND RELEASE";WF,AT,DE,SU,RE
30 IF WF=64 THEN INPUT "ENTER PULSEWIDTH";W
40 GOSUB 400:REM SET SOUND PARAMETERS
45 D=200:REM SET DURATION THEN PLAY SCALE:
50 READ P:IF P<0 THEN RESTORE:GOTO 25
60 GOSUB 500:GOTO 50
100 REM C MAJOR SCALE
110 DATA 4291,4817,5407,5728,6430,7217,8101,8583
120 DATA 8583,8101,7217,6430,5728,5407,4817,4291,-1
300 REM TURN VOICES OFF AND VOLUME UP
310 FOR I=0 TO 23:POKE RA+I,0:NEXT I
320 POKE RA+24,15
330 RETURN
400 REM SET SOUND PARAMETERS AT=ATTACK DE=DECAY SU=SUSTAIN RE=RELEASE
401 REM WF=WAVEFORM W=PULSEWIDTH (FOR WF=64 ONLY)
410 IF WF<>64 THEN 440
420 WH=INT(W/256):WL=W-256*WH
430 POKE RA+2,WL:POKE RA+3,WH
440 POKE RA+5,16*AT+DE:POKE RA+6,16*SU+RE
450 RETURN
500 REM MAKE NOTE P=PITCH D=DURATION
510 PH=INT(P/256):PL=P-256*PH
520 POKE RA,PL:POKE RA+1,PH
530 POKE RA+4,WF+1
540 FOR I=1 TO D:NEXT I
550 POKE RA+4,WF
560 FOR I=1 TO 50:NEXT I
570 RETURN

READY.
```

Producing Multiple Clicks

Let us now consider producing some simple sound effects using essentially the same subroutines for turning up the volume, setting sound parameters, and producing tones. For simplicity, we have simplified the latter two subroutines. We have omitted the lines for setting the pulsewidth in the "setting sound parameters" subroutine and we have dropped the sustain delay in the "producing tones" subroutine.

Figure 14.44 contains a program to produce N clicks with a time separation S at pitch P. Line 25 chooses the sawtooth waveform and a particular set of ADSR parameters and sets them. Lines 30-40 allow the pitch, time separation, and number of clicks to be entered from the keyboard. Make the modifications necessary to create this program and try it with various pitches P, time separations S, and number of clicks N. In particular, try P=2000 and P=32000 with S=1 and N=60. This gives a click and a blip, respectively, every second for one minute. Check the clicks and blips against your watch.

Producing a Phaser Noise

If you repeatedly call the tone subroutine of Figure 14.44 with different pitch values P, you can produce a variety of effects. For example, the program shown in Figure 14.45 produces "phaser" noises consisting of NC cycles of a sound in which the pitch varies from P1 to P2 in steps of DP.

To hear what this noise sounds like, execute the program and enter from the keyboard NC = 10, P1 = 8000, P2 = 5000, and DP = −400. Note that the pitch varies from high to low each cycle. Try a variety of different values for NC, P1, P2, and DP.

Producing a Siren Sound

A siren noise can be produced by repeatedly calling the tone subroutine, first with increasing values of P, and then with decreasing values of P. The program is shown in Figure 14.46.

In lines 30-50, the user is asked to choose values for the number of cycles NC, the starting pitch P1, the ending pitch P2, the pitch increment DP, and the holding time T, which controls the amount of time each pitch is held. The volume is turned up in line 20 and the waveform and ADSR parameters are set in line 25. The inner loop in lines 60–70 creates a wail in ascending pitch and that in lines 75-85 produces one in descending pitch. The outer loop in lines 55 and 90 counts the cycles. Line 95 turns the voice off and then loops to allow a new choice of parameters.

Figure 14.44 Program to produce N clicks with separation S and pitch P.

```
5 REM MAKE N CLICKS WITH SPACING S AND PITCH P
10 RA=54272:REM MUSIC REGISTERS REFERENCE ADDRESS
20 GOSUB 300:REM TURN VOICES OFF AND VOLUME UP
25 WF=32:AT=1:DE=15:SU=0:RE=0:GOSUB 400
30 INPUT "ENTER PITCH";P
35 INPUT "ENTER SEPARATION (SEC)";S
40 INPUT "ENTER NUMBER OF CLICKS";N
45 D=1
50 FOR J=1 TO N
55 GOSUB 500
60 FOR I=1 TO 730*S:NEXT I
65 NEXT J
70 GOTO 30
300 REM TURN VOICES OFF AND VOLUME UP
310 FOR I=0 TO 23:POKE RA+I,0:NEXT I
320 POKE RA+24,15
330 RETURN
400 REM SET SOUND PARAMETERS AT=ATTACK DE=DECAY SU=SUSTAIN RE=RELEASE
440 POKE RA+5,16*AT+DE:POKE RA+6,16*SU+RE
450 RETURN
500 REM MAKE TONE P=PITCH D=DURATION
510 PH=INT(P/256):PL=P-256*PH
520 POKE RA,PL:POKE RA+1,PH
530 POKE RA+4,WF+1
540 FOR I=1 TO D:NEXT I
550 POKE RA+4,WF
570 RETURN

READY.
```

Figure 14.45 Program to make a phaser noise.

```
5 REM PHASER NOISE
10 RA=54272:REM MUSIC REGISTERS REFERENCE ADDRESS
20 GOSUB 300:REM TURN VOICES OFF AND VOLUME UP
25 WF=32:AT=1:DE=15:SU=0:RE=0:GOSUB 400
30 INPUT "ENTER NO. CYCLES";NC
35 INPUT "ENTER STARTING PITCH";P1
40 INPUT "ENTER ENDING PITCH";P2
45 INPUT "ENTER PITCH INCREMENT";DP
50 FOR J=1 TO N
55 D=10:FOR J=1 TO NC
60 FOR P=P1 TO P2 STEP DP
65 GOSUB 500
70 NEXT P
75 NEXT J
80 GOTO 30
300 REM TURN VOICES OFF AND VOLUME UP
310 FOR I=0 TO 23:POKE RA+I,0:NEXT I
320 POKE RA+24,15
330 RETURN
400 REM SET SOUND PARAMETERS AT=ATTACK DE=DECAY SU=SUSTAIN RE=RELEASE
440 POKE RA+5,16*AT+DE:POKE RA+6,16*SU+RE
450 RETURN
500 REM MAKE TONE P=PITCH D=DURATION
510 PH=INT(P/256):PL=P-256*PH
520 POKE RA,PL:POKE RA+1,PH
530 POKE RA+4,WF+1
540 FOR I=1 TO D:NEXT I
550 POKE RA+4,WF
570 RETURN

READY.
```

Figure 14.46 Program to produce a siren sound.

```
5 REM SIREN NOISE
10 RA=54272:REM MUSIC REGISTERS REFERENCE ADDRESS
20 GOSUB 300:REM TURN VOICES OFF AND VOLUME UP
25 WF=16:AT=1:DE=9:SU=3:RE=15:GOSUB 400
30 INPUT "ENTER NO. CYCLES";NC
35 INPUT "ENTER STARTING PITCH";P1
40 INPUT "ENTER ENDING PITCH";P2
45 INPUT "ENTER PITCH INCREMENT";DP
50 INPUT "ENTER HOLDING TIME";D
55 FOR J=1 TO NC
60 FOR P=P1 TO P2 STEP DP
65 GOSUB 500
70 NEXT P
75 FOR P=P2 TO P1 STEP -DP
80 GOSUB 500
85 NEXT P
90 NEXT J
95 GOSUB 300:GOTO 30
300 REM TURN VOICES OFF AND VOLUME UP
310 FOR I=0 TO 23:POKE RA+I,0:NEXT I
320 POKE RA+24,15
330 RETURN
400 REM SET SOUND PARAMETERS AT=ATTACK DE=DECAY SU=SUSTAIN RE=RELEASE
440 POKE RA+5,16*AT+DE:POKE RA+6,16*SU+RE
450 RETURN
500 REM MAKE TONE P=PITCH D=DURATION
510 PH=INT(P/256):PL=P-256*PH
520 POKE RA,PL:POKE RA+1,PH
530 POKE RA+4,WF+1
540 FOR I=1 TO D:NEXT I
550 POKE RA+4,WF
570 RETURN

READY.
```

Run this program for the following values: NC = 4, P1 = 10000, P2 = 17000, DP = 200 and T = 10. Experiment with other values.

Adding Sound Effects to the "Plane with Phasers" Program

Recall in Chapter 11, we developed a program that fires phasers from a fighter plane. We will add some sound effects to that program for the Commodore 64. The original program of Chapter 11 (Figure 11.19) will be unchanged except to add a subroutine to initialize the sound parameters and to modify the subroutines for firing the left and right phasers.

The phaser-firing subroutines will be modified to fire a single "bullet" that makes a noise each time it moves. These modified subroutines are shown in Figure 14.47. Lines 235 and 335 produce a phaser noise by calling subroutine 1000. Lines 240 and 340 erase the previously plotted "bullet". The old subroutine at 400 that was used to erase the phaser is no longer needed.

Figure 14.47 Phaser-firing subroutines modified to produce noise.

```
200 REM FIRE LEFT PHASER
210 U=X-2
220 FOR V=Y TO 0 STEP -1
230 GOSUB 500:PRINT"●"
235 GOSUB 1000
240 GOSUB 500:PRINT" "
245 NEXT
250 RETURN
300 REM  FIRE RIGHT PHASER
310 U=X+2
320 FOR V=Y TO 0 STEP -1
330 GOSUB 500:PRINT"O"
335 GOSUB 1000
340 GOSUB 500:PRINT" "
345 NEXT
350 RETURN

READY.
```

The subroutines to make the phaser noise are shown in Figure 14.48. The subroutines in Figure 14.45 have been modified to fit the plane program. Lines 900-930 contain a combined subroutine to turn off all voices, turn up the volume, and set the waveform and ADRS parameters. Line 930 chooses D=1, P1=6000, P2=5000, and DP=−500. To call this subroutine, you should modify line 10 of the old main program to read: **10 PRINT "CLR": X=10: Y=10: GOSUB 900.**

Lines 1000-1070 contain a combined subroutine to control the phaser sound and to produce tones. When this modified plane program is run and a phaser is fired, a single bullet is released, as shown in Figure 14.38, and it makes a noise as it moves along.

The various sound subroutines for the Commodore 64 presented in this section should enable you to add interesting sound effects to your programs. How to incorporate music into a Commodore 64 program is described in more detail in Chapter 15.

DEFINING YOUR OWN GRAPHIC CHARACTERS

Appendix C contains the internal character set of the Commodore 64/VIC 20. This character set is stored in ROM starting at memory location 32768 on the VIC 20 and 53248 on the Commodore 64. Each character uses eight consecutive memory locations for its definition. The character is defined on an 8 × 8 grid where each row of the grid corresponds to a separate memory location and each column in the grid corresponds to one of the 8 bits in the data byte (0-255) stored at a particular location.

The characters given in Appendix C are listed in the order in which they are stored in ROM. Character @ is 0. Character A is 1; therefore its definition starts at location 32768 + 8*1 on the VIC 20 and 53248 + 8*1 on the Commodore 64 (remember, each definition takes

Figure 14.48 Sound producing subroutines for the plane with phasers.

```
900 RA=54272:FOR I=0 TO 23:POKE RA+I,0:NEXT I:POKE RA+24,15
910 WF=32:AT=1:DE=15:SU=0:RE=0
920 POKE RA+5,16*AT+DE:POKE RA+6,16*SU+RE
930 D=1:P1=6000:P2=5000:DP=-500:RETURN
1000 FOR P=P1 TO P2 STEP DP
1010 PH=INT(P/256):PL=P-256*PH
1020 POKE RA,PL:POKE RA+1,PH
1030 POKE RA+4,WF+1
1040 FOR I=1 TO D:NEXT I
1050 POKE RA+4,WF
1060 NEXT P
1070 RETURN

READY.
```

eight memory locations). In order to list the 8 bytes used to define the letter A, type in and run whichever of the following programs is appropriate. The result in the case of the Commodore 64 is shown in Figure 14.49.

```
10 REM VIC 20 PROGRAM
20 FOR J=0 TO 7
30 ? PEEK(32768+8*1+J)
40 NEXT J
```

```
10 REM COMMODORE 64 PROGRAM
20 POKE 56334,PEEK(56334) AND 254
30 POKE 1,PEEK(1) AND 251
40 FOR J=0 TO 7
50 ? PEEK(53248+8*1+J)
60 NEXT J
70 POKE 1,PEEK(1) OR 4
80 POKE 56334,PEEK(56334) OR 1
```

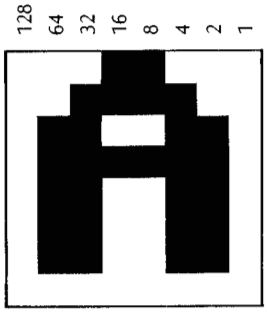
The above VIC 20 program is more straightforward than the Commodore 64 program. In the latter, lines 30 and 70 serve to switch the character set ROM into the memory locations starting at 53248 and then out again. Normally, these locations are used as input/output locations on the Commodore 64 and the ROM can't be seen. This illustrates what we discussed earlier; namely, some memory locations serve a dual purpose. Lines 20 and 80 serve to disable the keyboard while the character set ROM is switched in and then enable it once more. This is a safety feature. If a key is depressed when the I/O starting at 53248 is switched out, very strange things can happen.

Figure 14.49 PEEKing the eight values used to define the letter A.

```
LIST
20 POKE 56334,PEEK(56334) AND 254
30 POKE 1,PEEK(1) AND 251
40 FOR J=0 TO 7
50 PRINT PEEK(53248+8*1+J)
60 NEXT J
70 POKE 1,PEEK(1) OR 4
80 POKE 56334,PEEK(56334) OR 1
READY.

RUN
 24
 60
 102
 126
 102
 102
 102
 0

READY.
```

The relationship between the eight values in Figure 14.49 and the letter A is shown in Figure 14.50. Note that each row of the 8×8 grid has a number associated with it. The number is equal to the sum of the numbers at the top of each column in which a bit is "on". An "on" bit corresponds to a displayed point.

Now that you understand how the Commodore 64/VIC 20 defines the letter A (and all other characters), you may wonder if you can define your own characters. You can. First you must define the new characters you want on an 8×8 grid. Then you must store these eight values in the character set table. How can you do this when this table is in ROM (read only memory)? You must first move the character set table into RAM (read/write memory) and then tell the Commodore 64/VIC 20 where you moved it.



16 + 8 = 24
32 + 16 + 8 + 4 = 60
64 + 32 + 4 + 2 = 102
64 + 32 + 16 + 8 + 4 + 2 = 126
64 + 32 + 4 + 2 = 102
64 + 32 + 4 + 2 = 102
64 + 32 + 4 + 2 = 102
0

Figure 14.50 Defining the letter A.

To demonstrate how to generate our own characters, it will suffice to use a limited character set with only 64 characters. Because an unmodified VIC 20 has only 5 K bytes of RAM, this is about all it can handle because 64 characters takes 64*8 = 512 bytes of memory to store. Examination of Appendix C reveals that all of the uppercase letters, digits, and punctuation symbols are contained in the first 64 characters in the ROM character set.

On the VIC 20, memory location 7168 is a convenient place to store 512 bytes of character information, i.e. 64 characters. On the Commodore 64, memory location 12288 is a similar place. The VIC 20 uses the contents of memory location 36869 to tell where the character set is located. By POKEing the value 255 into address 36869, an unmodified VIC 20 will look for character information starting at location 7168. If the VIC 20 has a memory expansion, the value to be POKEd changes to 207. Similarly, by POKEing the value 28 into location 53272, it turns out the Commodore 64 will look for character information starting at location 12288.

The following loop will copy the first 512 bytes of the VIC 20 character set in ROM into the new location in RAM:

```
130 FOR I=0 TO 64*8-1
140 POKE 7168+I,PEEK(32768+I)
150 NEXT I
```

The same loop will work on the Commodore 64 with line 140 changed to read: **140 POKE 12288+I,PEEK(53248+I).**

To demonstrate making our own characters, let us replace the digits 0-9 in the normal character set, respectively, by the ten new symbols shown in Figure 14.51. These new symbols consist of the blankspace, eight directional arrows, and the reverse video blankspace.

If you have a VIC 20, consider the program listing of Figure 14.52 and, if you have a Commodore 64, consider that of Figure 14.53. In each case, lines 205-250 contain the definitions of the above new symbols. Line 205 defines the blankspace and line 250 defines the reverse video blankspace. The directional arrows are defined in lines 210-245 starting with the up-right arrow and then defining in order the right, down-right, down, down-left, left, up-left, and up arrows. The definitions of these characters can easily be derived from Figure 14.51 by the method described earlier for the letter A. Note that in all cases, the bottom row and the rightmost column are left blank to provide a separation between adjacent characters.

In both programs, lines 100-250 contain a subroutine that substitutes the new symbols for the digits 0-9. Line 110 directs the computer to look for its character information in RAM and lines 130-150 copy 64 characters from ROM into RAM. (If you have a VIC 20 with a memory expansion, change the value 255 to 207 in line 110.) Lines 170-190 POKE the new character definitions into the RAM character set in place of the digits 0-9. Note that the screen POKE code for 0 is 48 and that for 9 is 58. This explains the values in the FOR statement of line 170.

Lines 10-40 form the main program. In line 20, the program prints "WAIT" since changing the character set takes time. In line 30, the new characters are PRINTed out once in a row. Following execution of the program, of course, the new characters are still in force. This makes it possible to draw pictures with the new symbols as shown in Figure 14.54.

Figure 14.51 New graphic symbols to replace the digits.

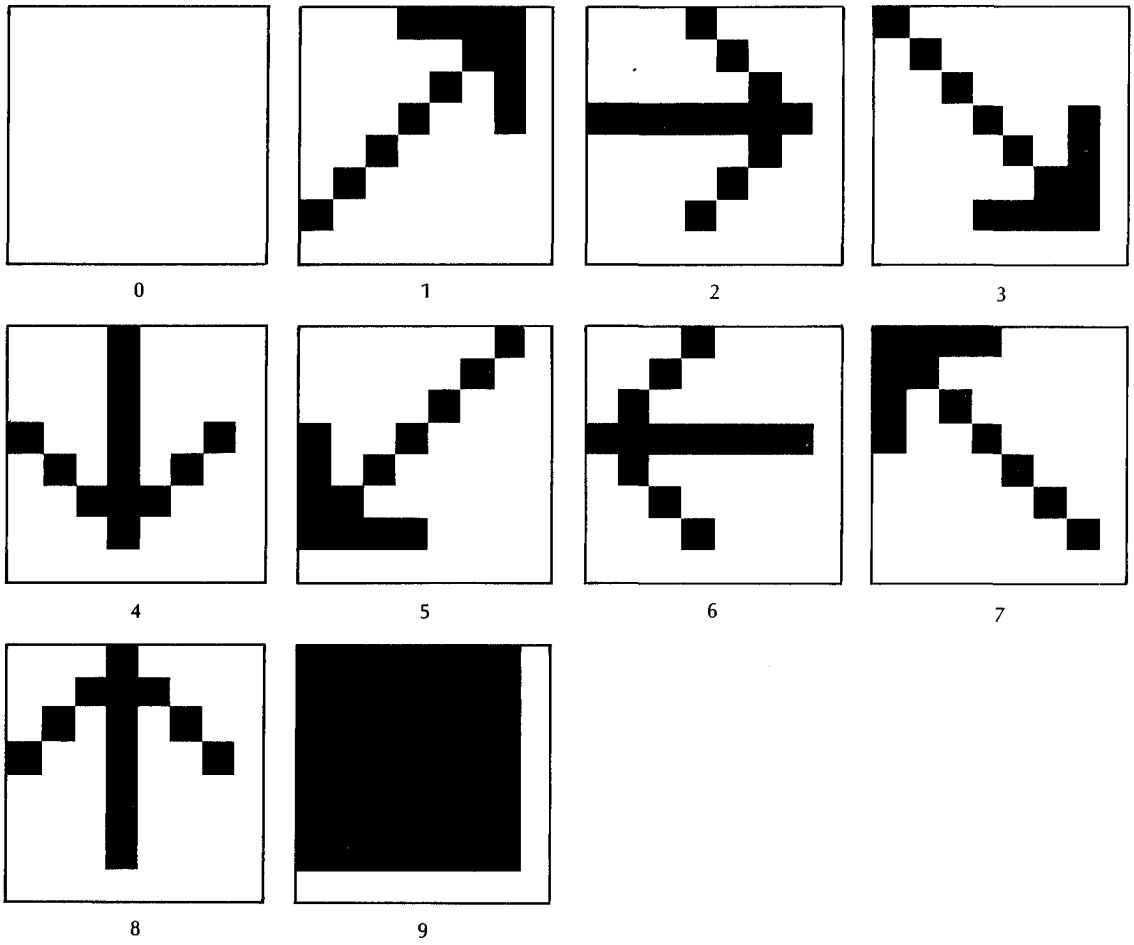


Figure 14.52 VIC 20 program to display new graphic characters in place of digits.

```
10 REM DISPLAY CUSTOM CHARACTERS
20 PRINT "■■WAIT":GOSUB 100:REM SETUP CUSTOM CHARACTERS
30 FOR I=48 TO 57:PRINT CHR$(I);:NEXT I
40 END
100 REM SET-UP CUSTOM GRAPHICS CHARACTERS
110 POKE 36869,255:REM TAKE CHARACTER INFO FROM RAM
120 REM COPY CHARACTERS FROM ROM INTO RAM
130 FOR I=0 TO 64*8-1
140 POKE 7168+I,PEEK(32768+I)
150 NEXT I
160 REM REPLACE DIGITS BY CUSTOM CHARACTERS
170 FOR I=48*8 TO 58*8-1
180 READ D:POKE 7168+I,D
190 NEXT I
200 RETURN
205 DATA 0,0,0,0,0,0,0,0
210 DATA 30,6,10,18,32,64,128,0
215 DATA 16,8,4,254,4,8,16,0
220 DATA 128,64,32,18,10,6,30,0
225 DATA 16,16,16,146,84,56,16,0
230 DATA 2,4,8,144,160,192,240,0
235 DATA 16,32,64,254,64,32,16,0
240 DATA 240,192,160,144,8,4,2,0
245 DATA 16,56,84,144,16,16,16,0
250 DATA 254,254,254,254,254,254,254,0

READY.
```

Figure 14.53 Commodore 64 program to display new graphic characters in place of digits.

```
10 REM DISPLAY CUSTOM CHARACTERS
20 PRINT "■■WAIT":GOSUB 100:REM SETUP CUSTOM CHARACTERS
30 FOR I=48 TO 57:PRINT CHR$(I);:NEXT I
40 END
100 REM SET-UP CUSTOM GRAPHICS CHARACTERS
110 POKE 53272,28 :REM TAKE CHARACTER INFO FROM RAM
114 POKE 56334,PEEK(56334) AND 254:REM IGNOR KEYBOARD INTERRUPTS
117 POKE 1,PEEK(1) AND 251:REM SWITCH IN CHARACTER ROM
120 REM COPY CHARACTERS FROM ROM INTO RAM
130 FOR I=0 TO 64*8-1
140 POKE 12288+I,PEEK(53248+I)
150 NEXT I
154 POKE 1,PEEK(1) OR 4:REM SWITCH OUT CHARACTER ROM
157 POKE 56334,PEEK(56334) OR 1:REM ENABEL KEYBOARD INTERRUPTS
160 REM REPLACE DIGITS BY CUSTOM CHARACTERS
170 FOR I=48*8 TO 58*8-1
180 READ D:POKE 12288+I,D
190 NEXT I
200 RETURN
205 DATA 0,0,0,0,0,0,0,0
210 DATA 30,6,10,18,32,64,128,0
215 DATA 16,8,4,254,4,8,16,0
220 DATA 128,64,32,18,10,6,30,0
225 DATA 16,16,16,146,84,56,16,0
230 DATA 2,4,8,144,160,192,240,0
235 DATA 16,32,64,254,64,32,16,0
240 DATA 240,192,160,144,8,4,2,0
245 DATA 16,56,84,144,16,16,16,0
250 DATA 254,254,254,254,254,254,254,0

READY.
```

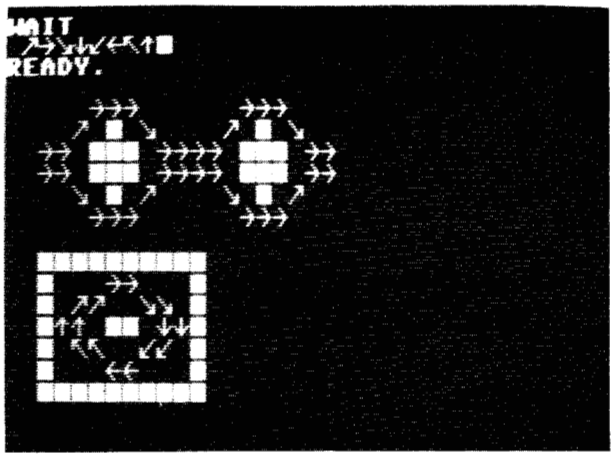


Figure 14.54 Drawing pictures with the new graphic symbols.

SPRITE GRAPHICS ON THE COMMODORE 64

The Commodore 64 has an advanced mode of graphics beyond anything available on the VIC 20. Using this mode, it is possible to create and move about on the screen as many as 8 small figures called *sprites*. Such figures are particularly appropriate for video games. Creation and control of sprites is accomplished totally by POKEing memory locations. In this section, we shall illustrate the use of Sprite graphics by drawing a multicolored sailboat that goes sailing on the screen.

A sprite is a figure contained in a 24×21 grid. Our sailboat is shown drawn in such a grid in Figure 14.55. To make our sailboat multicolored, we shall actually create three distinct sprites. Sprite 0 will be a white hull; Sprite 1 will be a yellow foresail; and Sprite 2 will be a light red mainsail.

The definition of a sprite is contained in a block of 64 consecutive memory locations. The last location, however, is not used. The sprite data in the first 63 locations consists of a scan of the sprite picture rowwise from top to bottom. The first three locations define the left, middle, and right thirds of the top row of the sprite. The next three locations define the left, middle, and right thirds of the second row of the sprite, etc.

Each memory location in the sprite definition controls 8 adjacent picture elements. These picture

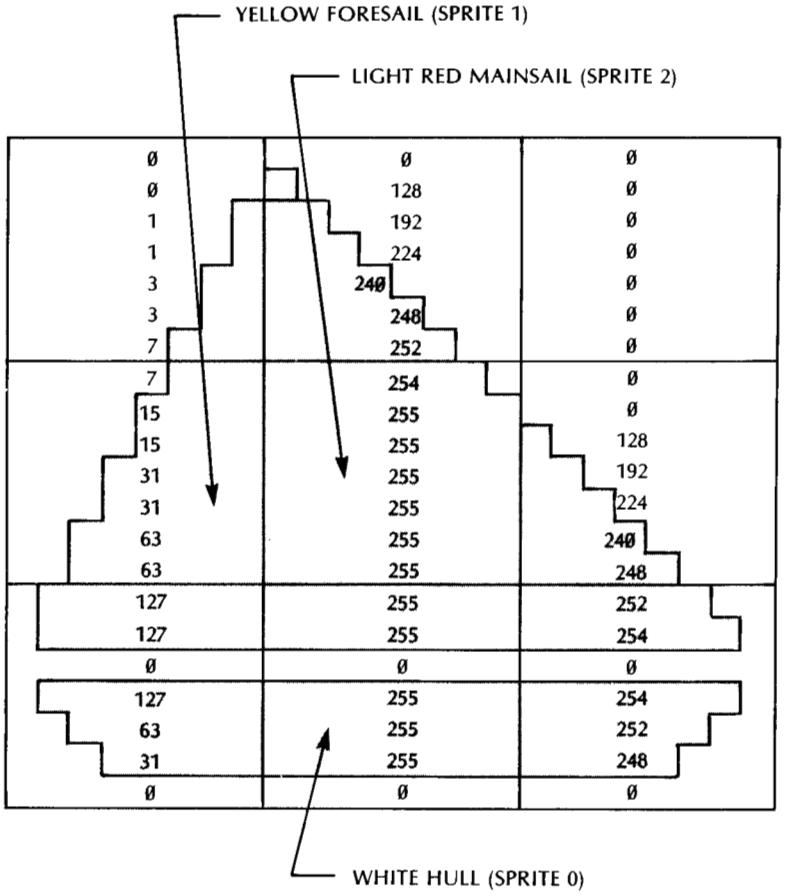


Figure 14.55 Design of a sprite sailboat for the Commodore 64.

elements have weights 128, 64, 32, 16, 8, 4, 2, and 1 from left-to-right. To compute the value to be POKEd into a memory location, we add up the weights of the picture elements that are "on" (colored). This is exactly the same method that is used to define the rows of screen characters, as was discussed in the previous section.

The sprite data for our sailboat is shown in Figure 14.56. In Figure 14.55, we show the sprite data we would use to create the entire sailboat with one sprite. To create a multicolored sailboat, we divide the sprite data into three parts and replace by 0 all values shown in Figure 14.55 that are outside of the particular part of the sailboat we are describing. For example, the data for the hull begins with 51 consecutive 0's to blank the first 17 rows of Sprite 0 (where the sails will be).

The memory locations that create and control sprites start with V = 53248. Figure 14.57 shows these memory locations and the data that is POKEd into each location. Location V+21 turns the sprites on and off. This is done in the usual fashion using the weights 1, 2, 4, 8, 16, 32, 64, and 128 for sprites 0–7 respectively. For example, to turn on sprites 0, 3, and 7, we POKE 1+8+128 = 137 into location V+21.

Locations V to V+16 determine the positions of the sprites on the screen as indicated in Figure 14.57. The entire screen is a grid that is 40*8 × 25*8 = 320 × 200 picture elements in size. To position a sprite on the screen, the distance from the top edge of the screen to the upper left corner of the sprite is POKEd into the Y position for that sprite. To be viewed on the screen, this value of Y must lie between 50 and 229. Similarly, the distance from the left edge of the screen to the upper left corner of the sprite is POKEd into the X position for that sprite. Any value for X between 24 and 255 is viewable.

However, there are viewable sprite positions still further to the right of the position where X = 255. To position the sprite further to the right, one POKEs the proper bit of location V+16 with a 1 (which is like adding 256 to the value of X) and then POKEing the difference between the desired horizontal distance and 256 into the X position for that sprite. To be viewable, this difference must lie between 0 and 65. For example, to position sprite 4 exactly 280 units from the left edge, use

POKE V+16,PEEK(V+16) OR 16:POKE V+8,24

The first POKE sets the most significant bit (MSB) of Sprite 4 to 1 and the second POKEs the remaining distance 280 − 256 = 24 into the X position of Sprite 4.

Locations V+23 and V+29, respectively, allow the size of a sprite to be doubled in the vertical and

Figure 14.57 Memory locations for creating and controlling sprites.

memory location	data
V	X position - Sprite 0
V+1	Y position - " "
V+2	X position - Sprite 1
V+3	Y position - " "
V+4	X position - Sprite 2
V+5	Y position - " "
V+6	X position - Sprite 3
V+7	Y position - " "
V+8	X position - Sprite 4
V+9	Y position - " "
V+10	X position - Sprite 5
V+11	Y position - " "
V+12	X position - Sprite 6
V+13	Y position - " "
V+14	X position - Sprite 7
V+15	Y position - " "
V+16	MSB X position - Sprites 0-7
V+21	Enable/disable - Sprites 0-7
V+23	Vertical expansion - Sprites 0-7
V+29	Horizontal expansion - Sprites 0-7
V+39	Color - Sprite 0
V+40	Color - Sprite 1
V+41	Color - Sprite 2
V+42	Color - Sprite 3
V+43	Color - Sprite 4
V+44	Color - Sprite 5
V+45	Color - Sprite 6
V+46	Color - Sprite 7

(V = 53248)

Figure 14.56 Data for a sprite sailboat on the Commodore 64.

```
300 REM  BOAT HULL
305 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
310 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
315 DATA 0,0,0,0,0,0,0,0,0,127,255,254,63,255,252,31,255,248,0,0,0
320 REM FORESAIL
325 DATA 0,0,0,0,0,0,1,0,0,1,0,0,3,0,0,3,0,0,7,0,0
330 DATA 7,0,0,15,0,0,15,0,0,31,0,0,31,0,0,63,0,0,63,0,0
335 DATA 127,0,0,127,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
340 REM MAINSAIL
345 DATA 0,0,0,0,128,0,0,192,0,0,224,0,0,240,0,0,248,0,0,252,0
350 DATA 0,254,0,0,255,0,0,255,128,0,255,192,0,255,224,0,255,240,0,255,248
355 DATA 0,255,252,0,255,254,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

READY.
```

horizontal directions. The particular sprite is chosen as usual by using the above weights. For example, to double the size of Sprite 2 in the horizontal direction, use

POKE V+29,PEEK(V+29) OR 4

Finally, the locations V+39 through V+46 determine the color of Sprites 0-7, respectively. The color numbers to use are the standard values 0-15 as given in Figure 14.5b.

The main program for producing a multicolored sailboat using sprites is given in Figure 14.58. Line 20 clears the screen and POKEs the color light blue into the background color, location 53281. The result is to make the entire screen light blue in color like an ocean or a lake. Line 25 assigns memory space for Sprites 0-2 and lines 30-40 read and POKE the data for the hull, foresail, and mainsail into Sprites 0-2, respectively. Line 45 sets the hull, foresail, and mainsail colors to white, yellow, and light red, respectively. Line 50 enables Sprites 0-2 and line 55 turns off all MSBs.

The loop in lines 65–80 causes the three Sprites to move in synchronization from right to left across the screen. Line 75 is a delay loop that makes the motion relatively slow. On exit from the outer loop, line 85 turns off the sprites and line 90 restores the background color to dark blue. Type in this program and the data from Figure 14.56 and go sailing for awhile!

You can double the width of the sailboat by adding the statement **57 POKE V+29,7.** You can double the height of the sailboat by adding the statement **58 POKE V+23,7.** You can do both by adding both statements. Try all combinations.

You can see that sprite graphics is a powerful mode of high resolution graphics. Of course, the function of some of the POKE locations is rather complicated. However, the most important thing is the shortness of programs required to create and control sophisticated graphical figures. In particular, note how short the sailboat program is.

Figure 14.58 Main program for a multicolored sailboat using sprites.

```
10 REM  SPRITE SAILBOAT
15 V=53248
20 PRINT "J":POKE 53281,14:REM BACKGROUND COLOR LIGHT BLUE
25 FOR I=0 TO 2:POKE 2040+I,13+I:NEXTI:REM RESERVE SPACE FOR SPRITES 0-2
30 FOR I=0 TO 62:READ D:POKE 832+I,D:NEXT I:REM SETUP SPRITE 0
35 FOR I=0 TO 62:READ D:POKE 896+I,D:NEXT I:REM SETUP SPRITE 1
40 FOR I=0 TO 62:READ D:POKE 960+I,D:NEXT I:REM SETUP SPRITE 2
45 POKE V+39,1:POKE V+40,7:POKE V+41,10:REM SET SPRITE COLORS
50 POKE V+21,7:REM TURN ON SPRITES 0-2
55 POKE V+16,0:REM TURN OFF MSBS
60 REM SHIP AHOY!
65 FOR J=250 TO 25 STEP -1
70 FOR I=0 TO 2:POKE V+2*I,J:POKE V+1+2*I,180:NEXT I
75 FOR I=1 TO 100:NEXT I
80 NEXT J
85 POKE V+21,0:REM TURN OFF SPRITES
90 POKE 53281,6:REM RESTORE NORMAL BACKGROUND COLOR
100 END

READY.
```

EXERCISE 14-4

Using the time function TI described in Chapter 4, count the number of jiffies that occur while a key is held down. Write a program that displays this time to the nearest tenth of a second.

EXERCISE 14-5

Plot the bar graph of New England states shown in Figure 10.10, and print the title and scale label in both upper and lower case letters. Use the alternate graphic symbol SHIFT £ (CHR$(169)) to plot the bar graph.

EXERCISE 14-6

Write a program that will POKE the pictures of the two skiers shown in Exercises 2-3(e) and 2-3(f) at the screen location X1, Y1 and X2, Y2 respectively. Store the POKE codes for these two pictures in a three-dimensional array.

EXERCISE 14-7

Write a program that stores a sequence of pitch (P) and length (L) values in a DATA statement and then plays this sequence of tones. The DATA statement (or statements) should be of the form

20 DATA P1, L1, P2, L2, P3, L3, . . .

where P1 and L1 are the pitch and length of the first tone, P2 and L2 are the pitch and length of the second tone, and so on. To play a complete song, see Exercise 15-3 in Chapter 15.

EXERCISE 14-8

To the sprite sailboat program, add three more sprites to draw the sailboat facing to the right. Then, modify the main loop so as to repeat, first, sailing the original sailboat across the screen from right to left and, then, sailing the new one from left to right.

15 LEARNING TO PUT IT ALL TOGETHER

Throughout this book you have learned how to use the various features of CBM BASIC. You have learned how to write short programs that draw pictures and make sounds. Now that you know the BASIC language, you will want to write your own programs. How do you go from an idea of something you would like the Commodore 64/VIC 20 to do to a working BASIC program that does it? That is what this chapter is all about.

In order to illustrate the various steps involved in developing a program, we will write two complete programs in this chapter. Although both programs are useful and fun to run, it is the process of developing the programs that we are trying to illustrate.

The first program plays a popular word-guessing game called "hangman." The second program converts your Commodore 64/VIC 20 into an organ on which you can play music from the keyboard.

In this chapter you will learn:

1. how to define what you want to do and describe the program in words
2. to define the variables you will need in the program
3. the technique of top-down programming
4. how to write a program to play hangman
5. how to store data on a cassette tape and floppy diskette
6. how to play music on your Commodore 64/VIC 20

HANGMAN

We will now develop a program to play the word-guessing game hangman. The following six steps will help you develop a program systematically. We will follow these six steps in developing hangman.

STEP 1: Define what you want the program to do

STEP 2: Describe the program in words

STEP 3: Define program variable names

STEP 4: Write and test the main program and essential subroutines

STEP 5: Write and test the remaining subroutines

STEP 6: Test the entire program and make improvements.

Defining What Hangman Will Do

This is the most important step in developing a program, but it is a step that is often omitted or not carried out adequately. There is an almost irresistible temptation to start writing BASIC code immediately. You *must* resist this temptation at all costs. You should not write any BASIC code until STEP 4.

Poor programming, like poor writing is usually a sign of poor and confused thinking. If you do not have a clear idea of what you want the program to do, you will have little chance of writing a program to do it. When you begin, you may not know all of the features that your program will eventually have. Programming is an interactive process, and often you will improve a program by rewriting it several times. However, you must understand enough about what you want the program to do to get started.

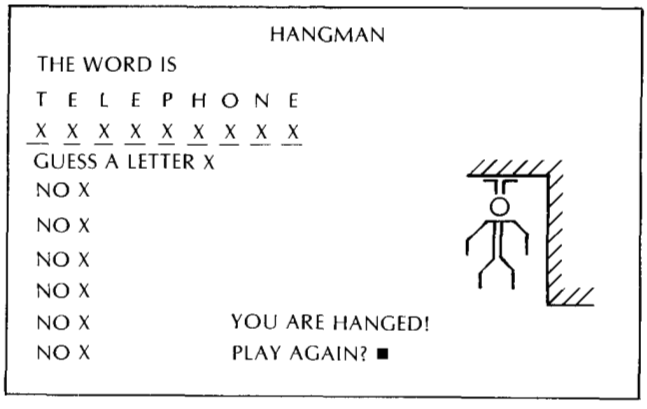
Hangman is a word-guessing game in which the Commodore 64/VIC 20 thinks of a word and displays a space for each letter of the word. You guess a letter. If the letter occurs in the word, it is inserted at all the locations at which it occurs. If the letter does not occur in the word, then another part of your body is added to the hanging gallows. You keep guessing letters until you either guess the word, or until you guess six wrong letters, at which time your body is complete and you are hanged.

The first things to decide are what you want the screen to look like and how you want the screen to respond to various inputs and conditions. The best idea is to sketch the screen to scale on a grid that is 23 × 22 in the case of the VIC 20 or 25 × 40 in the case of the Commodore 64. Figure 15.1 shows the screen layout for the hangman game in each case.

Figure 15.1 Screen layouts for Hangman program: (a) VIC 20 and (b) Commodore 64.

(a)

(b)



When the program is first executed, the name **HANGMAN** is written in reverse video at the top of the screen. The program then displays the spaces for the word, the gallows, and the words **GUESS A LETTER** followed by a blinking cursor. As each correct letter is guessed, it is filled in at the appropriate blank position(s). When an incorrect letter is guessed, then the phrase **NO X** (where **X** is the letter guessed) will be printed on the left lower part of the screen, and another part of the body will be added.

If you guess the word correctly, the word will be flashed and the words **YOU ARE SAVED!** will be printed next to the gallows. If you fail to guess the word, the correct word will be displayed above the spaces, and the words **YOU ARE HANGED!** will be printed next to the gallows. The words **PLAY AGAIN?** followed by a blinking cursor will then be displayed at the lower center of the screen.

Having laid out what you want the screen to look like and having thought about how the game is to be played, you are ready to write a *word-description* of the program.

A Word-Description of Hangman

At this point you should write a word-description of the program that will completely describe its logic. Use pseudocode, flowcharts, or whatever you find useful. A word-description for the hangman program, written in pseudocode is shown in Figure 15.2. Study this word-description carefully. Note that it follows closely

Figure 15.2 Pseudocode word-description of Hangman program.

```
loop: Clear screen
      Display HANGMAN
      Display gallows and the words GUESS A LETTER
      Find a random word
      Display spaces for word
      do until Word is guessed or you are hanged
            Guess a letter
            Search for letter in word
            if letter is in word
            then display letter at proper position
            else display NO "letter"
                  add part to body
      enddo
      if word is guessed
      then blink word
           display YOU ARE SAVED!
      else display correct word
           display YOU ARE HANGED!
      Ask to PLAY AGAIN?
repeat while answer is "Y"
```

the screen layout we made in Figure 15.1 and our ideas on how the program should work. Developing the word-description shown in Figure 15.2 is the most creative part of writing the program. At this point most of the hard work is done. This is why it is so important that you understand how to write word-descriptions like the one in Figure 15.2. Study it again. Note particularly how the *do until* loop is used to include the entire process of guessing a word. Being able to identify the appropriate looping structure for a particular problem (think *do until, repeat while,* or *for...next*) is one of the important skills you will need to develop in order to become a good programmer.

Figure 15.3 Definition of initial variables to be used in Hangman program.

```
W$ = word to be guessed
 L = length of word to be guessed
G$ = letter guessed
NL = number of correct letter positions guessed
NH = number of incorrect guesses
 R = [1 if G$ is in W$
     [0 if G$ is not in W$
```

Figure 15.4 More detailed version of *do until* loop used in Hangman program.

```
NL=0
NH=0
do until NL=L or NH=6
  Guess a letter G$
  for I=1 to L
      if G$=MID$(W$,I,1)
      then print G$ plus space
           NL=NL+1
           R=1
      else move cursor 2 spaces
  next I
  if R=1
  then do nothing
  else NH=NH+1
       display NO "letter"
       add part to body
enddo
```

Defining Program Variables

At this stage in the development of the program you should define names for those variables that you know you will need. You will not know all of the variables that you eventually use, but begin by defining the important ones that you do know. This will help you to focus on how you will implement various little algorithms. Be particularly conscious of defining appropriate string variables and arrays.

In the hangman program we will store the word to be guessed in the string W$. The length of this string (the number of letters in the word) will be L. How can you tell when the word has been guessed correctly or when it is time to be hanged? You will need to keep track of the number of spaces NL that have been correctly filled in. Also you will need the number of incorrect guesses. We will call this value NH. Each letter guessed will be stored in the string G$. You can determine whether a guessed letter G$ is in the word W$ by comparing G$ with each letter in W$ and noting where any matches occur. To let you know if any match occurred, define a flag R. Set R to 1 if any match occurs and set R to 0 if G$ is not in W$. We have now defined the variable names given in Figure 15.3.

We can now use these variable names to put a little more detail in the pseudocode description of the program given in Figure 15.2. For example, the *do until* loop can be rewritten in the form shown in Figure 15.4. Note that the word is guessed when NL=L, and you are hanged when NH=6. The algorithm to search for a letter in the word is given by the *for...next* loop. Note that this algorithm displays each correct letter in its proper position so that nothing more needs to be done in the *then* part of the subsequent *if...then...else* statement. Also note that the flag R is used to tell if a letter is in the word.

You have now developed the program to the point where you can begin to write some BASIC code. Since you have already done most of the work, the BASIC code will practically write itself.

Writing the Main Program

Your next step should be to write the main program in BASIC following the pseudocode description given in Figures 15.2 and 15.4. Try to write this *entire* program so that it fits on the screen and you can read it all at once. To do this, use subroutine calls for anything that takes a lot of coding and for anything you have not yet figured out how to do.

The main programs for hangman are shown in Figure 15.5. Line 20 clears the screen and prints **HANGMAN** in reverse video. Line 30 displays the gallows and the words **GUESS A LETTER** (subroutine 600) and finds a random word W$ (subroutine 1000). Line 50 finds the length, L, of the word W$ and moves the cursor to the first "blank" position. (Subroutine 500 is our usual "Move to X,Y" subroutine.) Each letter position will be separated from the next one by a space. Therefore, by computing the starting position (X) of the word by the equation X=10−L, each word will be approximately centered on the screen in the case of VIC 20. This will limit the

maximum length of words to 10. Line 60 prints the L spaces for the word to be guessed.

Lines 80-150 implement the *do until* loop shown in Figure 15.4. (The initialization of S$ to the null string in line 80 was added when the subroutine to guess a letter in line 400 was written.) Line 100 moves the cursor to the position of the blinking cursor following the words **GUESS A LETTER** and then calls subroutine 400 to guess a letter G$. Line 110 moves the cursor to the first letter position in the word and sets the flag R to 0. Lines 120-135 implement the *for...next* loop given in Figure 15.4. Lines 140-150 implement the last *if...then...else* statement in Figure 15.4. Subroutine 900, called in line 150, will display **NO "letter"** and add a part of the body.

Lines 160-180 implement the last *if...then...else* statement in Figure 15.2. We have actually interchanged the roles of *then* and *else*. That is, line 170 is equivalent to *if* word is *not* guessed. Subroutine 300, called in line 170, will display the correct word. Subroutine 700, called in line 180, will blink the word. Line 190 will ask **PLAY AGAIN?** and then get G$ from a blinking cursor in subroutine 800. Line 200 will cause a new game to be played if the answer to **PLAY AGAIN?** is Y.

We have now written a complete main program that implements the hangman algorithm given in Figure 15.2 We have also identified all of the subroutines that must still be written. These are summarized in Figure 15.6.

Figure 15.5 Main programs for Hangman: (a) VIC 20 and (b) Commodore 64.

```
10 REM  HANGMAN
20 PRINT"■";SPC(7);"■HANGMAN"
30 GOSUB 600:GOSUB 1000
50 L=LEN(W$):Y=5:X=10-L:GOSUB 500
60 FOR I=1 TO L:PRINT"▔ ";:NEXT
80 NL=0:NH=0:S$=""
90 IF NL=L OR NH=6 THEN 160
100 Y=7:X=15:GOSUB 500:GOSUB 400
110 Y=4:X=10-L:GOSUB 500:R=0
120 FOR I=1 TO L:IF G$=MID$(W$,I,1) THEN PRINT G$;"■";:NL=NL+1:R=1:GOTO 135
130 PRINT"■■";
135 NEXTI
140 IF R=1 THEN 90
150 NH=NH+1:GOSUB 900:GOTO 90
160 Y=20:X=6:GOSUB 500
170 IF NH=6 THEN PRINT"YOU ARE HANGED!":GOSUB 300:GOTO 190
180 PRINT"YOU ARE SAVED":GOSUB 700
190 Y=22:X=6:GOSUB 500:PRINT"PLAY AGAIN? ";:GOSUB 800
200 IF G$="Y" THEN 20
210 END

READY.
```

```
10 REM  HANGMAN
20 PRINT"■";SPC(15);"■HANGMAN"
30 GOSUB 600:GOSUB 1000
50 L=LEN(W$):Y=5:X=10-L:GOSUB 500
60 FOR I=1 TO L:PRINT"▔ ";:NEXT
80 NL=0:NH=0:S$=""
90 IF NL=L OR NH=6 THEN 160
100 Y=7:X=15:GOSUB 500:GOSUB 400
110 Y=4:X=10-L:GOSUB 500:R=0
120 FOR I=1 TO L:IF G$=MID$(W$,I,1) THEN PRINT G$;"■";:NL=NL+1:R=1:GOTO 135
130 PRINT"■■";
135 NEXTI
140 IF R=1 THEN 90
150 NH=NH+1:GOSUB 900:GOTO 90
160 Y=20:X=12:GOSUB 500
170 IF NH=6 THEN PRINT"YOU ARE HANGED!":GOSUB 300:GOTO 190
180 PRINT"YOU ARE SAVED":GOSUB 700
190 Y=22:X=12:GOSUB 500:PRINT"PLAY AGAIN? ";:GOSUB 800
200 IF G$="Y" THEN 20
210 END

READY.
```

The next step is to write the *minimum* amount of code in each subroutine that will allow you to run and test the main program. This *stub* can be nothing more than a RETURN statement that does nothing but return to the main program. Once you are certain that the main program is behaving properly, you can then write and test each subroutine separately. They can then be tested by running the *main* program to call the subroutines. This technique of *top-down programming* allows you to plan the entire program and begin to test it before you are involved in all the details of every subroutine. It also keeps your program well-modularized, which will make it much easier for you to debug and modify the program.

Figure 15.6 List of subroutines called from main program.

Line No.	Subroutine
600	Initial display (GUESS A LETTER, gallows)
1000	Find a Word, W$
500	Move to X,Y
400	Guess a Letter, G$
900	Wrong Guess—NO "letter", add to body
300	Print correct word
700	Blink word
800	Get G$ from blinking cursor

Writing the Subroutines

We have already written subroutine 500 and used it many times. It is shown in Figure 15.7 and should be added so that the main program can move the cursor to the proper location on the screen.

For subroutine 600, printing **GUESS A LETTER** is easy enough, but drawing the gallows will take more thought. Therefore, for now, type these statements for subroutine 600 and worry about drawing the gallows later:

```
600 REM INITIAL DISPLAY
610 X=0: Y=7: GOSUB 500
620 PRINT "GUESS A LETTER"
630 RETURN
```

In order to test the main program, you should store a known word in W$. Therefore, for subroutine 1000 type the following statements, which will assign the word HANGMAN to W$:

```
1000 REM FIND A WORD
1010 W$="HANGMAN"
1020 RETURN
```

Figure 15.7 Subroutine to "Move to X,Y".

```
500 REM  MOVE TO X,Y ON SCREEN
510 PRINT"■";:IF Y=0 THEN 530
520 FOR I=1 TO Y:PRINT:NEXT
530 PRINT TAB(X);:RETURN

READY.
```

It is a good idea to pick a test word that contains multiple occurrences of a single letter in order to make sure that the main program displays all letters in their proper locations. Later, you can come back and make subroutine 1000 produce random words.

Subroutine 800 will use the GET statement to get G$ from a blinking cursor. We have done this before, so we will just use the subroutine shown in Figure 15.8. Note that line 840 will print the value of the key pressed at the location of the blinking cursor and then back up the cursor so that it remains at the location at which the letter was printed. This will be important if you want to call subroutine 800 again without redefining the location of the cursor. (You will want to do this in subroutine 400.)

Figure 15.8 Subroutine to GET G$ from a blinking cursor.

```
800 REM  GET G$ FROM BLINKING CURSOR
810 PRINT"■ ■";:FOR I=1 TO 200:NEXT
820 GET G$:IF G$<>"" THEN 840
830 PRINT" ■";:FOR I=1 TO 200:NEXT:GOTO 810
840 PRINT G$;"■";:RETURN

READY.
```

Subroutine 400 is supposed to guess a letter, G$. At first this looks just the same as subroutine 800. However, you do not want to allow letters that have already been guessed. (Otherwise, you could hang yourself by typing the same wrong letter six times.) Therefore, subroutine 400 must keep track of all letters that have been typed and return only new values for G$. We will figure out how to do this later. For now just type the following statements to get a letter G$ by calling subroutine 800:

```
400 REM GUESS A LETTER
410 GOSUB 800: REM GET A LETTER G$
420 RETURN
```

For subroutine 900 type:

```
900 REM WRONG GUESS
920 PRINT "NO "; G$
930 RETURN
```

You know this will be printed at the wrong location on the screen, but it will help test the main program. You

can change its location later and figure out how to add a new part to the body each time there is a wrong guess.

For subroutines 300 and 700 just type stubs for now:

300 REM PRINT CORRECT WORD
310 RETURN

and

700 REM BLINK WORD
710 RETURN

With this much of the program written, you can run the main program and test to see if it is working properly. Figure 15.9 shows what the screen might look like during such a test.

After you have debugged the main program (it will not work the first time—Figure 15.5 was *not* the first version), you are ready to tackle the remaining subroutines one by one.

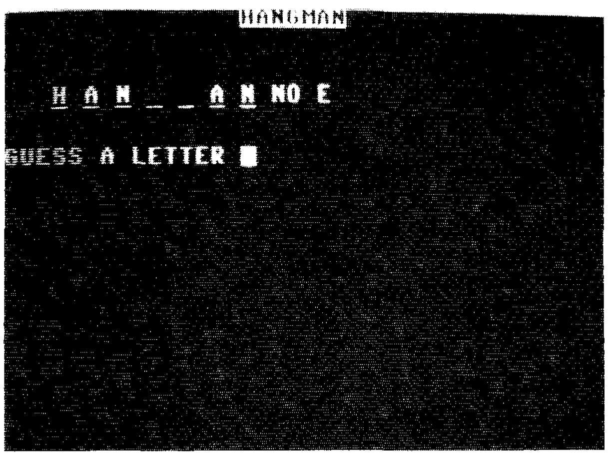


Figure 15.9 Testing the main program of Hangman.

Writing the Remaining Subroutines

You can now go through and finish the subroutines listed in Figure 15.6. Figure 15.10 is a listing of subroutine 600, where lines 630-660 have been added to draw the gallows shown in Figure 15.1.

The "guess a letter" subroutine (400) is shown in Figure 15.11. Lines 420-440 have been added to insure that no letter is guessed more than once. Line 440 keeps track of all letters that have been guessed by adding each new letter to the string S$. (This is why we initialized S$ to the null string "" in line 80 of the main program.) Each time line 410 GETs a net letter G$, it is compared with all previous letters (stored in S$) by the loop in lines 420-435. If a match is found in line 430, the program GETs a new letter in line 410.

The "wrong guess" subroutine (900) is shown in Figure 15.12. Lines 910-920 print **NO "letter"** at the

Figure 15.10 Initial display subroutine that draws gallows. (a) VIC 20 and (b) Commodore 64.

```
600 REM  INITIAL DISPLAY
610 X=0:Y=7:GOSUB 500
620 PRINT"GUESS A LETTER"
630 X=13:Y=9:GOSUB 500
640 PRINT"//////■■■■■■■■▔▔▔T▔▔▔L/";
650 FOR I=1 TO 7:PRINT"■■■ L/";:NEXT
660 PRINT"//■■■■▔▔▔▔"
670 RETURN

READY.
```

```
600 REM  INITIAL DISPLAY
610 X=0:Y=7:GOSUB 500
620 PRINT"GUESS A LETTER"
630 X=30:Y=9:GOSUB 500
640 PRINT"//////■■■■■■■■▔▔▔T▔▔▔L/";
650 FOR I=1 TO 7:PRINT"■■■ L/";:NEXT
660 PRINT"//■■■■▔▔▔▔"
670 RETURN

READY.
```

Figure 15.11 The subroutine to guess a letter.

```
400 REM  GUESS A LETTER
410 GOSUB 800:REM  GET A LETTER G$
420 FOR J=1 TO LEN(S$)
430 IF G$=MID$(S$,J,1) THEN 410
435 NEXTJ
440 S$=S$+G$:RETURN

READY.
```

appropriate location on the screen. Lines 930-985 add the appropriate part of the body to the hanging person. Note the use of the ON...GOTO statement (which is similar to the ON...GOSUB statement, except that it is an absolute branch) to add the appropriate part of the body.

Subroutine 300 shown in Figure 15.13 prints the correct word above the spaces when the person is hanged. Note the use of the function MID$ in line 330 to print each letter of the word W$ followed by a blank space.

Subroutine 700 shown in Figure 15.14 blinks the word by alternately displaying it ten times, first in normal and then in reverse video mode.

If all of the above subroutines are working properly, you can start adding some new random words in subroutine 1000. There are several ways to do this. One possibility is shown in Figure 15.15. Line 1010 sets a random seed for the random number generator. Line 1020 defines the number of words NW stored in the DATA statements starting at line 1100. You can add more words and increase the value of NW. In line 1030, X is assigned a random number between 1 and NW. Line 1040 moves the "pointer" to the beginning of the DATA statement. Line 1050 reads

the first X words. Therefore, word X will end up in W$. Note that with this subroutine the same word can occur more than once. If you want to avoid this, you will have to keep track of the values of X that have been used and not use the same ones more than once (see Exercise 15-1). Of course, you will then only be able to play ten times in one run of the program.

Sample runs of this program on the Commodore 64 are shown in Figure 15.16.

To make this game really interesting, you will need a large dictionary of possible words so that you can use different words each time you play. One way to do this is to store a large number of words on the cassette tape or floppy disk and then read in a random group of words each time the game is played.

Figure 15.12 The "wrong guess" subroutine prints NO "letter" and adds a part to the body: (a) VIC 20 and (b) Commodore 64.

(a)
```
900 REM  WRONG GUESS
910 X=1:Y=7:FOR J=1 TO NH:Y=Y+2:NEXT:GOSUB 500
920 PRINT"NO ";G$
930 X=14:Y=11:GOSUB 500
940 ON NH GOTO 960,965,970,975,980,985
960 PRINT" ◠■■■◡":RETURN
965 PRINT"■■┬■■■■ ":RETURN
970 PRINT"■■■/■■ ":RETURN
975 PRINT"■■■■\■■ ":RETURN
980 PRINT"■■■■/■■ ":RETURN
985 PRINT"■■■■■\■■ ":RETURN

READY.
```

(b)
```
900 REM  WRONG GUESS
910 X=1:Y=7:FOR J=1 TO NH:Y=Y+2:NEXT:GOSUB 500
920 PRINT"NO ";G$
930 X=31:Y=11:GOSUB 500
940 ON NH GOTO 960,965,970,975,980,985
960 PRINT" ◠■■■◡":RETURN
965 PRINT"■■┬■■■■ ":RETURN
970 PRINT"■■■/■■ ":RETURN
975 PRINT"■■■■\■■ ":RETURN
980 PRINT"■■■■/■■ ":RETURN
985 PRINT"■■■■■\■■ ":RETURN

READY.
```

Figure 15.13 This subroutine prints the correct word above the spaces.

```
300 REM  PRINT CORRECT WORD
310 PRINT"■■THE WORD IS"
320 Y=3:X=10-L:GOSUB 500
330 FOR I=1 TO L:PRINT MID$(W$,I,1);" ";:NEXT
340 RETURN

READY.
```

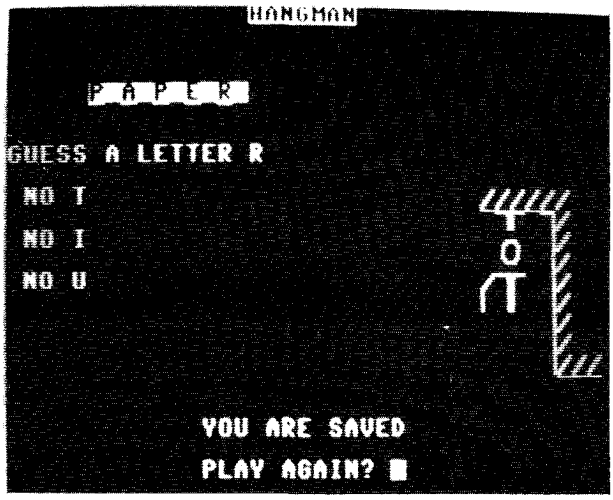
Figure 15.14 This subroutine blinks the word.

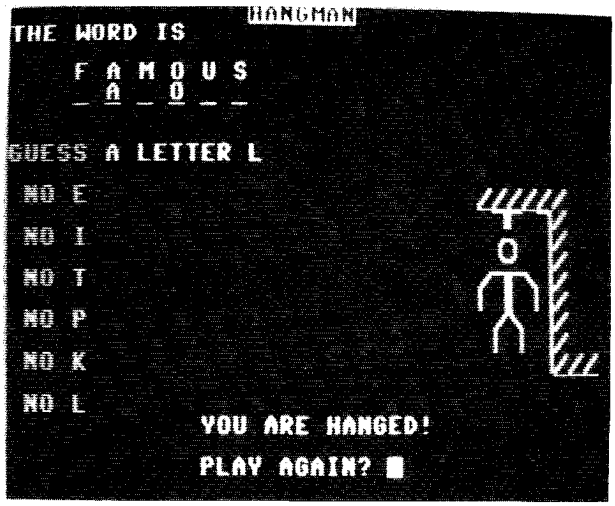
```
700 REM  BLINK WORD
710 FOR J=1 TO 10
720 Y=4:X=10-L:GOSUB 500
730 FOR I=1 TO L:PRINT MID$(W$,I,1);" ";:NEXT
740 X=10-L:GOSUB 500
750 FOR I=1 TO L:PRINT"■";MID$(W$,I,1);" ";:NEXT
760 NEXTJ:RETURN

READY.
```

Figure 15.15 Subroutine that finds one of ten words at random.

```
1000 REM  FIND A WORD
1010 X=RND(-TI)
1020 NW=10
1030 X=INT(RND(1)*NW+1)
1040 RESTORE
1050 FOR I=1 TO X:READ W$:NEXT
1060 RETURN
1100 DATA HYPOTENUSE,NURSE,FAMOUS,EMPIRE,ELK,DIGNITY
1110 DATA CONDITION,BRIBE,PAPER,QUAIL

READY.
```

Figure 15.16 Sample runs of Hangman program.





STORING DATA ON A CASSETTE TAPE OR FLOPPY DISKETTE

You have been using the cassette tape recorder and/or floppy disk drive to save and load your BASIC programs. It is also possible to include statements in your programs that will allow you to store data on a tape or diskette and later read back this data. (This may or may not be the same tape/diskette that contains your program.) You do this with the PRINT# and INPUT# statements. In order to use these statements, you must also use the OPEN and CLOSE statements.

The OPEN and CLOSE Statements

Before you can *write* data to a tape or diskette your program must execute the statement **OPEN K,N,1, "string"** where **K** is any number between 1 and 255, *string* is a filename of your choice, and **N** is either 1 or 8 depending upon whether you are storing data on a tape or diskette. Before you can *read* the data back, you must execute the statement **OPEN K,N,0 "string"**. For example, to write data on a tape, you could use

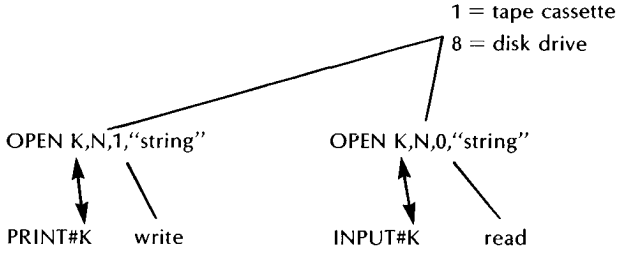
OPEN 1,1,1,"DATA"

and, to read a diskfile called KEYWORDS, you could use

OPEN 8,8,0,"KEYWORDS"

The number K above is called a logical file number, and it must be the same as the number used in the PRINT# and INPUT# statements. Figure 15.17 shows the meaning of the three numbers in the OPEN statement.

Figure 15.17 Meanings of the numbers in the OPEN statement.



The second number in the OPEN statement refers to a particular physical device. A 1 means that the data will be written to (or read from) the cassette tape unit. An 8 means the disk drive. Other physical devices use different numbers, as shown in Table 15.1.

TABLE 15.1 Physical Device Numbers

Physical Device	*Device Number*
keyboard	0
cassette tape unit	1
RS232 serial device	2
screen	3
printer	4 or 5
disk drive	8 or 9
serial device	4-127

If the physical device number is omitted from an OPEN statement then, by default, the OPEN statement will refer to the tape cassette unit. Also, when writing on or reading from tape (not the disk), the filename is optional and may be omitted. In this case, when reading from tape, the first file found will be read.

The third number in an OPEN statement is called the secondary address code, and its meaning depends upon the physical device being used (as defined by the second number in the OPEN statement). If the physical device is the tape cassette unit, the secondary address codes have the meanings shown in Table 15.2. If the secondary address code is omitted, a *read* operation is implied.

TABLE 15.2 Secondary Address Code for Tape Cassette Units

Secondary Address Code	*Meaning*
0 (default)	OPEN for read
1	OPEN for write
2	OPEN for write and add end-of-tape (EOT) mark when file is CLOSEd

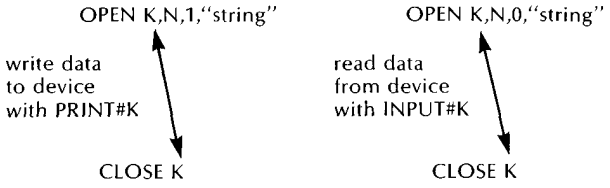
When a program executes the statement **OPEN 1,1,1,"string"** the Commodore 64/VIC 20 will display the following message on the screen (assuming no keys on the cassette unit are pressed): **PRESS RECORD & PLAY ON TAPE.** When you press these keys, the Commodore 64/VIC 20 will display **OK,** write a header on the tape, and then stop the tape. During this time, the Commodore 64 (only) blanks the screen. If you keep the PLAY and RECORD keys depressed, the Commodore 64/VIC 20 will automatically write data on the tape in response to PRINT# statements, as described below.

When the program executes the statement **OPEN 1,1,0,"string"** the Commodore 64/VIC 20 will display the following message on the screen (assuming no keys on the cassette unit are pressed): **PRESS PLAY ON TAPE.** When you press this key (or any key for that matter, since the Commodore 64/VIC 20 cannot tell which key you pressed), the Commodore 64/VIC 20 will display **OK** and the Commodore 64 will blank the screen. Then, it reads the tape until it finds the filename "*string*" on a previously written tape header, and then stops the tape. (If you omit the filename, it will stop at the first previously written tape header it finds.) If you keep the PLAY key depressed, the Commodore 64/VIC 20 automatically reads data from the tape in response to INPUT# statements, as described below.

When a program executes the statement **OPEN 8,8,1,"string"** the Commodore 64/VIC 20 creates a channel for communication to the disk drive and a header with the name "*string*" on a diskfile. When PRINT# statements are encountered, it transmits the associated data to the "*string*" diskfile. When the program executes the statement **OPEN 8,8,0,"string"** the Commodore 64/VIC 20 creates a channel for communication to the disk drive and looks for a diskfile with the filename "*string*". When INPUT# statements are encountered, it retrieves the necessary data from the "*string*" diskfile.

After a program has written data to (or read data from) either tape or diskette, the logical file that was opened with the OPEN statement must be closed with a CLOSE statement of the form **CLOSE K.** In this statement the number K is the logical file number and must agree with the logical file number used in the corresponding OPEN statement, as shown in Figure 15.18. The CLOSE statement will put an end-of-file (EOF) mark on the tape and close the communications channel to the disk.

Figure 15.18 Logical file numbers must be the same in OPEN and CLOSE statements.



The PRINT# Statement

After the statement **OPEN K,N,1"string"** is executed your program can write string data to the tape or diskette with the statement **PRINT#K,W$.** When writing this statement the question mark (?) *cannot* be used as an abbreviation for PRINT. The statement **PRINT#K** must be typed as shown with *no* blanks. You can also write numerical data to the cassette tape

by using a numerical variable name in the PRINT# statement.

The program shown in Figure 15.19 can be used to store a list of words on a tape or diskette for later use in the hangman program. Line 20 solicits a device number (1=tape or 8=disk) and a filename from the keyboard. Line 30 OPENs a logical file for writing. Line 40 displays the message **ENTER WORDS; TYPE £ TO STOP.** Line 50 waits for a word to be typed in and assigns this word to the string variable W$. If you type a poundsign (£), line 40 will branch to line 90, which CLOSEs the file and stops the program.

Figure 15.19 Program to store words on a tape or diskette as they are typed on the keyboard.

```
10 REM  STORE WORDS ON TAPE
20 INPUT "DEVICE#,FILENAME";N,F$
30 OPEN 1,N,1,F$
40 PRINT "ENTER WORDS; TYPE £ TO STOP"
50 INPUT W$
60 IF W$="£" THEN 90
70 PRINT#1,W$
80 GOTO 50
90 CLOSE 1
100 END

READY.
```

Line 70 writes the word stored in W$ to the tape or diskette. Actually it stores this word in a buffer (a section of memory). The Commodore 64/VIC 20 waits until the buffer is full (or until a CLOSE statement is executed) before writing all the characters in the buffer out to the tape or diskette.

Line 80 branches to line 50 which waits for another word to be entered. You can type in many words before the tape moves, because no writing is done until the buffer is full. If you do not fill up the buffer before you type the poundsign (£), the contents of the buffer are written to the tape when the CLOSE statement is executed.

A sample run of the program given in Figure 15.19 is shown in Figure 15.20. Once a long list of words has been stored on a tape or diskette, it can be used with the hangman program. The program can be modified, as described in the next section, to retrieve a different random set of words each time it is run.

The INPUT# Statement

After the statement **OPEN K,N,0, "string"** is executed, your program can read string data from a tape or diskette with the statement **INPUT#K,W$.** You can read numerical data from a cassette tape by using a numerical variable name in the INPUT# statement.

```
READY.
RUN
DEVICE#, FILENAME? 1,DATA
PRESS RECORD & PLAY ON TAPE
OK
ENTER WORDS; TYPE £ TO STOP
? HOUSE
? CAR
? COMPUTER
? BOAT
? EXCELLENT
? £

READY.
```

Figure 15.20 Sample run of program in Figure 15.19 that stores five words on a cassette tape.

As an example of using the INPUT# statement, the program shown in Figure 15.21 will read back the data that was stored in Figure 15.20. Line 20 solicits the device number and a filename. Line 30 OPENS a logical file for reading. Line 40 reads one string from the tape and stores it in the string variable W$.

Figure 15.21 Program to read data that was stored using the program in Figure 15.19.

```
10 REM  READ DATA
20 INPUT "DEVICE#,FILENAME";N,F$
30 OPEN 1,N,0,F$
40 INPUT#1,W$
50 PRINT W$
60 IF ST<>64 THEN 40
70 CLOSE 1
80 END

READY.
```

The Commodore 64/VIC 20 has an input/output *status word* whose value at any time is given by ST. The value of ST depends on the result of the last input or output operation performed. It is useful to know that if the Commodore 64/VIC 20 reads an end-of-file (EOF) mark, the value of ST will be set to 64. This fact can be used, as shown in line 60 of Figure 15.21, to determine when all the data has been read.

Each string read is automatically PRINTed in line 50. If an end-of-file (EOF) mark is not read, line 60 branches to line 40, which reads the next string. The result of executing this program, using the data tape created in Figure 15.20, is shown in Figure 15.22.

Suppose that you have created a data tape or diskette containing a large number, NT, of hangman words, using the program given in Figure 15.19. (Assume NT=60.) How can a different random set of ten of these words be used each time hangman is run? One possible method is suggested in Figure 15.23.

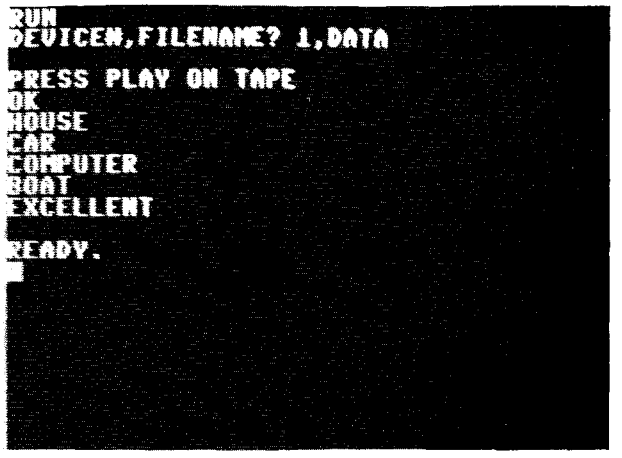


Figure 15.22 Result of running the program in Figure 15.21 using the data tape/diskette created in Figure 15.20.

Ten words from the tape will be stored in the array W$(I). In order to store a different set of words each time, a random number of words, R, will be skipped for each word that is actually stored in the array W$(I), as shown in Figure 15.23.

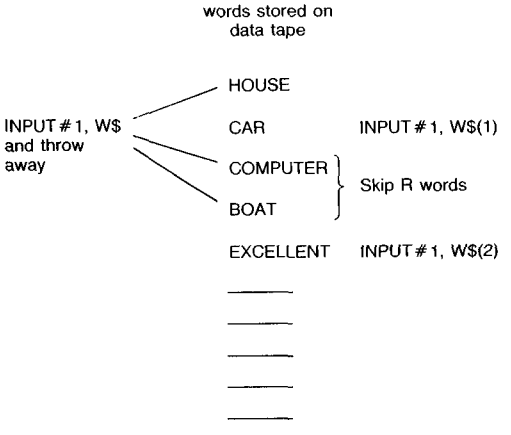


Figure 15.23 To produce a new random set of words, skip R (a random number) words between each word stored in the array W$(I).

In order to skip a word you must read the word and discard it. You can do this with a statement such as **INPUT#1,W$.** To keep a word, put it into the array with the statement **INPUT#1,W$(I).**

What range of values should the random number R have? If you pick N words (10), equally spaced, from a total of NT (say 60) words, then you will pick one out of every NT/N words and there will be NT/N−1 words for every one selected. If you picked a random number R between 1 and NT/N−1, you would seldom get much beyond the middle of the list before N words are selected. This is because the average of all the R values will be about (NT/N−1)/2. Therefore, in order to spread the N words out over all of the NT words, you can let R be a random number between 1 and 2*(NT/N−1).

To incorporate these ideas into the hangman program add the lines **12 INPUT "ENTER STORAGE DEVICE# & WORD FILENAME"; N1,F1$** and **15 N=0:I=RND(−TI):DIM W$(50)** to the main program shown in Figure 15.5. Then replace the "find a word" subroutine given in Figure 15.15 with the subroutine shown in Figure 15.24. The first time this subroutine is called (or after N words have been used), the value of N will be 0 and lines 1020-1095 will be executed. These lines read another N random words from the data tape/diskette.

Line 1020 computes the value W=2*(NT/N−1), which will be the maximum value of the random number R corresponding to the number of skipped words on the tape. Line 1030 reminds the user to load and rewind the data tape if the device number is 1, and line 1040 then waits until any key is pressed. Line 1050 opens a logical file for reading.

The loop in lines 1055-1085 will read N words into the array W$(I) (line 1080). After each word is stored in the array W$(I), R words will be read and discarded by the loop in lines 1065-1075. A different random value of R between 1 and W is computed in line 1060 every time through the outer loop.

Because R is "random," its value may often be large enough to get to the end of the data file before N words are stored in W$(I). To test for this possibility the status word ST is checked each time a string is read in lines 1070 and 1080. If an end-of-file (EOF) mark was read at line 1070 (ST=64), then the number of words read up to that point (I−1) is stored in N, and the program branches to line 1090.

Line 1090 erases the messages associated with reading the tape. Line 1095 closes the data file, and the subroutine continues at line 1100.

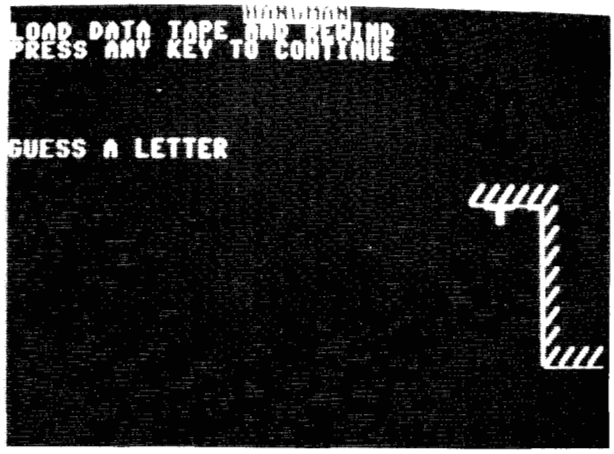
Once N words have been read into the array W$(I), then up to N hangman games can be played before the tape has to be read again. If the value of N is not 0 when subroutine 1000 is entered, the program branches immediately to line 1100, where a random number I between 1 and N is computed. The new word W$ to be used in hangman is the word stored in W$(I). Line 1110, after assigning W$(I) to W$, replaces the word in location I (which has just been used) with the word in location N. The value of N is then decremented by one in line 1120. This will insure that no word is used more than once, since each word is effectively removed from the list as it is used.

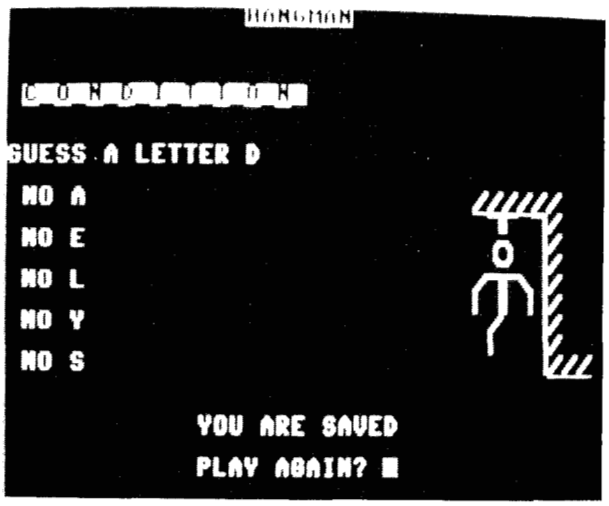
A sample run of the modified hangman program using the subroutine in Figure 15.24 is shown in Figure 15.25.

Figure 15.24 Modified Hangman subroutine that will find a word from a collection of words stored on a data tape or diskette.

```
1000 REM  FIND A WORD
1010 IF N<>0 THEN 1100
1020 NT=60:N=10:W=2*(NT/N-1)
1025 IF N1=8 THEN 1050
1030 PRINT"■■LOAD DATA TAPE AND REWIND"
1035 PRINT"PRESS ANY KEY TO CONTINUE"
1040 GET G$:IF G$="" THEN 1040
1050 OPEN 1,N1,0,F1$
1055 FOR I=1 TO N
1060 R=INT(W*RND(1)+1)
1065 FOR J=1 TO R
1070 INPUT#1,W$:IF ST=64 THEN N=1-1:GOTO 1090
1075 NEXTJ
1080 INPUT#1,W$(I):IF ST=64 THEN N=1:GOTO 1090
1085 NEXTI
1090 PRINT"■■";:FOR I=1 TO 5:PRINT"                                    ":NEXT
1095 CLOSE 1
1100 I=INT(N*RND(1)+1)
1110 W$=W$(I):W$(I)=W$(N)
1120 N=N-1:RETURN

READY.
```

Figure 15.25 Sample run of Hangman program using words stored on a cassette data tape.





C64/VIC ORGAN

As another example of developing a BASIC program, we will turn the Commodore 64 and VIC 20 into musical instruments. First, we will figure out how to play the notes of the scale by pressing keys on the keyboard, and then we will develop a complete program that will display the musical keyboard on the screen.

Playing a Tone While a Key is Pressed

Review the section in Chapter 14 on playing scales with either the VIC 20 or the Commodore 64, whichever is appropriate. Also, recall from Figure 14.3 that **PEEK(197)** will be equal to 64 if no key is pressed. Therefore, if you turn on a tone you can keep it on as long as the key is pressed by using the delay loop **1530 IF PEEK(197)<>64 THEN 1530.** This will require that one key be completely released before another one is pressed. Since pressing a second key will normally change the value in location 197, the two statements below will produce a delay as long as the same

1525 K1=PEEK(197)

1530 IF PEEK(197)=K1 AND PEEK(197)<>64 THEN 1530

key (and only that key) is being depressed. You may think that the condition PEEK(197)<>64 is not needed in 1530. However, if you press and release a key quickly, the program to be described will enter a subroutine that turns on the tone. By the time

statement 1525 is executed, you may have released the key, in which case K1=64. Without the second condition in line 1530 the tone would stay on even though no key is being pressed.

Initialization of the sound and subroutines to play single tones on both the VIC 20 and Commodore 64 were discussed in Chapter 14. In Figure 15.26 are two subroutines each for the VIC 20 and Commodore 64 that initialize the sound and play a single tone as long as a key is held down. The subroutines at line 1600 initialize the sound and, in particular, turn up the volume. The subroutines at 1500 play a single note with pitch P as long as a keyboard key is pressed. The subroutine for the VIC 20 uses the soprano voice (voice #2) and the subroutine for the Commodore 64 is initialized to produce an organ sound (see Figure 14.42).

To test the VIC subroutine, type **GOSUB 1600:P=195:GOSUB 1500.** To test the Commodore 64 subroutine, type **GOSUB 1600:P=4291:GOSUB 1500.** The tone should last as long as you hold down the return key.

Pitch Values for the Musical Scale

To turn the VIC 20 and Commodore 64 into musical instruments, we must know what pitch values P correspond to the notes of the musical scale. In Figure 15.27 are given the pitch values recommended by Commodore to produce a midrange three octave C scale on each computer. Lines 310, 320, and 330 contain the ascending pitch values for notes C, D, E, F, G, A, and B (plus C, D, and E only on the Commodore 64) in successively higher octaves. Lines 311, 321, and 340 contain the pitch values for the chromatics C#, D#, F#, G#, and A# (plus C# and D# on the Commodore 64) corresponding to the same respective octaves.

Screen Layout

The VIC 20 program we will write will display seven white keys and five black keys (exactly one octave) according to the layout shown in Figure 15.28a. The Commodore 64 program will display ten white keys and seven black keys as shown in layout of Figure 15.28b. The name of the Commodore 64/VIC 20 key corresponding to each key on the screen will be printed on each key. The name of the musical note for each white key will also be printed.

When a note is played, an asterisk "pointer" will be displayed on the key being played for as long as the tone continues. Thus, as a song is played this "pointer" will move from key to key.

Figure 15.26 (a) VIC 20 and (b) Commodore 64 subroutines to produce a tone with pitch P while key is being pressed.

```
1500 REM  PRODUCE TONE WHILE KEY IS PRESSED
1510 POKE 36876,P
1520 K1=PEEK(197)
1530 IF PEEK(197)=K1 AND PEEK(197)<>64 THEN 1530
1540 POKE 36876,0:RETURN
1600 REM  INITIALIZE SOUND
1610 POKE 36878,15
1620 RETURN

READY.
```

```
1500 REM  PRODUCE TONE WHILE KEY IS PRESSED
1510 PH=INT(P/256):PL=P-256*PH
1520 POKE RA,PL:POKE RA+1,PH
1530 POKE RA+4,WF+1
1540 K1=PEEK(197)
1550 IF PEEK(197)=K1 AND PEEK(197)<>64 THEN 1550
1560 POKE RA+4,WF:RETURN
1600 REM  INITIALIZE SOUND
1605 RA=54272
1610 FOR I=0 TO 23:POKE RA+I,0:NEXT I
1620 POKE RA+24,15
1630 WF=16:AT=1:DE=15:SU=15:RE=0:
1640 POKE RA+5,16*AT+DE:POKE RA+6,16*SU+RE
1650 RETURN

READY.
```

Figure 15.27 Pitch values for a three octave scale on (a) the VIC 20 and (b) the Commodore 64.

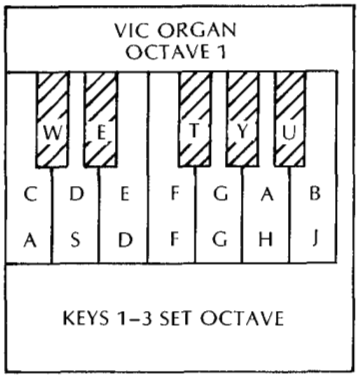
(a)
```
300 REM PITCH TABLE
310 DATA 135,147,159,163,175,183,191
311 DATA 143,151,167,179,187
320 DATA 195,201,207,209,215,219,223
321 DATA 199,203,212,217,221
330 DATA 225,228,231,232,235,237,239
340 DATA 227,229,233,236,238

READY.
```

(b)
```
300 REM PITCH TABLE
310 DATA 2145,2408,2703,2864,3215,3608,4050,4291,4817,5407
311 DATA 2273,2551,3034,3406,3823,4547,5103
320 DATA 4291,4817,5407,5728,6430,7217,8101,8583,9634,10814
321 DATA 4547,5103,6069,6812,7647,9094,10207
330 DATA 8583,9634,10814,11457,12860,14435,16203,17167,19269,21629
340 DATA 9094,10207,12139,13625,15294,18188,20415

READY.
```

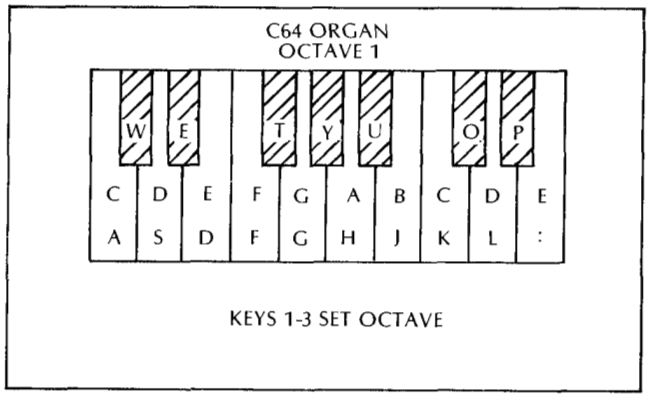
Both organs will start out in octave 1. Pressing key 2 will change them to octave 2 and pressing key 3 will change them to octave 3. Thus, the total range of the VIC organ will be exactly three octaves and that of the Commodore 64 more than three octaves. In the case of the Commodore 64, for example, pressing key A on the keyboard when in the octave 2-mode will produce the same note as pressing key K when in the octave-1 mode. The octave number that is active is displayed above the keyboard on the screen.

Figure 15.28 Screen layouts for (a) the VIC organ and (b) the C64 organ.

(a)



(b)



Word-Description of Program

The program for both organs can be understood from the pseudocode word-description given in Figure 15.29.

Figure 15.29 Pseudocode word-description for both organ programs.

```
        Initialize variables
        Display keyboard
loop:   Wait for key to be pressed
        Identify key
        if key is a note key
        then play note
        else if key is 1, 2, or 3
             then change octave number
repeat forever
```

Variable Definitions

The major variables used in the two organ programs are defined in Figure 15.30. Using these variables, the pseudocode program shown in Figure 15.29 can be refined to the one shown in Figure 15.31. Note how the *do until* loop is used to find the index I corresponding to the key that was pressed. This value of I is then used to find the proper pitch, which is stored in P(O,I) where O is the active octave number. If key 1, 2, or 3 is pressed, no match will occur in the *do until* loop and L

Figure 15.30 Major variables used in the two organ programs.

K$(18)	—an array containing the PET characters: A,S,D,F,G,H,J,K,L,: for the white keys W,E,T,Y,U,O,P for the black keys
P%(17)	—an integer array containing the pitch values for the corresponding keys in K$(I)
T(3)	—an array containing the timbre values 15, 51, and 85 for octaves 1, 2, and 3
T	—timbre value to store in Shift Register
P	—pitch value to store in Timer 2
A$	—character value corresponding to PET key pressed

Figure 15.31 Pseudocode description of both organ programs.

```
        Fill arrays K$(I), P%(I), and T(I) with
          appropriate values
        Set initial value of T to 15
        Display keyboard
loop:   Wait for key A$ to be pressed
        I=1
        do until A$=K$(I) or I=18
          I=I+1
```

will equal 12 in the case of the VIC 20 and 18 in the case of the Commodore 64. (Note that this is why K$(I) must be dimensioned as shown.) In this case I is changed to VAL(A$) (which will be 1, 2, or 3 for a valid key), and O is changed to I.

The Main Program

The main programs for the two organs are shown in Figure 15.32. They follow closely the pseudocode description shown in Figure 15.31. Line 20 initializes the sound. Lines 30-60 fill the arrays with the appropriate data. Note that all of the "white" keys are stored first in K$(I). Also note that the order of the pitch values stored in P(O,I) is the same as the corresponding key values in K$(I), according to Figure 15.27.

Lines 110-200 are a direct implementation of the *loop...repeat forever* loop in Figure 15.31. Subroutine 1400, called in line 200, will play the note. Code will be added to this subroutine to display the asterisk "pointer" on the screen keyboard before playing the note. However, for now you can test the playing of the organ by typing the following statements to call the tone-producing subroutine shown in Figure 15.26:

```
1400 REM DISPLAY POINTER
1440 GOSUB 1500: REM PRODUCE TONE
1450 RETURN
```

You will also need to write the following stub for the keyboard display subroutine:

```
600 REM DISPLAY KEYBOARD
610 RETURN
```

Try running the program now.

Remaining Subroutines

Once you have the musical part of the organ working, you can finish the keyboard display subroutine as shown in Figure 15.33. This subroutine will produce the same (respective) keyboard shown in Figure 15.34. The lettering on the keyboard is printed using subroutine 700 shown in Figure 15.35. Lines 710-740 print all the information on the screen. Lines 750-760 initialize some tab values stored in the array TB(I) that will be used to display the asterisk "pointer" on the black keys of the keyboard.

Subroutine 1400 shown in Figure 15.36 displays the asterisk "pointer" when a key is pressed. The value of I in line 1410 is the value found in the "do until" loop in lines 100-120 of the main program. If this value is greater than 7 in the case of the VIC 20 or 10 in the case of the Commodore 64, then a "black" key was pressed and its asterisk is printed by the statements starting in line 1460. The position at which this asterisk is printed by line 1470 is determined by the value in the tab array TB(1-7) for the VIC 20 and TB(I-10) for the Commodore 64 in line 1460. TB(I) was initialized in lines 750-760 of Figure 15.35. After the asterisk is displayed by line 1470, the note is played by line 1480 until the key is released. The asterisk is then erased by line 1485.

Lines 1420-1450 produce a similar effect for the white keys. Since the spacing of the white keys is uniform, the position of the asterisk on the line can be calculated by the equation K = 3*I−2 on the VIC 20 and by K = 3*I+3 on the Commodore 64. Examples of the asterisks displayed when the C64 organ is played are shown in Figure 15.37.

CONCLUSION

The Hangman and the C64/VIC organ programs were developed using the six steps outlined at the beginning of this chapter. This is not the only way to develop a program, and these steps may not always be appropriate for all the programs that you write. However, they are a good guide to use when you get stuck and do

Figure 15.32 Main programs for (a) the VIC 20 and (b) the Commodore 64 organs.

```
10 REM  VIC ORGAN
20 GOSUB 1600:REM INITIALIZE SOUND
30 DIM K$(13),P(3,12)
40 DATA A,S,D,F,G,H,J,W,E,T,Y,U
50 FOR I=1 TO 12:READ K$(I):NEXT
60 FOR I=1 TO 3:FOR J=1 TO 12:READ P(I,J):NEXT J:NEXT I
70 O=1
80 GOSUB 600:REM  DISPLAY KEYBOARD
90 GET A$:IF A$="" THEN 90
100 I=1
110 IF A$=K$(I) OR I=13 THEN 130
120 I=I+1:GOTO 110
130 IF I<13 THEN 200
140 I=VAL(A$):IF I<1 OR I>3 THEN 90
145 O=I:PRINT"■■■";SPC(13);I
150 GOTO 90
200 P=P(O,I):GOSUB 1400:GOTO 90

READY.
```

```
10 REM  C64 ORGAN
20 GOSUB 1600:REM INITIALIZE SOUND
30 DIM K$(18),P(3,17)
40 DATA A,S,D,F,G,H,J,K,L,":",W,E,T,Y,U,O,P
50 FOR I=1 TO 17:READ K$(I):NEXT
60 FOR I=1 TO 3:FOR J=1 TO 17:READ P(I,J):NEXT J:NEXT I
70 O=1
80 GOSUB 600:REM  DISPLAY KEYBOARD
90 GET A$:IF A$="" THEN 90
100 I=1
110 IF A$=K$(I) OR I=18 THEN 130
120 I=I+1:GOTO 110
130 IF I<18 THEN 200
140 I=VAL(A$):IF I<1 OR I>3 THEN 90
145 O=I:PRINT"■■■";SPC(21);I
150 GOTO 90
200 P=P(O,I):GOSUB 1400:GOTO 90

READY.
```

not know how to proceed. You will probably develop your own approach to writing computer programs. Programming is a skill that requires insight, creativity, a knack for problem-solving, and *practice*.

If you have read this entire book, typed in all the examples on your Commodore 64/VIC 20, and worked a good number of the exercises, you will have a good understanding of how to write BASIC programs on a Commodore 64/VIC 20. It is now time for you to start writing your own programs. A great many useful programs can be written for the Commodore 64/VIC 20. Pick an area in which you are an expert. How can the Commodore 64/VIC 20 help you in this area? Start by writing a short program, and then expand it into a longer, more complex program. You will find that writing computer programs is challenging, rewarding, and fun. Good luck!

EXERCISE 15-1

Modify the "find a word" subroutine in Figure 15.15 so that no word is selected more than once.

EXERCISE 15-2

Write a program to play the game MASTER MIND. The computer thinks of an N-digit number where each digit can be in the range 1-M. The player is allowed to select N and M at the beginning of the game. The player guesses a number (all N digits), and the computer responds with two numbers P and W. P is the number of digits that were correctly guessed that are in the correct position in the number, and W is the number of digits guessed that are in the number but were guessed in the wrong position. The player continues to guess numbers until the number is

Figure 15.33 Subroutine to display the keyboard for the (a) VIC and (b) C64 organs.

(a)
```
600 REM  DISPLAY KEYBOARD
610 U2$="TTTTTTTTTTT"
615 POKE 36879,11:PRINT "■"
620 PRINT"□";TAB(7);"■VIC ORGAN"
630 PRINT:PRINT TAB(7);"OCTAVE 1"
635 PRINT
640 FOR J=1 TO 12
645 FOR I=1 TO 7
650 PRINT"■  ";:NEXTI:PRINT:NEXTJ
655 B$="■■■■■■■■■■■■■■■■■■■■■"
660 S1$="▌▐▀▀▌▐▀▀▀▀▌▐▀▀▌▐▀▀▀"
665 S2$="▌▐  ▌▐  ▌▐▐▐▐▌▐  ▌▐  "
670 PRINT"■ ";U2$;S1$;B$;
680 FOR I=1 TO 5:PRINT S2$;B$;:NEXT
690 GOSUB 700:REM  DISPLAY LETTERING
695 RETURN

READY.
```

(b)
```
600 REM  DISPLAY KEYBOARD
605 RA=54272:REM MUSIC REGISTERS REFERENCE ADDRESS
610 U2$="TTTTTTTTTTT"
615 POKE 53280,5:POKE 53281,0:PRINT "■"
620 PRINT"□";TAB(15);"■C64 ORGAN"
630 PRINT:PRINT TAB(15);"OCTAVE 1"
635 PRINT
640 FOR J=1 TO 12:PRINT "     ";
645 FOR I=1 TO 10
650 PRINT"■  ";:NEXTI:PRINT:NEXTJ
655 B$="■■■■■■■■■■■■■■■■■■■■■■■■■■■■■"
660 S1$="▌▐▀▀▌▐▀▀▀▀▌▐▀▀▌▐▀▀▀▀▌▐▀▀▀"
665 S2$="▌▐  ▌▐  ▌▐▐▐▐▌▐  ▌▐  ▌▐▐▐▐▌▐  "
670 PRINT"■     ";;U2$;S1$;B$;
680 FOR I=1 TO 5:PRINT S2$;B$;:NEXT
690 GOSUB 700:REM  DISPLAY LETTERING
695 RETURN

READY.
```

Figure 15.34 *Keyboards for the (a) VIC 20 and (b)* Commodore 64 subroutines shown in Figure 15.33.

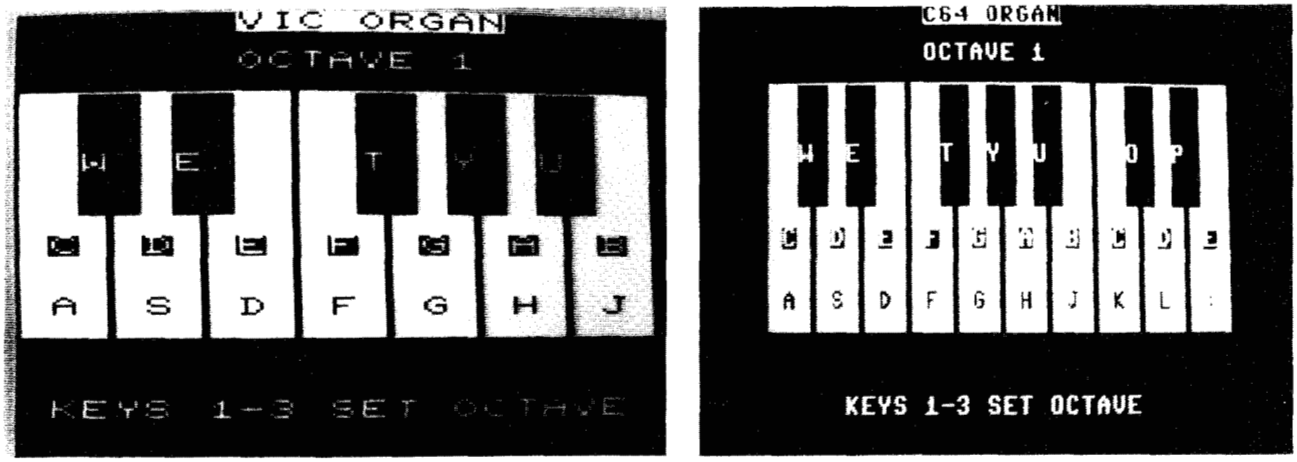


Figure 15.35 Subroutine to display the lettering on the (a) VIC 20 and (b) Commodore 64 organ keyboard.

(a)
```
700 REM  DISPLAY LETTERING
710 PRINT"
720 PRINT"
730 PRINT"
740 PRINT"    KEYS 1-3 SET OCTAVE"
750 DATA 2,5,11,14,17
760 FOR I=1 TO 5:READ TB(I):NEXT
770 RETURN

READY.
```

(b)
```
700 REM  DISPLAY LETTERING
710 PRINT"
720 PRINT"
730 PRINT"
740 PRINT"              KEYS 1-3 SET OCTAVE"
750 DATA 7,10,16,19,22,28,31
760 FOR I=1 TO 7:READ TB(I):NEXT
770 RETURN

READY.
```

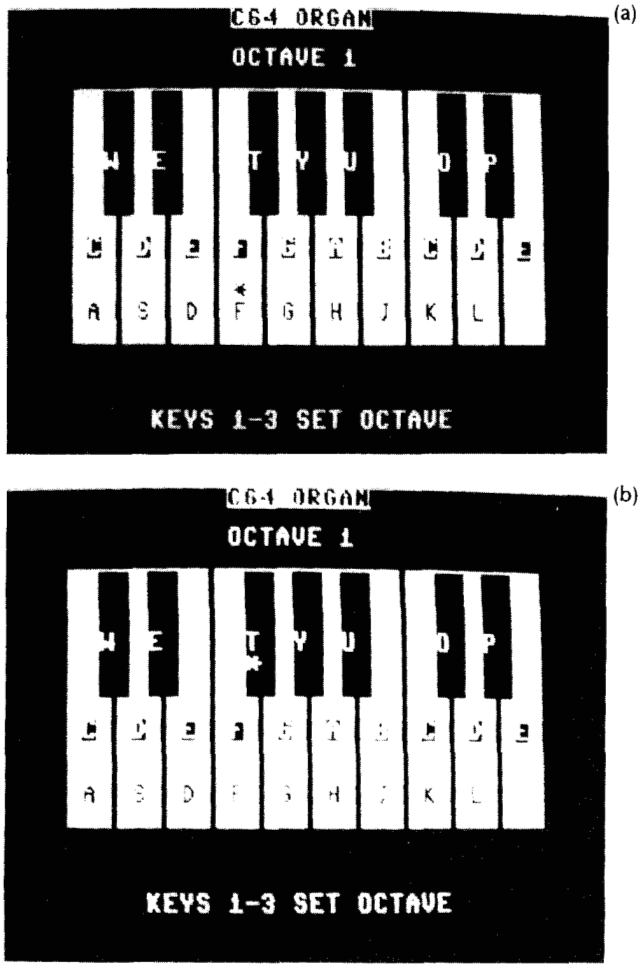
Figure 15.36 Subroutine to display the asterisk "pointer" and produce the tone on the (a) VIC 20 and (b) Commodore 64.

(a)
```
1400 REM  DISPLAY POINTER
1405 D8$="         ":D3$=D8$+"     "
1410 IF I>7 THEN 1460
1420 K=3*I-2
1430 PRINT"  ";D3$;SPC(K);"**  ";
1440 GOSUB 1500:REM  PRODUCE TONE
1450 PRINT"   ":RETURN
1460 K=TB(I-7)
1470 PRINT"  ";D8$;SPC(K);"*  ";
1480 GOSUB 1500
1485 PRINT"  ":RETURN

READY.
```

(b)
```
1400 REM  DISPLAY POINTER
1405 D8$="         ":D3$=D8$+"     "
1410 IF I>10 THEN 1460
1420 K=3*I+3
1430 PRINT"  ";D3$;SPC(K);"**  ";
1440 GOSUB 1500:REM  PRODUCE TONE
1450 PRINT"   ":RETURN
1460 K=TB(I-10)
1470 PRINT"  ";D8$;SPC(K);"*  ";
1480 GOSUB 1500
1485 PRINT"  ":RETURN

READY.
```

guessed (or until the player gives up and asks for the answer). When the number is guessed, the computer displays the number of tries that it took to guess the number.

Figure 15.37 C64 organ when (a) playing a white key (F) and (b) a black key (T).



EXERCISE 15-3

Write a program that stores a song in DATA statements (see Exercise 14-7) and then plays the song when the program is run.

EXERCISE 15-4

Modify the program in Exercise 15-3 to store several songs that the user can select by number.

EXERCISE 15-5

Write a program that will remember each key pressed on the C64/VIC organ and the length of time that each key is pressed (see Exercise 14-4). Have the program replay the song when the zero key is pressed.

EXERCISE 15-6

Write a program to play the card game blackjack against the computer. The player first places a bet. Two cards are dealt to the player, and two cards are dealt to the computer (one face up and one face down). The player can ask for a hit (another card) as many times as he or she wants. The player's goal is to have a higher count than the computer without going over 21. Face cards count as 10, and an ace can count as either 1 or 11. Being dealt an ace and a face card is a *blackjack* and an automatic win. If the player's count goes over 21, it is a "bust" and the player loses. After the player stops taking hits (with the card count less than or equal to 21), the computer turns over its face-down card and can then take additional cards to try to beat the player. The computer will always take a hit if its card count is less than 17. The computer will always stand for a card count of 17 or greater. If the player wins, the amount of the bet is added to his or her winnings. If the computer wins, the amount of the bet is subtracted from the player's winnings. No money is won or lost on a tie. Have the program continue playing and keep a running total of the player's winnings.

APPENDICES

APPENDIX A—RESERVED WORDS

The following reserved words cannot be used as variable names or parts of variable names in a BASIC program:

ABS	GET	OPEN	SPC
AND	GET#	OR	SQR
ASC	GOSUB	PEEK	ST
ATN	GOTO	POKE	STEP
CHR$	IF	POS	STOP
CLOSE	INPUT	PRINT	STR$
CLR	INT	PRINT#	SYS
CMD	LEFT$	READ	TAB
CONT	LEN	READ#	TAN
COS	LET	REM	THEN
DATA	LIST	RESTORE	TI
DEF	LOAD	RETURN	TI$
DIM	LOG	RIGHT$	TO
END	MID$	RND	USR
EXP	NEW	RUN	VAL
FN	NEXT	SAVE	VERIFY
FOR	NOT	SGN	WAIT
FRE	ON	SIN	

APPENDIX B—ASCII CODES

Table B.1 shows the ASCII codes for all letters, digits, punctuation marks, and the standard graphic symbols. Table B.2 shows the ASCII codes for other special keys.

APPENDIX C—PEEK/POKE CODES

Table C.1 shows the PEEK/POKE screen codes for the standard uppercase/graphics character set. Table C.2 gives the PEEK/POKE screen codes for the alternate lowercase/uppercase character set.

APPENDIX D—HEXADECIMAL NUMBERS

Although not needed when programming in BASIC, hexadecimal numbers are useful when dealing with data as it is actually stored in the computer. The following is a brief introduction.

Consider a box containing one marble. When the marble is in the box, we will say that the box is *full* and associate the digit 1 with the box. If we take the marble out of the box, the box will be empty, and we will then associate the digit 0 with the box. The two binary digits 0 and 1 are called *bits,* and with one bit we can count from zero (box empty) to one (box full) as shown in Figure D.1.

TABLE B.1 ASCII codes for letters, digits, punctuation marks, and graphic symbols.

32		48	0	64	@	80	P	96	─	112	┐
33	!	49	1	65	A	81	Q	97	♠	113	●
34	"	50	2	66	B	82	R	98	│	114	─
35	#	51	3	67	C	83	S	99	─	115	●
36	$	52	4	68	D	84	T	100	─	116	│
37	%	53	5	69	E	85	U	101	─	117	╱
38	&	54	6	70	F	86	V	102	─	118	╳
39	'	55	7	71	G	87	W	103	│	119	○
40	(	56	8	72	H	88	X	104	│	120	♣
41	)	57	9	73	I	89	Y	105	╲	121	│
42	*	58	:	74	J	90	Z	106	╲	122	♦
43	+	59	;	75	K	91	[	107	╱	123	+
44	,	60	<	76	L	92	£	108	└	124	▒
45	-	61	=	77	M	93	]	109	╲	125	│
46	.	62	>	78	N	94	↑	110	╱	126	π
47	/	63	?	79	O	95	←	111	┌	127	◥
160		176	┌								
161	▌	177	┴								
162	▄	178	┬								
163	▔	179	┤								
164	▁	180	├								
165	▏	181	▌								
166	▒	182	▐								
167	▕	183	▔								
168	𝔸	184	▁								
169	◤	185	▄								
170	▕	186	└								
171	┣	187	▪								
172	▪	188	▝								
173	▙	189	┘								
174	┐	190	▝								
175	▁	191	▚								

TABLE B.2 ASCII codes for special keys.

5	WHITE	31	BLUE	142	UPPERCASE
13	RETURN	133	F1	144	BLACK
14	LOWERCASE	134	F3	145	CRSR UP
17	CRSR DOWN	135	F5	146	RVS OFF
18	RVS ON	136	F7	147	CLR
19	HOME	137	F2	148	INST
20	DEL	138	F4	156	PURPLE
28	RED	139	F6	157	CRSR LEFT
29	CRSR RIGHT	140	F8	158	YELLOW
30	GREEN	141	SHIFT RETURN	159	CYAN

TABLE C.1 PEEK/POKE codes for standard uppercase character set.

0	@	16	P	32		48	0	64	─	80	┐
1	A	17	Q	33	!	49	1	65	♠	81	●
2	B	18	R	34		50	2	66	│	82	─
3	C	19	S	35	#	51	3	67	─	83	●
4	D	20	T	36	$	52	4	68	─	84	│
5	E	21	U	37	%	53	5	69	─	85	╱
6	F	22	V	38	&	54	6	70	─	86	╳
7	G	23	W	39	'	55	7	71	│	87	○
8	H	24	X	40	(	56	8	72	│	88	♣
9	I	25	Y	41	)	57	9	73	╲	89	│
10	J	26	Z	42	*	58	:	74	╲	90	♦
11	K	27	[	43	+	59	;	75	╱	91	+
12	L	28	£	44	,	60	<	76	└	92	▒
13	M	29	]	45	-	61	=	77	╲	93	│
14	N	30	↑	46	.	62	>	78	╱	94	π
15	O	31	←	47	/	63	?	79	┌	95	◥
96		112	┌								
97	▌	113	┴								
98	▄	114	┬								
99	▔	115	┤								
100	▁	116	├								
101	▏	117	▌								
102	▒	118	▐								
103	▕	119	▔								
104	𝔸	120	▁								
105	◤	121	▄								
106	▕	122	└								
107	┣	123	▪								
108	▪	124	▝								
109	▙	125	┘								
110	┐	126	▝								
111	▁	127	▚								

TABLE C.2 PEEK/POKE codes for alternate lower case character set.

0	@	16	P	32		48	0	64	─	80	P
1	a	17	Q	33	!	49	1	65	A	81	Q
2	b	18	R	34		50	2	66	B	82	R
3	c	19	S	35	#	51	3	67	C	83	S
4	d	20	T	36	$	52	4	68	D	84	T
5	e	21	U	37	%	53	5	69	E	85	U
6	f	22	V	38	&	54	6	70	F	86	V
7	g	23	W	39	'	55	7	71	G	87	W
8	h	24	X	40	(	56	8	72	H	88	X
9	i	25	Y	41	)	57	9	73	I	89	Y
10	j	26	Z	42	*	58	:	74	J	90	Z
11	k	27	[	43	+	59	;	75	K	91	+
12	l	28	£	44	,	60	<	76	L	92	▒
13	m	29	]	45	-	61	=	77	M	93	│
14	n	30	↑	46	.	62	>	78	N	94	π
15	o	31	←	47	/	63	?	79	O	95	◥
96		112	┌								
97	▌	113	┴								
98	▄	114	┬								
99	▔	115	┤								
100	▁	116	├								
101	▏	117	▌								
102	▒	118	▐								
103	▕	119	▔								
104	𝔸	120	▁								
105	◤	121	▄								
106	▕	122	▏								
107	┣	123	▪								
108	▪	124	▝								
109	▙	125	┘								
110	┐	126	▝								
111	▁	127	▚								

Consider now a second box that can also be full (1) or empty (0). However, when this box is full it will contain *two* marbles as shown in Figure D.2. With these two boxes (two bits) we can now count from zero to three as shown in Figure D.3. Note that the value of each two-bit binary number shown in Figure D.3 is equal to the total number of marbles in the two boxes.

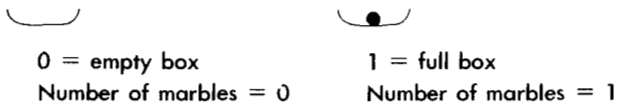


FIGURE D.1 You can count from 0 to 1 with one bit.

FIGURE D.2 This box can contain either two marbles (full) or no marbles.

We can add a third bit to the binary number by adding a third box that is full (bit=1) when it contains four marbles and is empty (bit=0) when it contains no marbles. It must be either full (bit=1) or empty (bit=0). With this third box (three bits) we can count from zero to seven as shown in Figure D.4.

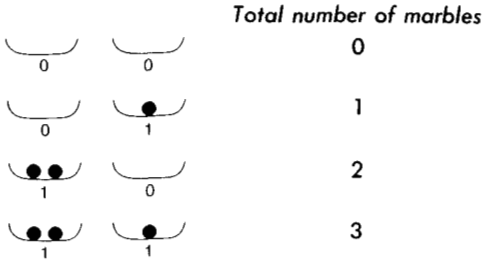
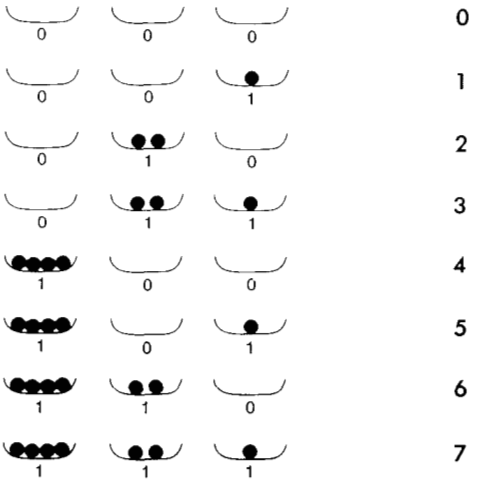


FIGURE D.3 You can count from 0 to 3 with two bits.

FIGURE D.4 You can count from 0 to 7 with three bits.



If you want to count beyond seven, you must add another box. This box must contain eight marbles. The binary number whose value is eight would then be written as 1000. Remember that a 1 in a binary number means that the corresponding box is full of marbles. The number of marbles that constitutes a full box varies as 1, 2, 4, 8 starting at the right. This means that with four bits we can count from 0 to 15 as shown in Figure D.5.

FIGURE D.5 You can count from 0 to 15 with four bits.

No. of Marbles in each full box (bit = 1)				*Total no. of marbles*	*Hex Digit*
8	4	2	1		
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	10	A
1	0	1	1	11	B
1	1	0	0	12	C
1	1	0	1	13	D
1	1	1	0	14	E
1	1	1	1	15	F

It is convenient to represent the total number of marbles in each set of boxes (that is, each four-bit binary number) shown in Figure D.5 by a single character, called a *hexadecimal digit*. The sixteen hexadecimal digits are shown in the right-hand column of Figure D.5. The hexadecimal digits 0-9 are the same as the decimal digits 0-9. The hexadecimal digits A-F are used to represent the decimal numbers 10-15. Thus, for example, the hexadecimal digit D is equivalent to the decimal number 13.

In order to count beyond 15 in binary you must add more boxes. Each full box you add must contain twice as many marbles as the previous full box. With eight bits you can count from 0 to 255. A few examples are shown in Figure D.6. Given a binary number, the corresponding decimal number is equal to the total number of marbles in all of the boxes. To find this number, just add up all the marbles in the full boxes (the ones with binary digits = 1).

Binary numbers become more cumbersome to work with as their length increases. We then use the corresponding *hexadecimal number* as a shorthand method of representing the binary number. This is very easy to do. You just divide the binary number into groups of four bits starting at the right and then

FIGURE D.6 You can count from 0 to 255 with eight bits.

No. of marbles in each full box (bit = 1)								Total no. of marbles
128	64	32	16	8	4	2	1	
0	0	1	1	0	1	0	0	52
1	0	1	0	0	0	1	1	163
1	1	1	1	1	1	1	1	255

represent each four-bit group by its corresponding hexadecimal digit. For example, the binary number 10011010 is equivalent to the hexadecimal number 9A(1001=9 and 1010=A). You should verify that the total number of marbles represented by this binary number is 154.

Instead of counting the marbles in the "binary boxes" you can count the marbles in "hexadecimal" boxes. The first "hexadecimal" box can contain any number of marbles from 0 to 15 (0-F hex). The second "hexadecimal" box must contain multiples of 16 marbles (16, 32, 64, . . .). For the hexadecimal number 9A, the first box contains A*1=10 marbles and the second box contains 9*16=144 marbles. Therefore, the total number of marbles is equal to 144+10=154.

A third hexadecimal box would contain a multiple of 16^2=256 marbles, and a fourth hexadecimal box would contain a multiple of 16^3=4,096 marbles. As an example the 16-bit binary number

1000011111001001
8 7 C 9

is equivalent to the decimal number 34,761 (that is, it represents 34,761 marbles). This can be seen by expanding the hexadecimal number as follows:

$$\begin{aligned} 8 \times 16^3 &= 8 \times 4{,}096 = 32{,}768 \\ 7 \times 16^2 &= 7 \times 256 = 1{,}792 \\ C \times 16^1 &= 12 \times 16 = 192 \\ 9 \times 16^0 &= 9 \times 1 = 9 \\ & \qquad\qquad 34{,}761 \end{aligned}$$

You can see that by working with hexadecimal numbers you can reduce by a factor of four the number of digits that you have to work with.

The following table will allow you to convert hexadecimal numbers of up to four digits to their decimal equivalents. Note, for example, how the four terms in the conversion of 87C9 given above can be read directly from the table.

TABLE D.1 Hexadecimal and Decimal Conversion

15 BYTE 8				7 BYTE 0			
15 CHAR 12		11 CHAR 8		7 CHAR 4		3 CHAR 0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15

APPENDIX E—SUMMARY OF BASIC STATEMENTS

The following summary gives examples of using various statements in CBM BASIC. For a more detailed discussion of each statement, refer to the sections beginning on the pages cited.

APPENDIX F—USING MACHINE LANGUAGE SUBROUTINES WITH BASIC

This appendix assumes that you know how to write 6502 assembly language programs. If you have a machine language subroutine, there are two ways that you can access this subroutine from a BASIC program. The first is to use the SYS command, and the second is to use the USR function.

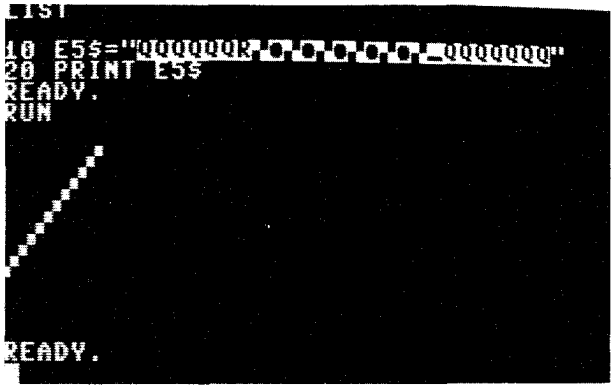
The BASIC command SYS(addr) will transfer control to a machine language subroutine starting at the decimal address *addr*. This command can be used in either the immediate mode or the deferred mode. When the machine language subroutine executes an RTS instruction, control will return to the calling BASIC program.

In BASIC, data values can be passed to and from a machine language subroutine by using the USR function. When the BASIC statement X=USR(A) is executed, the value of A is placed in the floating point accumulator in the hex locations $61-$66 (97-102 decimal), and control is then transferred to a machine language subroutine whose starting address has been placed in locations 0001 (LSB) and 0002 (MSB) on the VIC 20 and locations 785 (LSB) and 786 (MSB) on the Commodore 64.

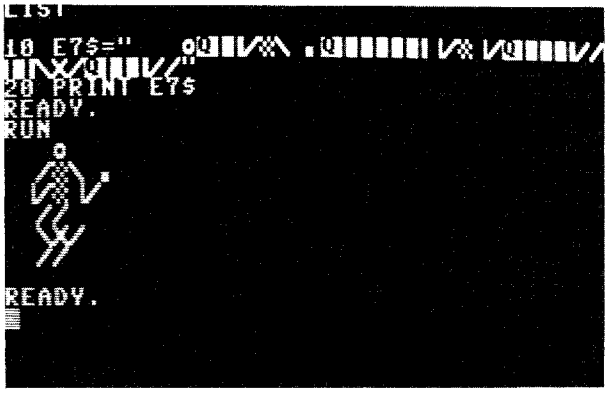
When your machine language subroutine executes an RTS instruction, the floating point number currently stored in the floating point accumulator will be assigned to the value of the function USR. That is, it will become the value of X in the statement X = USR(A).

APPENDIX G—ANSWERS TO SELECTED EXERCISES

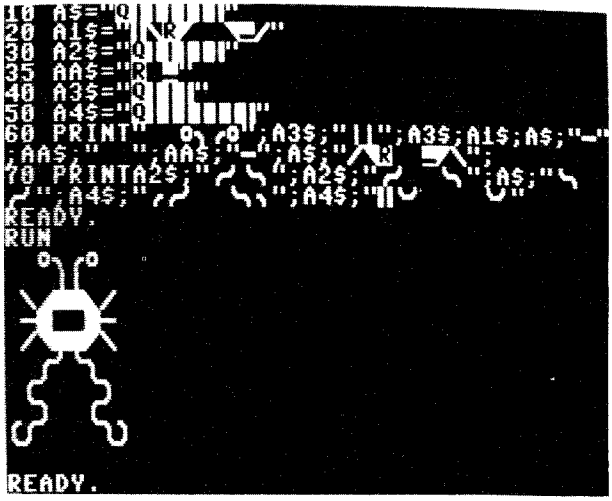
EXERCISE 2-3

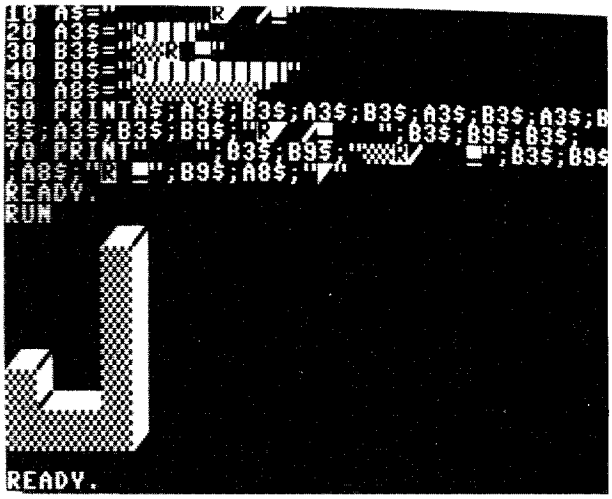


EXERCISE 2–3 contd.



EXERCISE 4-1





EXERCISE 4-1

```
10 B3$="
20 B9$="
30 A1$="
40 A2$="
50 A3$="
60 A4$=A2$+"
70 A5$=A2$+"
80 PRINTA1$;B9$;A4$;B9$;A5$;B9$;A3$;B3$;"
90 PRINTB9$;"
READY.
```

```
10 B3$="
15 B6$="
20 B9$="
25 B7$="
30 A1$="
40 A2$="
50 A3$="
55 A4$="
60 PRINT A1$;B9$;A2$;"
70 PRINT B7$;A4$;"
READY.
```

EXERCISE 4-2

```
10 B$="■■■■■■■■■"
20 A$="■5       "
30 A1$="◆◆    ◆ "
40 A2$="        "
50 A3$="   ◆    "
60 A4$=" ◆   ◆◆"
70 A5$="      5"
80 PRINTA$;B$;A1$;B$;A2$;B$;A2$;B$;A3$;B$;A2$;B$;A2$;B$;A4$;B$;A5$;

READY.

10 B7$="■■■■■■■■■"
20 A$="■J       "
30 A1$="  ▀▄ ▄▀  "
40 A2$=" |  ||   "
50 A3$=" | ■    ■ | "
60 A4$=" | ■    ■ | "
70 A5$=" | ■    ■ | "
80 A6$="   ||  | "
90 A7$="  ▄▀ ▀■  "
100 A8$="      J"
110 PRINTA$;B7$;A1$;B7$;A2$;B7$;A3$;B7$;A4$;B7$;A5$;B7$;A6$;B7$;A7$;B7$;A8$;
120 S1$="▐■▙"
130 S2$="▐▌▌▌▌▌▌■■■■■■■■▙"
140 PRINTS1$;S2$;"■■■■■■■■■"

READY.

10 B7$="■■■■■■■■■"
20 A$="■0       "
30 A1$="  ▄▀▐   "
40 A2$="  ▄▀■| |   "
50 A3$="  ■    ■▖  "
60 A4$="  ■      ■ "
70 A5$="  ▀■     ■ "
80 A6$="   | |▄ "
90 A7$="   |_▄▀■  "
100 A8$="      Q"
110 PRINTA$;B7$;A1$;B7$;A2$;B7$;A3$;B7$;A4$;B7$;A5$;B7$;A6$;B7$;A7$;B7$;A8$;
120 S1$="▐■▙"
130 S2$="▐▌▌▌▌▌■■■■■■■■▙"
140 PRINTS1$;S2$;"■■■■■■■■■"

READY.

10 B7$="■■■■■■■■■"
20 A$="■K       "
30 A1$="  ▀▄  ▄▀  "
40 A2$="   | ■ ■  "
50 A3$="  ▄▀    ▀■ "
60 A4$="  ■      ■ "
70 A5$="  ▙▖    ▄▀ "
80 A6$="   ■ ■ |   "
90 A7$="  ▄▀  ▀■  "
100 A8$="      K"
110 PRINTA$;B7$;A1$;B7$;A2$;B7$;A3$;B7$;A4$;B7$;A5$;B7$;A6$;B7$;A7$;B7$;A8$;
120 S1$="▐■▙"
130 S2$="▐▌▌▌▌▌■■■■■■■■▙"
140 PRINTS1$;S2$;"■■■■■■■■■"

READY.
```

EXERCISE 4-3

a. 1.09544512 **b.** −0.05 **c.** 0.416666667 **d.** 1.29099445 **e.** 3.62686041

INDEX

PERSONAL COMPUTING

This easy-to-use book provides beginning and advanced programmers with a hands-on step-by-step approach to programming a Commodore 64™ or VIC 20™ in BASIC.

Complete with actual photographs taken from the computers' video screens, **Commodore 64/VIC 20 BASIC** uses graphical examples to develop many fundamental programming concepts. This unique teaching method gives you the opportunity to actually see a direct visual picture of what the program is doing. What's more, this comprehensive introduction features numerous examples to help you write your own programs for specific applications.

Ideal for self-study or classroom instruction, **Commodore 64/VIC 20 BASIC** includes valuable information on:

- drawing simple graphic figures like those shown here
- string variables and string functions
- loops and arrays
- writing a Hangman word game
- developing a Commodore 64/VIC 20 organ that plays three octaves of musical notes
- and much more.

Richard Haskell holds a Ph.D. from Rensselaer Polytechnic Institute and is a professor of engineering and chairman of electrical and computer engineering at Oakland University in Michigan. He has written five other Prentice-Hall books entitled *Apple II–6502 Assembly Language Tutor* (1983), *TRS-80 Extended Color BASIC* (1983), *Atari BASIC* (1983), *Apple BASIC* (1982), and *PET/CBM BASIC* (1982).

Thomas Windeknecht holds a Ph.D. from Case Western Reserve University and is a computer science professor at Oakland University in Michigan.

Cover design by Hal Siegel

PRENTICE-HALL, Inc.
Englewood Cliffs, New Jersey 07632

P

ISBN 0-13-152281-7

0 21898 15228 2