RICHARD HASKELL

atari basic

$13.95 A SPECTRUM BOOK

Richard Haskell holds a Ph.D. from Rensselaer Polytechnic Institute and is an engineering professor at Oakland University in Michigan. In addition, he has designed numerous microprocessor-based systems for industrial application and written four other books in the Prentice-Hall computer series entitled *PET/CBM BASIC, APPLE BASIC, APPLE BASIC: 6502 Assembly Language Tutor,* and *TRS-80 Extended Color BASIC.*

ATARI BASIC

RICHARD HASKELL

PRENTICE-HALL, INC.
Englewood Cliffs, NJ 07362

Library of Congress Cataloging in Publication Data

Haskell, Richard E.
Atari basic.

"A Spectrum Book."
Includes index.
1. Atari (Computer)—Programming. 2. Basic (Computer program language) I. Title

QA76.8.A82H37 1983 001.64'24 82-25070
ISBN 0-13-049809-2
ISBN 0-13-049791-6 (pbk.)

This book is available at a special discount when ordered in bulk quantities. Contact Prentice-Hall, Inc., General Publishing Division, Special Sales, Englewood Cliffs, N.J. 07632.

10 9 8 7 6 5 4

Editorial/production supervision by Cyndy Lyle Rymer
Interior design by Frank Moorman
Page layout by Fred Dahl
Manufacturing buyer Cathie Lenard

ISBN 0-13-049791-6 {PBK.}

ISBN 0-13-049809-2

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall of Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*
Editora Prentice-Hall do Brasil Ltda., *Rio de Janeiro*

CONTENTS

PREFACE

Anyone planning to teach a BASIC programming course using ATARI computers is faced with the problem of selecting an appropriate text. Programming manuals provided by the manufacturer are generally more suitable for reference than for teaching. Standard textbooks on BASIC programming will describe a BASIC language enough at variance with ATARI BASIC to lead to considerable frustration on the part of the student. Such texts will be of no help in learning to use the graphics capability of the ATARI computer. Since many interesting programs on the ATARI will involve graphics, these texts will be of little value.

This book is designed to be used as a text for learning to program in BASIC using the ATARI computer. It should be suitable for introductory programming courses at the high school, junior college, or university levels. It can also be used for self-study with an ATARI computer.

Three companion texts entitled *PET/CAM BASIC, APPLE BASIC,* and *TRS-80 EXTENDED COLOR BASIC* are also available for use with these computers.

The strategy of this book is "learning by doing." Step by step the student is led through all aspects of BASIC programming on an ATARI computer. All examples are illustrated with many photographs taken from the computer's TV 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 for anyone wishing to write programs in a specific applications area.

Chapter 1 introduces the ATARI keyboard and the idea of string variables. Chapter 2 talks about the general nature of BASIC programs and covers the operation of the cassette tape recorder and floppy disk drive.

Chapter 3 covers numerical variables, arithmetic expressions, and the ATARI's built-in functions. The INPUT statement is covered in Chapter 4 with examples that include making sounds. Chapter 5 introduces the FOR . . . NEXT loop with programs to produce several sound effects. Chapter 6 introduces the IF . . . THEN statement and relational and logical operators.

The use of low-resolution graphics is introduced in Chapter 7, where the American flag is displayed on the screen. A more complete discussion of loops is given in Chapter 8. The use of subroutines is covered in Chapter 9, where the student learns to draw multiple figures of varying size and to use the game paddles and joysticks.

The READ . . . DATA statement is introduced in Chapter 10, which covers the method of drawing bar graphs. The topic of arrays is discussed in Chapter 11.

Strings and string functions are described in detail in Chapter 12 with examples given for dealing a hand of cards. High-resolution graphics is covered in Chapter 13, including examples showing how to plot circles and polygons. Chapter 14 describes the use of the PEEK and POKE statements, including how to use graphics and text modes 1 and 2, how to read the keyboard, how to define your own character set, and how to write text on the high-resolution graphics screen. Chapter 15 describes the development of two complete programs: the HANGMAN word game and an ATARI organ that plays four octaves of music from the ATARI keyboard. This chapter also includes examples of how to store data on a floppy disk.

Students who complete this text will have a solid foundation in fundamental programming techniques and will have acquired the particular skill of being able to program the ATARI computer using BASIC.

It is a pleasure to acknowledge my students of many years who had to learn to program on a large computer with none of the graphics capability of the ATARI computer, but on whom many of the ideas in this book were first tested. Invaluable help in using the ATARI computer was received from Anne Jaworski and Laura Snider-Feldmesser. Special thanks go to Sharon Rix, who typed the manuscript with skill, patience, and good humor.

ATARI BASIC

1

LEARNING TO USE THE ATARI KEYBOARD

There is only one way to learn how to program a computer. You must write programs and run them on a 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 by actually using an ATARI computer.

The ATARI computer is one of several popular personal computers (such as the PET, the Apple II, and the TRS-80) 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, this language is implemented somewhat differently on each of these various computers. This is particularly true with respect to how graphics programming is done. This means that a BASIC program written for an Apple II computer will not, in general, run on an ATARI without some modification. It also means that 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 is written with the assumption that you have an ATARI 800 or ATARI 400 computer available for you to use.

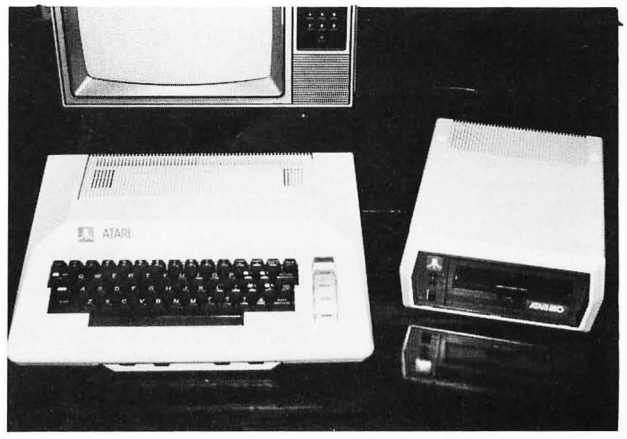
The programs shown in this book were all written on an ATARI 800 computer, shown in Figure 1.1. However, most of the programs will run on an ATARI 400.

In this chapter you will become familiar with the use of the ATARI keyboard. In particular you will learn the meaning of the special keys shown in Figure 1.2. In the process you will learn how to draw simple graphic figures.

You will also learn

1. to use the PRINT statement

FIGURE 1.1 The ATARI 800 computer.



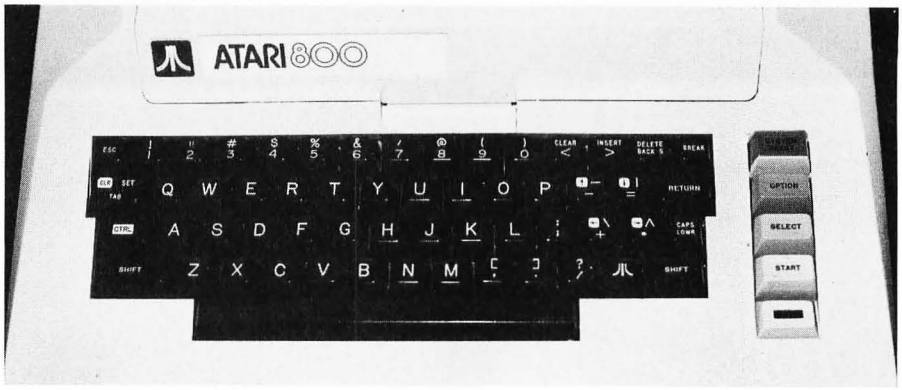


FIGURE 1.2 Special keys discussed in this chapter.

2. what strings and string variables are
3. the difference between the immediate and deferred modes of execution
4. to use the LIST and RUN commands
5. to edit a statement.

THE ATARI KEYBOARD

Begin by turning on your ATARI computer. This is done with the switch on the right side of the computer, shown in Figure 1.3. If you're using a disk drive, turn the disk drive on first, insert a master diskette, and then turn on the computer. The TV screen should have the display shown in Figure 1.4.

FIGURE 1.3 Turning on the ATARI computer.

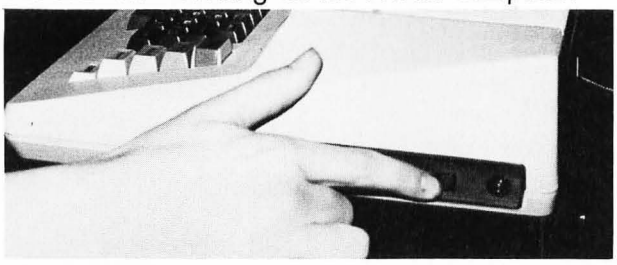
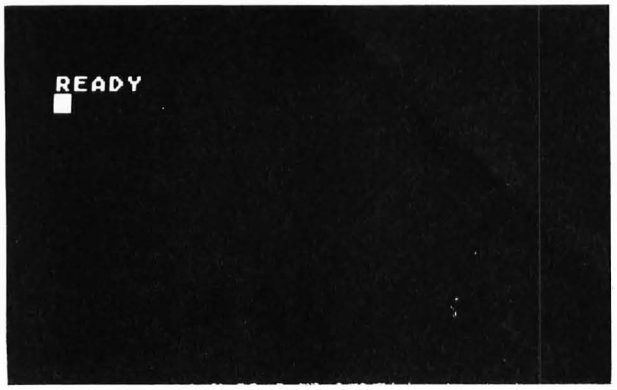


FIGURE 1.4 Initial screen display using an ATARI computer.



Your ATARI computer contains a *read only memory* cartridge, called a ROM, that contains the BASIC interpreter. This cartridge can be seen by opening the lid above the keyboard, as shown in Figure 1.5. Another ROM cartridge is located under the ATARI's main cover. This cartridge contains a collection of systems programs called an *operating system*. The location of this ROM cartridge is shown in Figure 1.6.

FIGURE 1.5 ROM cartridge containing the BASIC interpreter.

FIGURE 1.6 Cartridges containing read/write memory (RAM) and the ATARI's operating system (ROM).

In addition to the read only memory (ROM), your ATARI computer also contains some *read/write memory,* called RAM for *random-access memory.* These RAM cartridges are also shown in Figure 1.6. 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 can't be changed. Also, when you turn the power to your ATARI computer off, 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 always there every time you turn your computer on. On the other hand, every program you write is stored in RAM and is lost whenever you turn the power off. This is why you must save your programs on a cassette tape or diskette if you wish to run them at a later time without having to type in the entire program again.

The amount of RAM you can have in your ATARI depends on the number and size of the RAM cartridges you have. Each cartridge may be either an 8K or 16K RAM cartridge. The more RAM you have, the larger the programs you can run and the more data you can store in the computer. If you have 16K of RAM, your ATARI contains 16,384 bytes of RAM (1K = 1,024 bytes). A byte is 8 bits, where a bit is a 1 or a 0. Thus, for example, 10101101 is a byte. It takes 1 byte to store a character in the ATARI. If your ATARI contains 32K bytes of RAM, you really have 32,768 bytes of RAM. Three 16K cartridges would give your ATARI 48K or 49,152 bytes of RAM.

When the TV screen contains the word READY followed (on the next line) by a square cursor, the computer is ready and waiting for you to type in something. Try typing your name and then press the RETURN key. If your name is JOHN you should see something like the display shown in Figure 1.7.

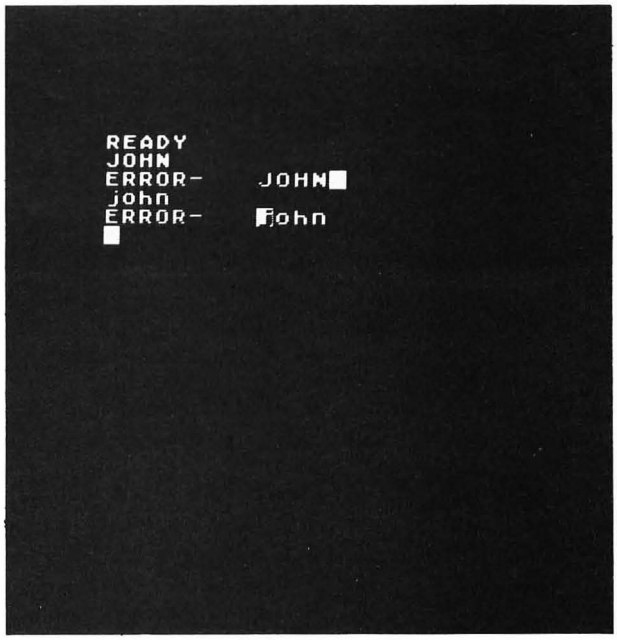


FIGURE 1.7 An ERROR occurs when you type an invalid BASIC command.

Note that the message

ERROR- JOHN

appeared on the screen when you pressed the RETURN key. This is because JOHN is not a valid BASIC command and the computer can only respond to BASIC commands that it understands. You will learn all of these valid BASIC commands in this book. If you still type an invalid command because of misspelling, for example, the ATARI will respond with an ERROR message. You cannot hurt the computer by pressing the wrong key. If it doesn't like what you typed, it will let you know.

When you turn on the ATARI it will print letters in upper case (CAPS). All BASIC statements must be typed in all caps. If you want to type lower-case letters, press the CAPS/LOWR key located on the right side of the keyboard (see Figure 1.2). Then type JOHN again as shown in Figure 1.7. Note that you still get an ERROR, but this time the ATARI does not even recognize the first letter because it is lower case. To return to all caps, press the CAPS/LOWR key while holding down the SHIFT key.

Graphic Keys

All letter keys (and some punctuation keys) have graphic symbols associated with them. These are shown in Figure 1.8 as they are positioned on the keyboard.

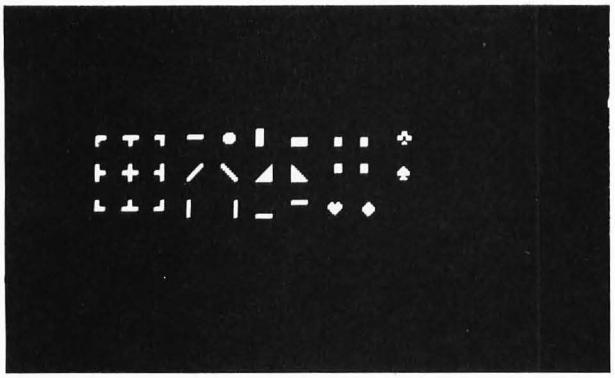
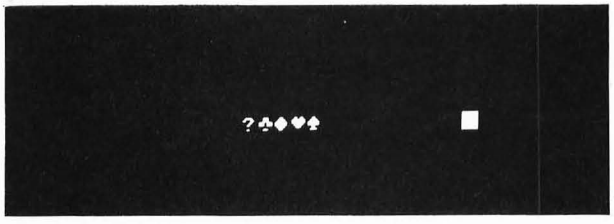


FIGURE 1.8 Graphic symbols can be printed by holding down the CTRL key.

These symbols can be typed on the screen by holding down the CTRL key while pressing the particular graphic key. For example, if you press keys P . , ; while holding the CTRL key down you will display the spade, heart, diamond, and club used in playing cards, as shown in Figure 1.9.

FIGURE 1.9 Playing card graphic symbols.



Note that when you press RETURN after typing only graphic symbols you also get an ERROR. Try typing some of the graphic symbols to see what they look like. They are shown in Figure 1.10.

Cursor Keys

The four keys labeled *cursor keys* in Figure 1.2 contain four arrows on small white squares. If you press any of these keys while holding the CTRL key down, the cursor will move in the direction of the arrow. Try it.

Key	*Graphic Symbol*	*Key*	*Graphic Symbol*	*Key*	*Graphic Symbol*
comma		J		T	
A		K		U	
B		L		V	
C		M		W	
D		N		X	
E		O		Y	
F		P		Z	
G		Q		Period	
H		R		semicolon	
I		S			

FIGURE 1.10 Graphic symbols available on the keyboard.

If you continue to hold a cursor key down (while also holding the CTRL key down) the cursor will continually move across the screen. Try this. In fact, all keys on the keyboard will repeat if they are held down.

Note that if you move the cursor above the top of the screen, it will come in at the bottom. Similarly, if you move it below the bottom of the screen it will come in at the top. Moving the cursor past the right edge of the screen will cause it to come in at the left on the same line. Similarly, moving it off the left edge of the screen causes it to come in at the right edge on the same line. Try this.

CLEAR Key

The CLEAR key is located on the top row of keys. The word CLEAR appears over the < symbol (see Figure 1.2). If you press this key while holding down the SHIFT key, the screen will clear. Try this. The screen will also clear if you press the CLEAR key while holding down the CTRL key.

MAKING GRAPHIC FIGURES

The graphic keys and the cursor keys can be combined to form graphic figures. For example, clear the screen (by pressing SHIFT CLEAR); then move the cursor down near the center of the screen. Now type the following keys. (CTRL Q directs you to type the graphic symbol on key Q.)

CTRL Q
CTRL E
CTRL ↓
CTRL ←
CTRL ←
CTRL Z
CTRL C

You should have generated a square figure, as shown in Figure 1.11.

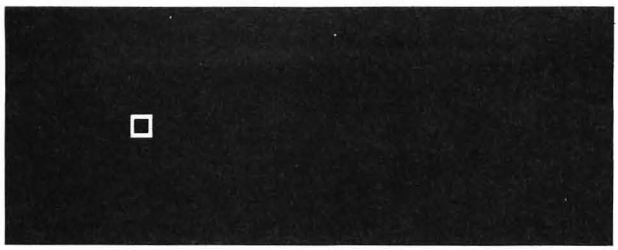


FIGURE 1.11 Square figure generated used keys, Q, E, Z, and C.

Similar shapes can be made using other graphic keys. For example, in Figure 1.12 the solid square is made using the graphics on keys I, O, K, and L. The diamond is made using the graphics on keys F and G. Try making these figures.

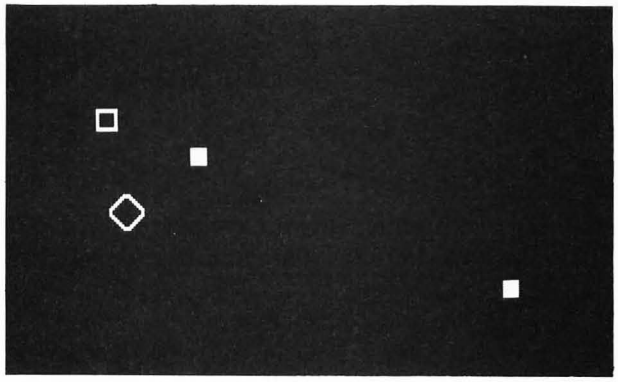


FIGURE 1.12 Hollow square generated using keys Q, E, Z, and C; solid square generated using keys I, O, K, and L; diamond generated using keys F and G.

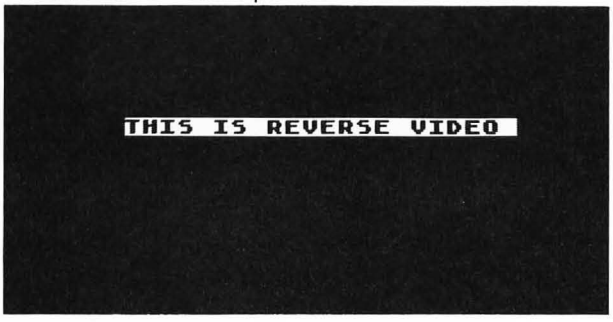
REVERSE VIDEO

The ATARI key shown in Figure 1.2 is used to turn the reverse video on and off. Clear the screen, press the ATARI key, and then type

THIS IS REVERSE VIDEO

The result should be as shown in Figure 1.13. To turn the reverse video off, press the ATARI key again.

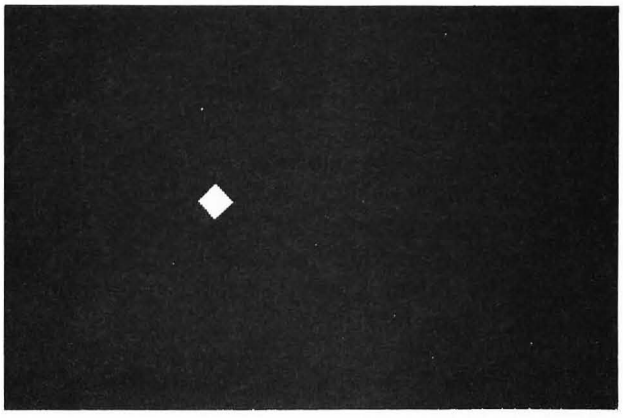
FIGURE 1.13 Example of reverse video.



The reverse video key can be useful when making certain graphic figures. For example, the graphic figure in Figure 1.14 can be made by typing the following keys:

CTRL H
CTRL J
CTRL ↓
CTRL ←
CTRL ←
REVERSE VIDEO ON (ATARI KEY)
CTRL J
CTRL H
REVERSE VIDEO OFF (ATARI KEY)

FIGURE 1.14 Graphic figure generated using keys H and J plus reverse video.

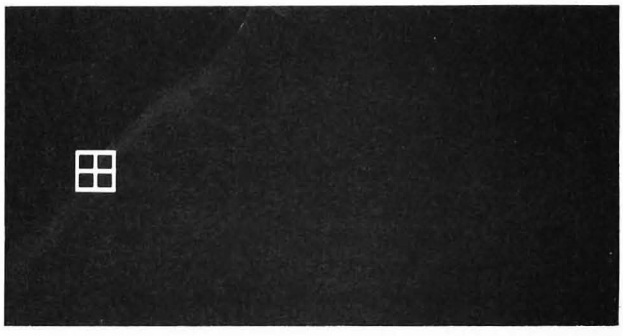


Try making this figure.

EXERCISE 1.1

Try to generate the graphic figure shown in Figure 1.15.

FIGURE 1.15 Graphic figure for Exercise 1.1.

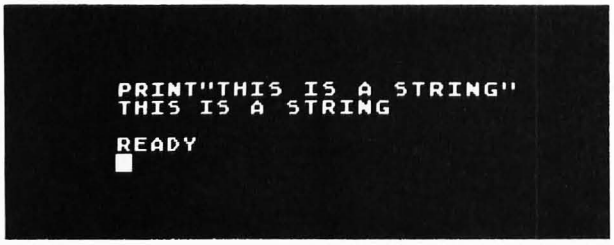


Clear the screen and type

PRINT "THIS IS A STRING"

followed by RETURN. The result should be as shown in Figure 1.16. 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 this string (without the quotation marks) on the screen. This is called the *immediate mode* of execution.

FIGURE 1.16 Using the PRINT statement in the immediate mode of execution.



When writing a BASIC statement, spaces are normally ignored by the computer. Thus, for example, you could have included a space following the word PRINT in Figure 1.16. However, when included as part of a string (that is, between quotation marks) spaces are not ignored.

A string may include the graphic characters. For example, try printing the playing card symbols as shown in Figure 1.17 (keys P . , ;;).

FIGURE 1.17 Printing graphic symbols using the PRINT statement.

A string may also include the *cursor* moves. This may seem a little strange at first, but as you will see, this is what allows you to prestore an entire graphic figure as a string. If you press the ESC key (see Figure 1.2) before you press one of the cursor keys in a string (that is, after you have typed one quotation mark) the cursor movement does not take place at that time. Instead, the cursor arrow is printed in the string. Later, when this string is printed, the cursor movements will occur in the order in which they appear in the string.

For example, suppose that you want to print the square graphic figure shown in Figure 1.11. Just include the key strokes

CTRL Q
CTRL E
ESC CTRL ↓
ESC CTRL ←
ESC CTRL ←
CTRL Z
CTRL C

as a string in a PRINT statement. The result will look like Figure 1.18. Try it. Note that when the PRINT statement is executed, it is just as if you typed all the keys, including the cursor moves, very rapidly.

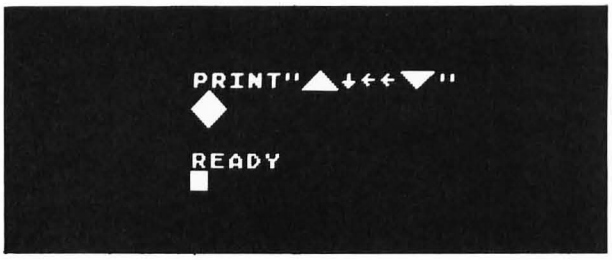
FIGURE 1.18 PRINT statement using cursor moves.

The arrows that appear in a string when you press the ESC key before typing the cursor key are simply used to tell the ATARI what to do with the cursor when the PRINT statement is executed.

EXERCISE 1.2

Use the PRINT statement to generate the graphic figure shown in Figure 1.14. Your result should look like Figure 1.19.

FIGURE 1.19 Answer to Exercise 1.2.



DEFERRED MODE OF EXECUTION

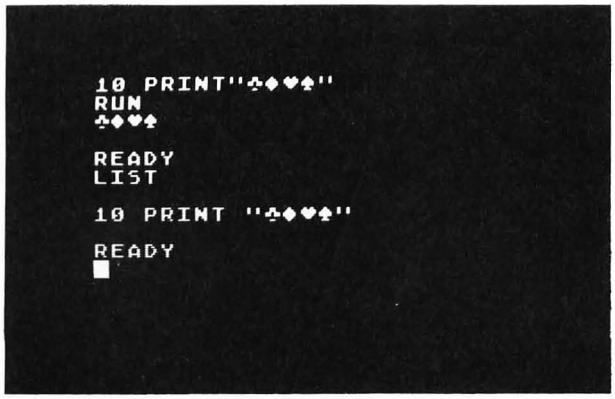
If a BASIC statement such as PRINT is preceded by a line number (such as 10), the statement is *not* executed immediately but rather its execution is deferred until the command RUN is typed. For example, Figure 1.20 shows how to print the playing card symbols using the deferred modes of execution. When BASIC statements have line numbers, these statements 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.*

FIGURE 1.20 PRINT statement using the deferred mode of execution.

Note that you must press RETURN at the end of each statement (such as PRINT) or command (such as RUN). The ATARI does not look at what you have typed on a line until you press RETURN. The ATARI then deciphers what you typed on the line and decides what to do.

You can always look to see what BASIC statements you currently have stored in the ATARI by typing LIST. Try it. You should have listed the single PRINT statement number 10 shown in Figure 1.21.

FIGURE 1.21 The LIST command will list all BASIC statements stored in memory.

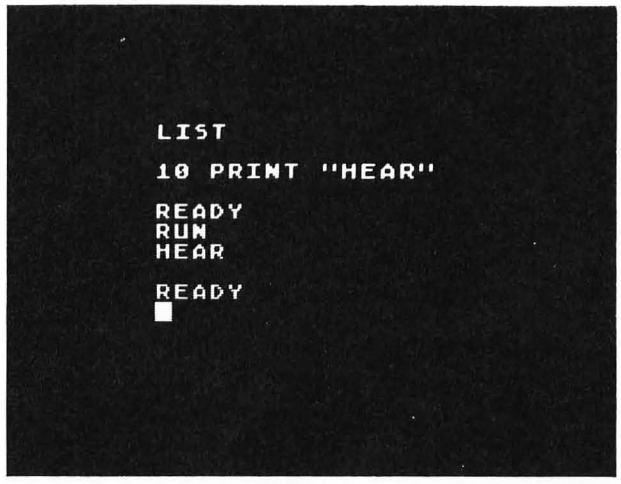


*Type NEW before typing the statement in Figure 1.20. See Chapter 2 for a discussion of the command NEW.

You can now edit this PRINT statement by using the cursor keys. Suppose that you want to print the word HEAR instead of the playing card symbols. Using the cursor keys, move the cursor over the club in the PRINT statement. Then type HEAR followed by RETURN. Now move the cursor down below READY and type RUN. The result should be as shown in Figure 1.22.

FIGURE 1.22 (a) Editing a PRINT statement; (b) running edited program.

(a)



(b)

The question mark (?) can be used as an abbreviation for the word PRINT. Try typing

?"THIS WILL STILL PRINT"

In the deferred mode, if you type

10 ? "HELLO"
RUN

the word HELLO will be printed. Note that if you now type LIST, a space will be inserted after the question mark (see Figure 1.23).

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

READY
10?"HELLO"
RUN
HELLO

READY
LIST

10 ? "HELLO"

READY
```

FIGURE 1.23 The question mark (?) can be used as a substitute for the word PRINT.

You can clear the screen in a BASIC program by including the keystrokes ESC SHIFT CLEAR in a string in a PRINT statement. For example, try the statement

10 ?"ESC SHIFT CLEAR"

Note that a special arrow symbol that points up and to the left is displayed between the double quotes. When you execute this statement by typing RUN the screen will clear. Try it.

EDITING USING THE BACKSPACE, DELETE, AND INSERT KEYS

The backspace key contains the words DELETE/BACK S; it is located above the RETURN key. When you press the backspace key the cursor moves one space to the left and erases any character that may have been located at that position on the screen. For example, type

PRINT "ABCSEF"

but *do not* press the RETURN key. Now press the backspace key four times so that SEF" is erased. Now retype DEF" and then press RETURN. The letters ABCDEF should be printed on the screen. If you press the DELETE/BACK S key while holding down the SHIFT key, the entire current line will be erased.

Suppose that you have already pressed the RETURN key before you notice a mistake. There are a couple of things you can do. You can just type the entire line over again. Any time you type a BASIC statement beginning with a particular line number, this new statement will replace any previous statement having the same line number. This is also an easy way to erase an entire line in a BASIC program. For example, if you type the number 50 and then press the RETURN key immediately, line 50 will be completely erased from the program.

You can also edit a line by using the INSERT and DELETE keys while holding down the CTRL key. For example, suppose that you want to change the word HEAR in the PRINT statement in Figure 1.22 to HEARING. First list the statement by typing LIST. Then move the cursor over the last quotation mark. While holding the CTRL key down press the INSERT key three times. This will move the quotation mark over three places. Now you can type ING followed by RETURN. If you now move the cursor down and type RUN, the computer should print the word HEARING, as shown in Figure 1.24.

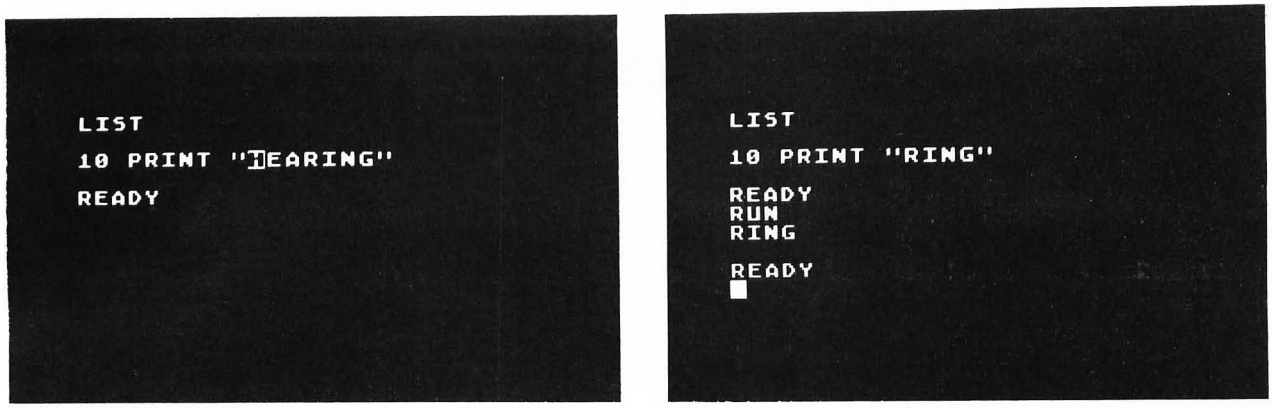
FIGURE 1.24 Inserting ING into "HEAR."

```
LIST
10 PRINT "HEAR  "
READY
```

(a)

```
LIST
10 PRINT "HEARING"
RUN
HEARING

READY
```

(b)



(a) (b)

FIGURE 1.25 Deleting HEA from "HEARING."

Suppose that you now want to change HEARING to RING. Type LIST (this isn't necessary but it minimizes the distance you have to move the cursor) and then move the cursor over the letter H in HEARING. Hold down the CTRL key and press the DELETE key three times. You should have deleted the letters HEA. Note that pressing the DELETE key while holding down the CTRL key will delete the character at the location 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 1.25.

The INSERT and DELETE keys can be used to edit any BASIC statement.

STRING VARIABLES

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

"HELLO"
"****"
"THIS IS A STRING"

Any character or graphic symbol can be included in a string. To the ATARI a blank space is just another character when it is included in a string.

A string can be given a special name and then can be referred to by this name. These string names are sometimes called *string variables*. The name of a string must start with a letter and end with a dollar sign ($); it can contain from 1 to 120 alphanumeric characters. Thus, the following are valid string names:

A$
B3$
AXE$
HOUSE$
FIVE$

The ATARI has a number of *reserved* words that it is constantly looking for. For example, RUN and LIST are reserved words. A complete list of reserved words is given in Appendix A. If a variable name starts with one of these reserved words, your program may not run properly. Thus, it is probably a good idea to keep your names short. Shorter names will also use less memory.

A string variable such as A$ can be assigned a particular string such as "THIS IS A STRING". This string contains 16 characters (including all blanks). Before this string can be assigned to A$, the variable A$ must be dimensioned to 16 using the DIM statement.*

```
10 DIM A$(16)
```

This statement will reserve 16 character positions for the string A$.

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 DIM A$(16)
20 A$="THIS IS A STRING"
```

*This use of the DIM statement in ATARI BASIC differs from the more common use of the DIM statement in other versions of BASIC to dimension string arrays.

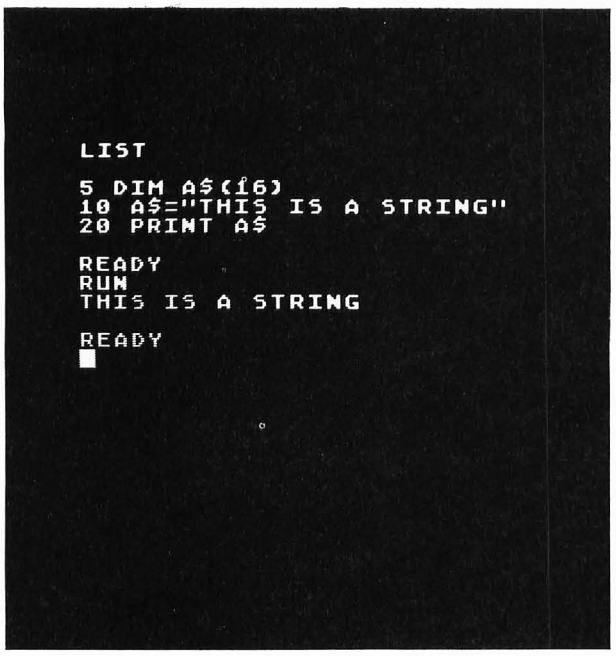
**Some versions of BASIC require you to use the word LET in an assignment statement. Thus, you would write

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.

30 PRINT A$

Try this. You should get the result shown in Figure 1.26.

FIGURE 1.26 Using string variables in a PRINT statement.



If you change line 10 in Figure 1.26 to

10 DIM A$(12)

then only the first 12 characters of the string will be printed. Try it. You can always dimension a string with a larger value than the actual number of characters in the string.

Of course, you can include graphic characters and cursor moves in your definition of a string variable. Thus, for example, to draw the square figure in Figure 1.18 you could define a string variable A$, as shown in Figure 1.27.

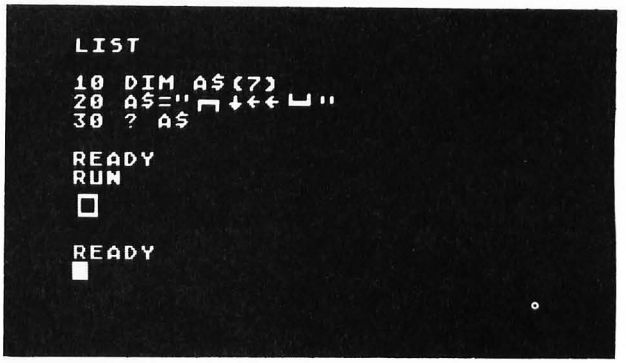


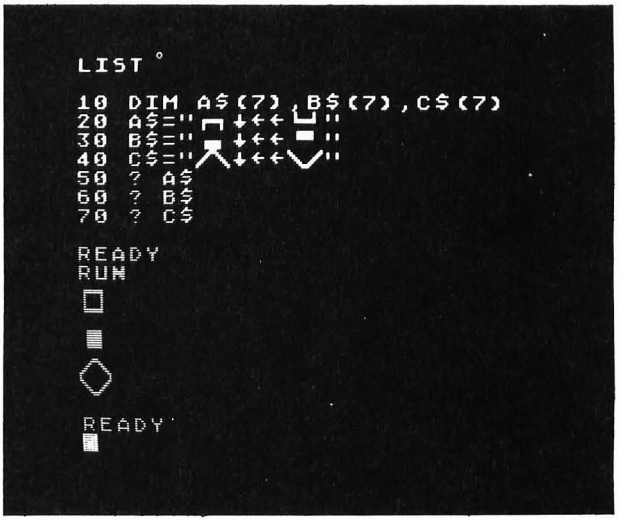
FIGURE 1.27 Defining a graphic figure as a string variable.

You can also define more than one graphic figure and then print them all. For example, the three graphic figures shown in Figure 1.12 can be defined as the following three string variables:

A$ = hollow square
B$ = solid square
C$ = diamond

Figure 1.28 shows a program that defines each of these string variables and then prints each figure. Type in this program and run it.

FIGURE 1.28 String variable definitions of the figures in Figure 1.12.



LINE LENGTH

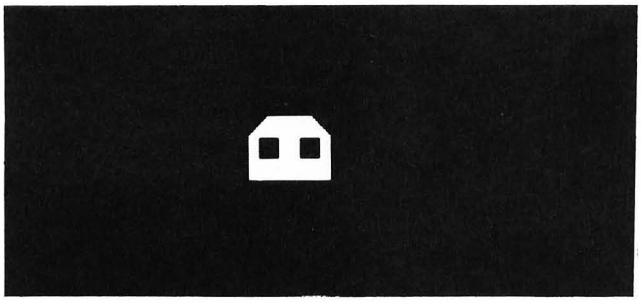
Each line on the ATARI screen contains 38 character positions. (There are really 40 character positions on the screen but the printing starts in the third column.) However, the computer is able to process up to three screen lines per BASIC statement. Thus, for example, if you are defining a string using a statement such as

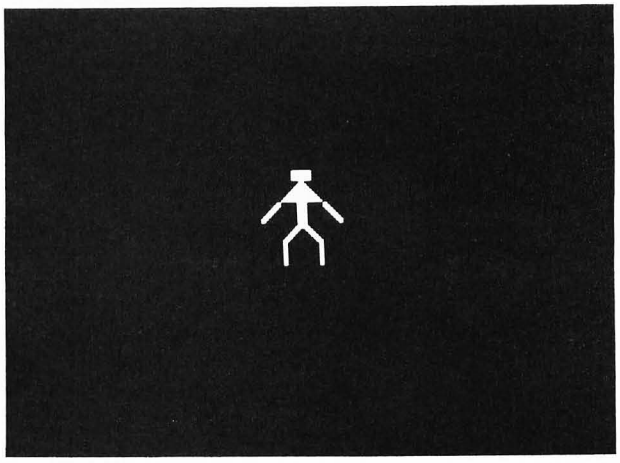
10 A$="-------

and you get to the end of the line on the screen, you just keep on typing. DO NOT PRESS RETURN. The ATARI will automatically continue the statement on the next line. When you finish the statement you must then press RETURN.

EXERCISE 1.3

Write and run a BASIC program that will draw each of the following graphic figures on the screen:





20 LET A$="THIS IS A STRING"

The use of the word LET is optional in ATARI BASIC. We will not use it.

2

LEARNING TO PROGRAM IN BASIC

In Chapter 1 you became familiar with using your ATARI keyboard. You also learned how to draw simple graphic figures. We will now begin to look at some of the ideas associated with writing BASIC programs.

In this chapter you will learn

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

THE BASIC PROGRAMMING LANGUAGE

The programming language BASIC was developed at Dartmouth College in 1963. The word BASIC stands for Beginners All-purpose Symbolic Instruction Code, and the language was designed to be easy to learn and easy to use. Over the years the BASIC language has been extended and modified by various manufacturers. ATARI BASIC is similar to the BASIC that is found on most microcomputers today.

The main advantages of using BASIC are that it is simple to use and is available on a cartridge for your ATARI. For all its simplicity, you will find that ATARI BASIC is quite powerful, allowing you to write high-performance programs fairly easily.

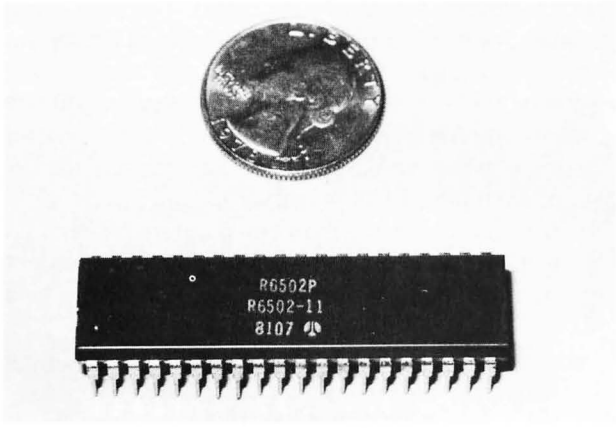
There are, however, certain drawbacks to ATARI BASIC. First of all, it is slow. You probably won't notice this until you try to draw a large picture quickly. The reason it is slow is that the ATARI contains a BASIC *interpreter* in its ROM cartridge. This means that each time you run your program the ATARI decodes and executes each of your BASIC statements one by one. This takes time.

Assembly Language

If you want to really speed up the execution time of a program you must write the program in *assembly*

language rather than BASIC. This is a lower-level language that the ATARI can execute directly. The "brain" of the ATARI is a 40-pin chip called a 6502 microprocessor. It is this chip, shown in Figure 2.1, that can 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 and PET computers also use a 6502 microprocessor. Radio Shack TRS-80 Level II and Level III computers use the Z80 microprocessor, which executes a completely different assembly language. The TRS-80 Color Computer uses a 6809 microprocessor, which executes still a different assembly language. We will not consider assembly language programming further in this book.

FIGURE 2.1 The 6502 microprocessor is the "brain" of the ATARI.



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 minimize 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 a particular task. 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. We will look at more details of the PRINT statement in the next chapter. *Semantics* refers to what it is that 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 is the hard part of computer programming. You must determine what you have to tell the computer in order for it to 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 will 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 any BASIC statements. Understanding exactly what you want to do is a major step in solving a problem.

It turns out that there are only a few basic rules for telling a computer what to do. Computers like to 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 5 and 8. The other thing computers like to do is to make a simple choice between two alternatives. This process of making choices will be described in Chapter 6. Any computer program can be constructed by combining loops with the process of making simple choices.

SAVING YOUR PROGRAMS

Your ATARI may have either a cassette tape recorder or a floppy disk drive connected to it. These devices are used to store your programs on either a cassette tape or a floppy disk. A floppy disk drive is more expensive than a cassette tape recorder, but it is much more convenient. You can store many programs on a floppy disk and retrieve any one quickly by name. In this section we will show you how to save a program on either a cassette tape or a floppy disk.

NEW

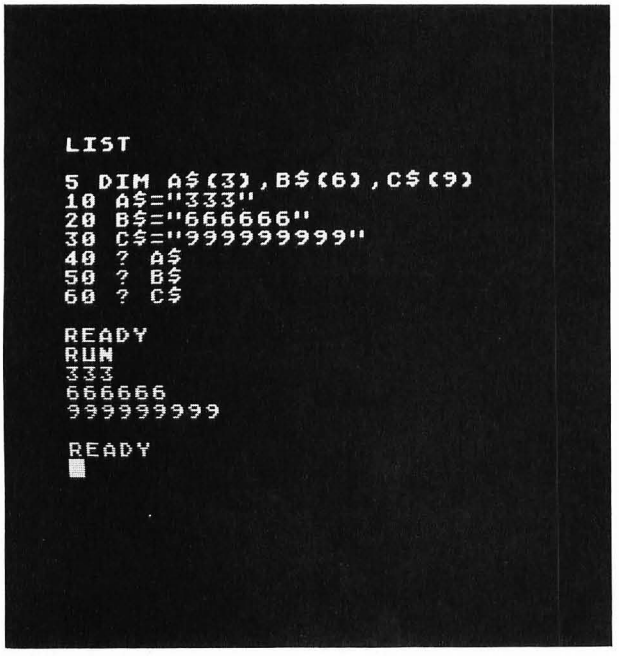
Type NEW followed by RETURN. This will clear any BASIC program that you have stored in the computer. You should type NEW before you begin typing in a new program. Failure to do this may cause parts of old programs to be combined with your new program.

Now type in the following program:

```
 5 DIM A$(3), B$(6), C$(9)
10 A$="333"
20 B$="666666"
30 C$="999999999"
40 ?A$
50 ?B$
60 ?C$
```

This program listing and its execution are shown in Figure 2.2.

FIGURE 2.2 This program prints three strips of numbers.



Cassette Tape Recorder

Suppose that you wish to save the program shown in Figure 2.2 on a cassette tape. First make sure that the tape recorder is properly connected to the ATARI and is plugged in. Rewind the tape (if necessary) and then type

CSAVE

followed by RETURN. You should hear two "beeps," indicating that you should press the PLAY and RECORD buttons on the recorder. Do this and then press the RETURN key on the ATARI keyboard. The tape should start moving. This means that the ATARI has started to store your program on the cassette tape. When it has finished writing your program on the tape (usually less than a minute), the READY message will reappear on the screen.

Your program is now stored on the cassette tape. In order to verify this, type NEW, which will clear your BASIC program in the ATARI. For example, if you now type LIST you will find that nothing gets listed. In order to retrieve your program, you must load it in from the cassette tape.

Rewind the tape and type

CLOAD

followed by RETURN. You should hear a single "beep," indicating that you should press the PLAY button on the recorder. Do this and then press the RETURN key on the ATARI keyboard. The tape should start moving. Your program is now being loaded into the ATARI. When the READY message returns to the screen, the tape will stop and your program will have been completely loaded. You can see the program listing now by typing LIST; you can execute the program by typing RUN.

You can store more than one program on each side of the tape and can use the counter on the recorder to position the tape to the proper location before using CLOAD or CSAVE.

Floppy Disk Drive

If your ATARI has a floppy disk drive connected to it, you can save your program on a floppy disk by typing

SAVE "D:NUMBERS"

where NUMBERS is the name of the program. You can make up any name containing up to eight letters or digits (the first character should be a letter) optionally followed by a suffix, .XXX. The top red light on the disk drive will light up and the disk drive will make a whirring sound for a few seconds while your program is being written on the disk.

After the top red light on the disk drive goes out, type DOS followed by A RETURN RETURN. This will list all of the programs that are stored on the disk. This list should now contain the name

NUMBERS

To verify that your program is really on the disk, type NEW, which will clear your BASIC program in the ATARI. If you now type LIST you will find that nothing gets listed. In order to retrieve your program from the disk, type

LOAD "D:NUMBERS"

The top red light on the disk drive will come on and you will hear the whirring sound again. Your program is being loaded into the ATARI memory from the disk. When the top red light goes out, your program will be completely loaded. You can see the program listing by typing LIST and execute the program by typing RUN.

It is possible to both load and run a program with a single command. If you type

RUN "D:NUMBERS"

the ATARI will first load the program NUMBERS from the floppy disk and then execute the program as soon as it is completely loaded into the ATARI memory.

STOPPING PROGRAM EXECUTION

Add the statement

70 GOTO 40

to the program shown in Figure 2.2. If the program in Figure 2.2 is in the computer memory, you can add this statement by simply typing it as shown. Type LIST in order to see the entire program. It should look like Figure 2.3.

FIGURE 2.3 Program to display a continuous sequence of number strings.

```
5 DIM A$(3),B$(6),C$(9)
10 A$="333"
20 B$="666666"
30 C$="999999999"
40 ? A$
50 ? B$
60 ? C$
70 GOTO 40
```

The statement

70 GOTO 40

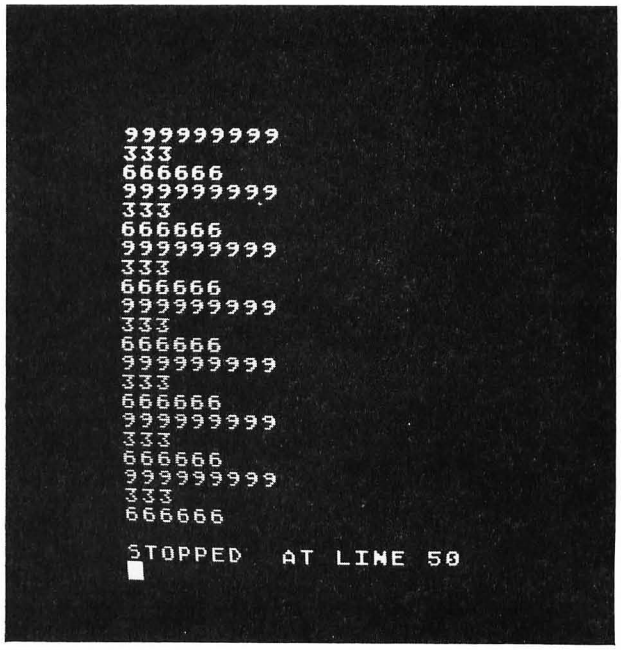
means exactly what it says. When statement number 70 is executed it simply branches back and executes statement number 40 again. This is a *loop* that continues indefinitely, as shown in Figure 2.4.

FIGURE 2.4 An indefinite loop that prints numbers until your press BREAK.

```
┌40 ?A$        prints three 3s
│ 50 ?B$       prints six 6s
│ 60 ?C$       prints nine 9s
└70 GOTO 40
```

Now run this program. As you can see, the three strings A$, B$, and C$ are being printed endlessly. In order to stop this program, press the BREAK key located above the RETURN key. Note that you get a BREAK message, as shown in Figure 2.5.

FIGURE 2.5 Stopping a program by pressing BREAK.



CONT

The program shown in Figure 2.3 displays three 3s, six 6s, and nine 9s over and over again. Each time the program is run it will always start with the three 3s. This can be seen in Figure 2.6, where the program was stopped by pressing BREAK just after displaying the six 6s. Note that the program starts again by displaying the three 3s.

If a program has been stopped, the statement CONT can be used to continue the program from the place where it left off. This is illustrated in Figure 2.7, where the program was again stopped just after displaying the six 6s. Note that after typing CONT the program restarts at the point of displaying the nine 9s.

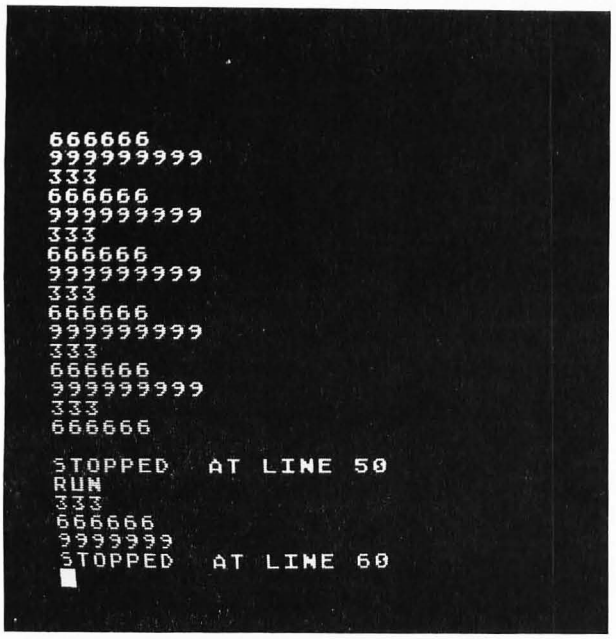


FIGURE 2.6 RUN causes the program to start at the beginning.

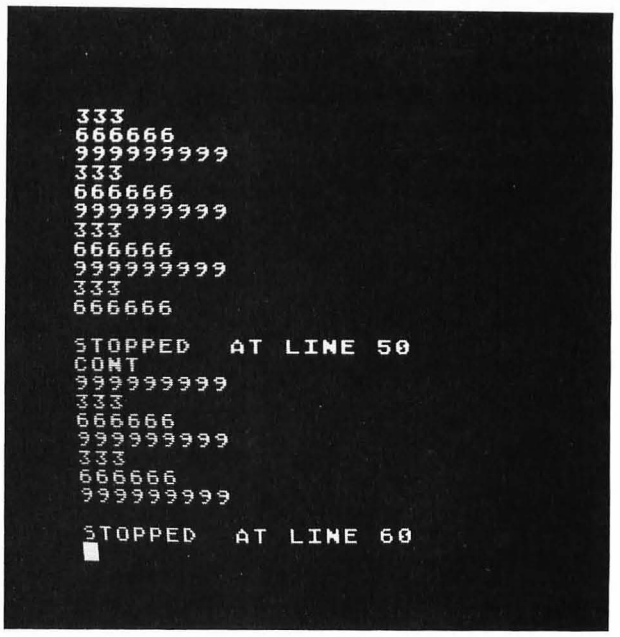


FIGURE 2.7 CONT causes the program to start from the place where it left off.

STOP

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

END

The END statement is traditionally used to stop a BASIC program at the end of the program. It does not cause a BREAK message as the STOP statement does. In ATARI BASIC the END statement is optional. As you have seen, we have not been using the END statement in our programs. A BASIC program in the ATARI will automatically stop 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, the statement with the lowest sequence number is executed first. Additional statements are then executed in order of increasing sequence number. Sequence numbers can be any integer between 0 and 32767.

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

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

This is done so that if you later want to insert a new statement between the second and third statement, you can just type

```
25 new statement
```

and this new statement will be inserted between statement 20 and statement 30. If you hadn't left any room between the second and third statements, you would have had to renumber all of your statements.

If you think that you may want to add some new

```
3 REM PROGRAM TO PRINT THREE STRINGS OF NUMBERS
5 DIM A$(3),B$(6),C$(9)
10 A$="333"
20 B$="666666"
30 C$="999999999"
40 ? A$
50 ? B$
60 ? C$
70 GOTO 40
```

FIGURE 2.8 Use of the REM statement to make remarks in a program.

statements at the beginning of your program, it would be a good idea to start your program with a higher sequence number, such as 100, and then continue with 110, 120, 130, and so on.

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 then be used for any kind of remark. These remarks are ignored by the ATARI when the program is executed. Their only purpose is to make the program easier to understand. For example, in the program shown in Figure 2.3 you may want to add the statement

```
3 REM PROGRAM TO PRINT THREE
STRINGS OF NUMBERS
```

as shown in Figure 2.8.

As mentioned earlier, any BASIC statement can use more than one screen line. When you type the remark in Figure 2.8 and reach the end of the first line, you must keep on typing. *Do not type* RETURN at the end of the first line or you will terminate the statement at that point. You must then start the next line with a new sequence number and another REM statement. R. or .(SPACE) can be used as an abbreviation for REM. They will both be changed to REM when the program is listed.

Multiple Statements per Line

ATARI BASIC allows you to write more than one BASIC statement per line by separating the statements with a colon (:). By a *line* we mean the characters from the line (or sequence) number to the RETURN, which may consist of up to three screen lines. 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. The remaining BASIC statements on the line are simply separated from the preceding one by a colon.

There are, however, some disadvantages to writing more than one statement per line. If it is done indiscriminately, it can result in a program that is very difficult to read and understand. You will not be able to branch 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, it is more difficult to insert a new statement between existing multiple statements. 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 tell what is going on in the program. For example, in Figure 2.9 we have added three remarks that tell what each PRINT statement prints. Note that a *colon* (:) is used to separate multiple statements on a single line.

FIGURE 2.9 Multiple statements on a single line are separated by a colon (:).

```
3 REM PROGRAM TO PRINT THREE STRINGS OF NUMBERS
5 DIM A$(3),B$(6),C$(9)
10 A$="333"
20 B$="666666"
30 C$="999999999"
40 ? A$:REM PRINT THREES
50 ? B$:REM PRINT SIXES
60 ? C$:REM PRINT NINES
70 GOTO 40
```

More about LIST

We have seen that the command LIST will list the entire BASIC program that is stored in memory.

It is also possible to list only selected parts of a program. For example, if you type

LIST 30

only the line with the sequence number 30 will be printed on the screen.

You can also list lines 20 through 40 by typing

LIST 20,40

These examples are shown in Figure 2.10.

FIGURE 2.10 Examples of the LIST statement.

```
LIST30
30 C$="9999999999"
READY
```

```
LIST 20,40
20 B$="666666"
30 C$="999999999"
40 ? A$
READY
```

Memory Locations and Computer Programs

A computer program is like a train going on a trip. The seats in the train are the memory locations or memory cells in the computer. 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$. Each seat would have a different name.

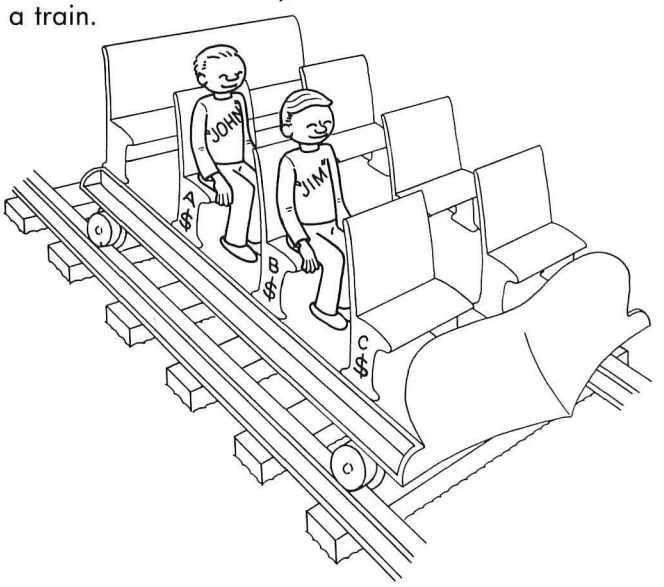
Whoever or whatever is in a particular seat corresponds to the *contents* of a particular memory location in the computer. For example, if "JOHN" is sitting in seat A$, then the BASIC statement

A$="JOHN"

can be interpreted as meaning: Put "JOHN" in seat A$. It is very important to clearly distinguish between the name of the memory location or seat on the train (A$) and the contents of that memory location or seat ("JOHN"). See Figure 2.11.

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 name does *not* end with a dollar sign, the 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 2.11 Memory locations are like seats on a train.



3

LEARNING MORE ABOUT PRINT

In the first two chapters of this book you have written short programs that print strings. In this chapter you will see how the ATARI can work with *numbers* as well as strings. You will find that the ATARI can serve as a very good calculator.

In this chapter you will learn

1. how to use the ATARI as a calculator
2. to write arithmetic expressions involving addition, subtraction, mutliplication, division, and exponentiation
3. how to use the comma and semicolon in a PRINT statement
4. how to use the TAB key in a PRINT statement
5. to use the POSITION statement
6. how to display letters in *reverse video* and lower case
7. some of the built-in functions in the ATARI.

THE ATARI AS A CALCULATOR

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

Addition

If you type

```
PRINT 5+3
```

the ATARI will respond with 8. You can use the question mark as an abbreviation for PRINT. Thus if you type

```
?5+3
```

the ATARI will also respond with 8, as shown in Figure 3.1. Try it.

Subtraction

If you type

```
?12-5
```

the ATARI will respond with 7, as shown in Figure 3.1. Try it.

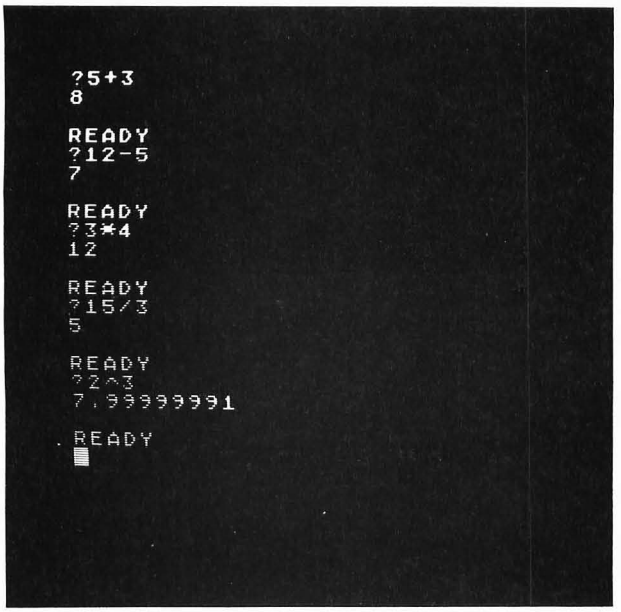


FIGURE 3.1 Using the ATARI in the calculator mode.

Multiplication

The symbol for multiplication in BASIC is the asterisk (*). Thus, if you type

?3*4

the ATARI will respond with 12, as shown in Figure 3.1. Try it.

Division

The symbol for division in BASIC is the slash (/). Thus, if you type

?15/3

the ATARI will respond with 5, as shown in Figure 3.1. Try it.

Exponentiation

The symbol for exponentiation in BASIC is the upward arrow (^); to print it press the same key as the asterisk (*) while holding the SHIFT key down. Thus, if you want to raise 2 to the power of 3 (2 cubed) you would type

?2^3

and the ATARI should respond with 8. Try it. As you can see, the ATARI displays 7.99999991 due to truncation errors. Note that when the exponent is an integer, exponentiation is equivalent to repeated multiplication. Thus,

2^3=2*2*2

Arithmetic Expressions

The arithmetic operators +, −, *, /, and ^ can be combined in a single arithmetic expression. For example, if you type

?5+3−2

the ATARI will respond with 6. What do you think the ATARI will display if you type the following expression?

?6+12/2+4

Try it. Did it display what you thought it would?

You have found that the ATARI gave the answer 16. This is because the ATARI does division before addition. All computer languages don't work this way. For example, the language APL evaluates all expressions from right to left. Thus, it would give the preceding expression a value of 8. Do you see why?

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, the expression is evaluated from *left to right.* Parentheses can always be used to change the 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 ATARI to check your results. The answers are shown in Figure 3.2.

FIGURE 3.2 Evaluation of arithmetic expressions on the ATARI.

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

Did you guess the correct answer to the last expression? Remember that the two divisions are evaluated from left to right, so 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 the last example in Figure 3.2 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 ATARI. Any time you want to multiply anything on the ATARI you *must* use the multiplication symbol *.

NUMERICAL VARIABLES

We have seen that strings such as "JOHN" can be stored in memory cells with names such as S3$. If a memory cell name does *not* end with a dollar sign, the ATARI will assume that the memory cell contains a numerical value. For example, if you type

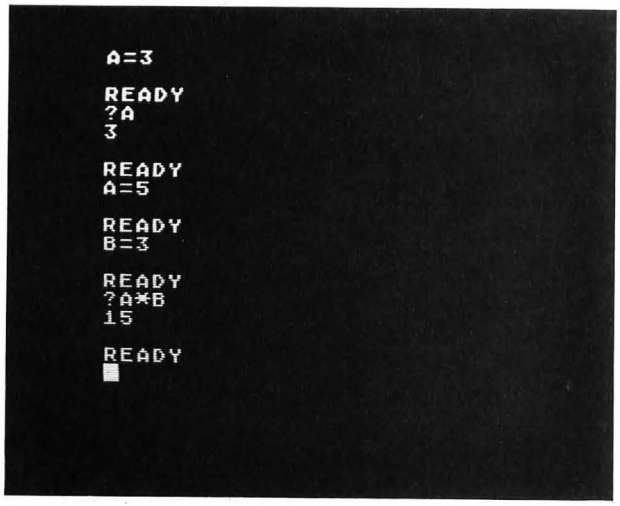
```
A=3
?A
```

the ATARI will respond with 3, as shown in Figure 3.3. Similarly, if you type

```
A=5
B=3
?A*B
```

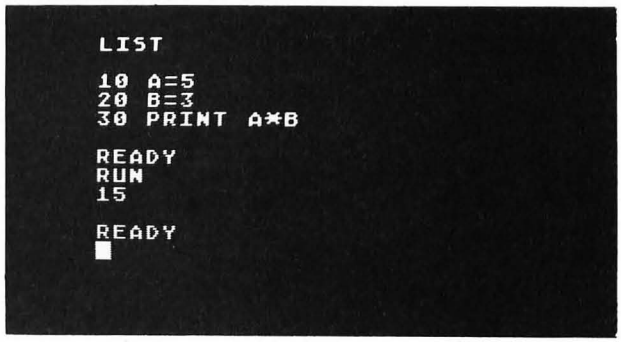
the ATARI will respond with 15, as shown in Figure 3.3.

FIGURE 3.3 Numerical variables can be used in the immediate mode of execution and in arithmetic expressions.



Note that these examples use the immediate mode of execution. The deferred mode of execution can also be used, as shown in Figure 3.4.

FIGURE 3.4 Use of numerical variables in the deferred mode of execution.



How many digits of a number does the ATARI display? Try typing

```
?1/3
```

and

```
?2/3
```

as shown in Figure 3.5. Note that 10 digits are displayed (excluding the leading zero) with any remaining digits truncated.

FIGURE 3.5 The ATARI displays 10 digits.

Scientific Notation

What happens if you type in a number containing 11 or more digits? Try typing

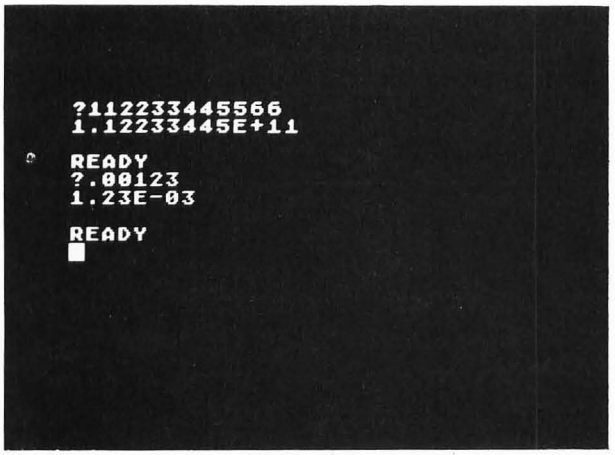
?112233445566

as shown in Figure 3.6. Note that the ATARI has 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 you must move the decimal point in order to obtain the correct number. 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 3.6.

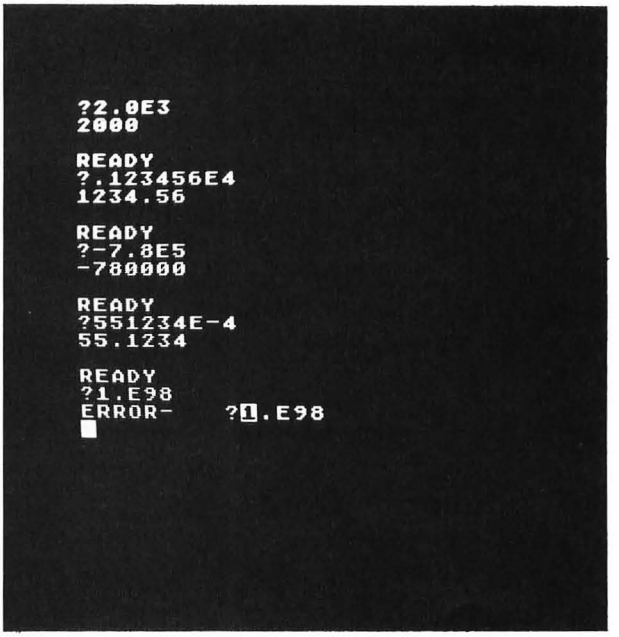
FIGURE 3.6 Scientific notation is used by the ATARI for numbers greater than 9999999999 and less than 0.01.



The ATARI uses scientific notation for numbers greater than 9999999999 and less than 0.01. You can use scientific notation if you want; the ATARI will convert it to standard notation if your number is between 0.01 and 9999999999. Some examples are shown in Figure 3.7. Note that the ATARI printed ERROR when we tried to print 1E98.

If you try to store a number larger than 1E97 you will get an error. Also, any number with a magnitude smaller than 1E–98 will be stored in the ATARI as 0. However, only two digits can be used in the exponent when using scientific notation.

FIGURE 3.7 You can use scientific notation in your programs.



CONTROLLING PRINTED OUTPUT

When you use the PRINT statement you can control where on the screen the output is printed by using commas, semicolons, the TAB key, and the POSITION statement.

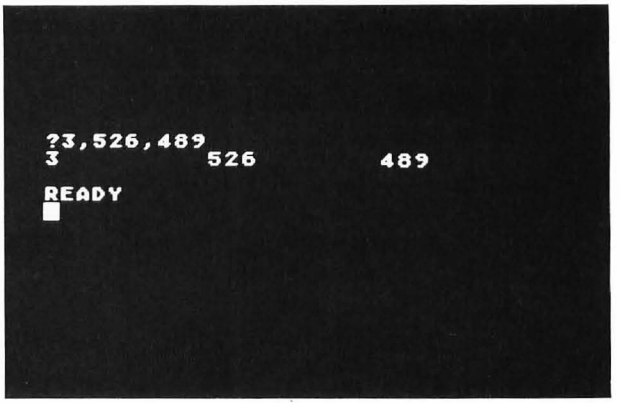
Comma

The comma has a special meaning in BASIC. It *cannot* be used in the customary way to separate every 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

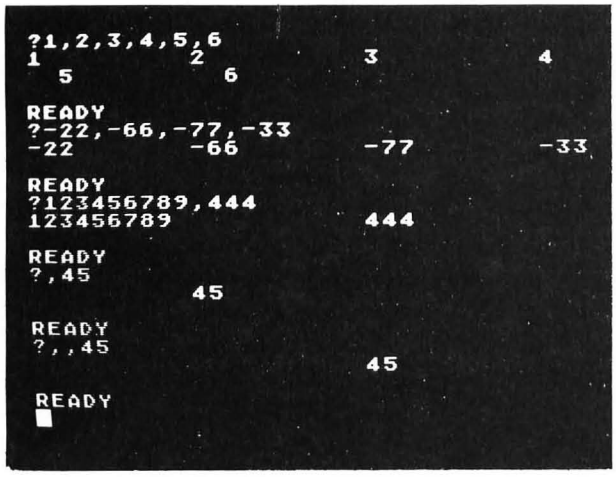
?3,526,489

FIGURE 3.8 The comma acts like a *tab* in a PRINT statement.



as shown in Figure 3.8. Note that instead of printing one number the ATARI thought you wanted to print the *three* numbers 3, 526, and 489. In a PRINT statement the comma is used to move to the next fixed *tab* position. The fixed tab positions are located in columns 2, 12, 22, and 32, where the screen columns are numbered 0 through 39. (The first print position on the screen is really column number 2, as you will see.) If you try to print more than four numbers on a line, separated by commas, the extra numbers will be printed on the next line, as shown in Figure 3.9 (example 1). Note in the second example of Figure 3.9 that the negative sign in a negative number is printed at the tab position. If the number contains more than eight digits, a second number is moved to the next tab position, as shown by the third example of Figure 3.9. One or more commas can precede a number in order to skip tab positions, as shown in the last two examples of Figure 3.9.

FIGURE 3.9 Using the comma as a tab.

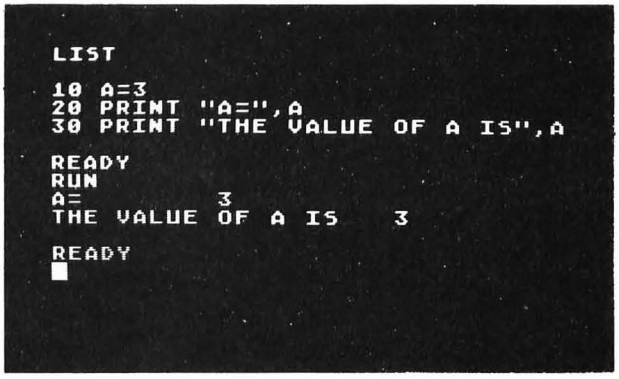


The comma can also be used with strings, as shown in Figure 3.10. Note that up to eight characters can be included in a string before a tab position is skipped prior to printing a second string. Also note that strings begin printing in column number 2 and at the start of all other tab positions (12, 22, and 32).

FIGURE 3.10 Using the comma tab with strings.

The comma can be used in PRINT statements to separate strings from numerical variables, as shown in Figure 3.11. Note that after the string "A=" is printed, the comma causes a tab to column number 12 before the value of A (3) is printed. This looks a little awkward. This gap can be eliminated by using a semicolon instead of a comma.

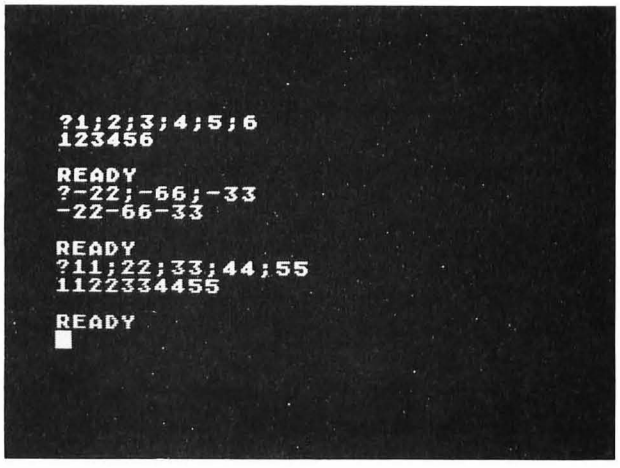
FIGURE 3.11 Using the comma to separate strings and numerical variables.



Semicolon

If numerical values are separated by semicolons instead of commas, then no space is inserted after each value, as shown in Figure 3.12. Note that commas and semicolons can be mixed in a single PRINT statement.

FIGURE 3.12 Using the semicolon to separate numerical values.



When it is used with *strings* the semicolon leaves no blank spaces between two strings, as shown in Figure 3.13. When strings and numerical variables are combined, the semicolon can be used to eliminate unsightly gaps, as shown in Figure 3.14. Note that you need to include a blank space at the end of the string in line 30 of Figure 3.14 in order to leave a space before the number.

FIGURE 3.13 The semicolon leaves no blank spaces between strings.

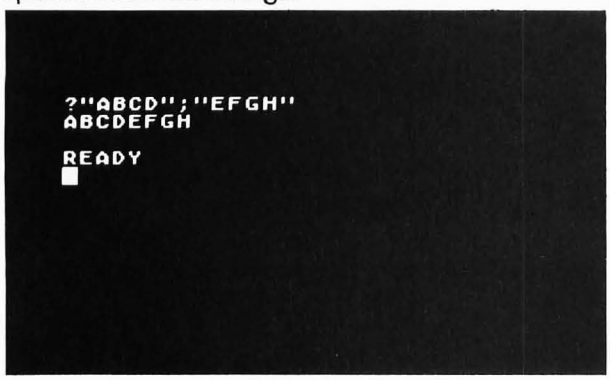
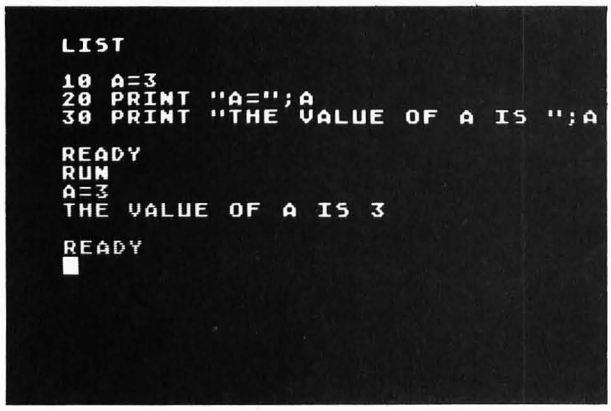


FIGURE 3.14 Using the semicolon to separate strings and numerical variables.



The TAB Key

The TAB key is located on the left side of the keyboard just below the ESC key. Press this key several times. The cursor should move across the screen, stopping at various tab positions. The cursor will eventually be tabbed to the beginning of the next line.

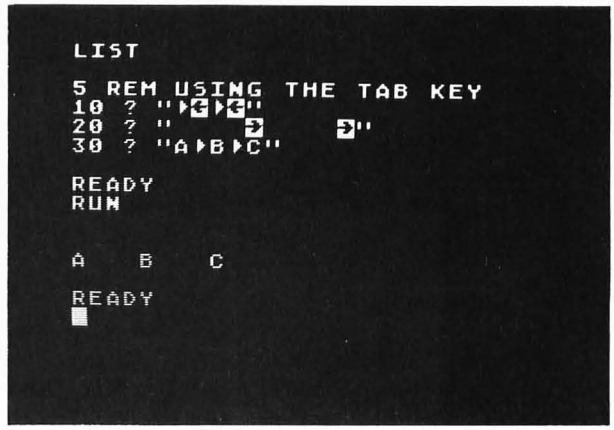
Any of these tab positions can be cleared by moving the cursor to the tab position (pressing the TAB key) and then pressing the TAB key while holding down the CTRL key (CTRL TAB). To set a new tab position, move the cursor to the desired location and then press the TAB key while holding down the SHIFT key (SHIFT TAB).

If you press the ESC key before any of these keystrokes and include them all in a string in a PRINT statement, then these tab operations can be performed in a BASIC program. For example, the statement

10?"ESC TAB ESC CTRL TAB ESC TAB ESC CTRL TAB"

will clear the first two tab positions on a line. This statement will appear on the screen as shown in line 10 in Figure 3.15. Type this statement.

FIGURE 3.15 Using the TAB key in PRINT statements.



The statement

20?"SPACE SPACE SPACE SPACE ESC SHIFT TAB SPACE SPACE SPACE SPACE ESC SHIFT TAB"

will set two tabs at screen columns 6 and 10 (remember that the printing starts at column 2). This statement will appear on the screen as shown in line 20 in Figure 3.15. Type this statement.

The statement

30 ?"A ESC TAB B ESC TAB C"

will print the letter A, tab to the next tab position, print the letter B, tab to the next position, and print the letter C. This statement will appear on the screen as shown in line 30 in Figure 3.15. Type this statement and run the program. The result should be as shown in Figure 3.15.

The POSITION Statement

Sometimes you may want to print something at a particular location on the screen. This can be done using the POSITION statement before a PRINT statement. The screen contains 24 rows of 40 characters for a total of 960 different print locations.

The statement

POSITION X,Y

will move an "invisible" cursor to column number X (0–39) and row number Y (0–23). A PRINT statement containing a string will then print the string starting at location X,Y.

As an example, suppose that you want to print the message shown in Figure 3.16. You can use the POSITION statement to move to the starting location for each string, as shown in the program listing in Figure 3.17. Statement 5 in Figure 3.17a will clear the screen. It is typed by pressing the keys

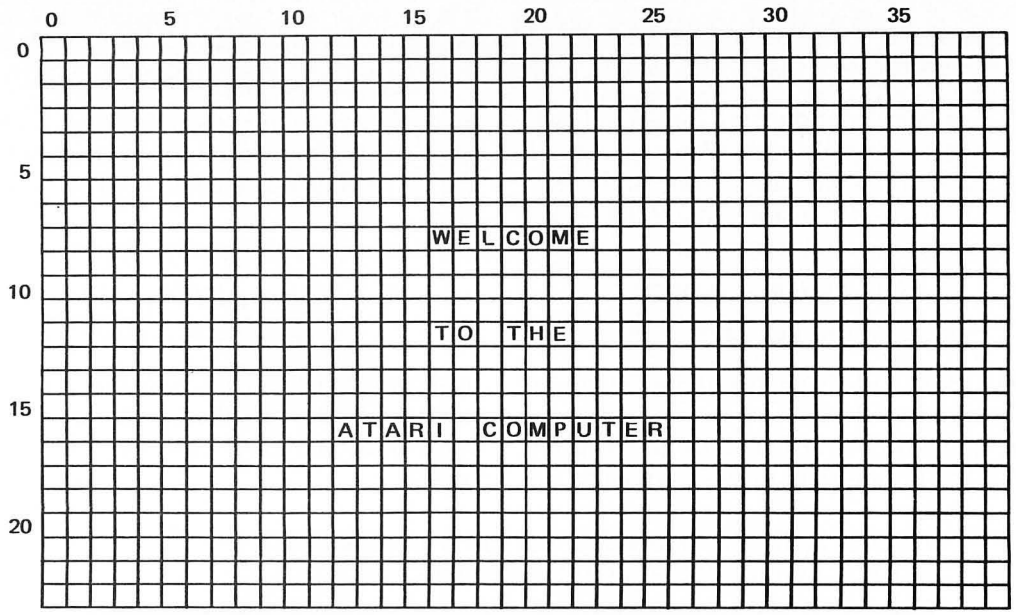


FIGURE 3.16 Screen layout for POSITION statement.

```
5 ?"ESC SHIFT<"
```

In the POSITION statement the screen coordinates X,Y can be variables, and numerical as well as string variables can be printed. For example, if X = 10, Y = 6, and A = −2, then the statements

```
POSITION X,Y
PRINT A
```

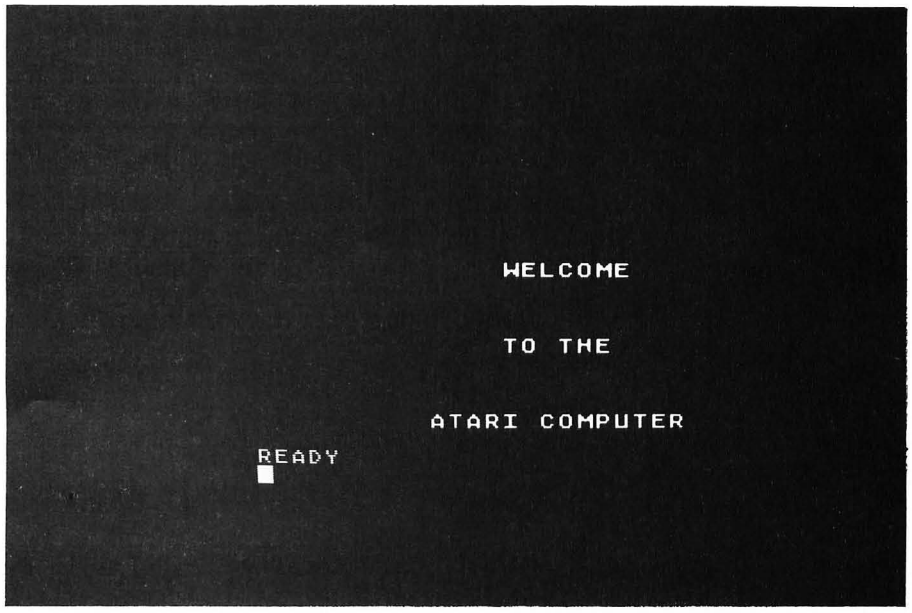
will print the value −2 at the screen location 10,6.

FIGURE 3.17 The PRINT statement can be used following the POSITION statement to print a string anywhere on the screen.

(a)

```
5 ? "▮"
10 POSITION 16,8
20 ? "WELCOME"
30 POSITION 16,12
40 ? "TO THE"
50 POSITION 12,16
60 ? "ATARI COMPUTER"
```

(b)



REVERSE VIDEO AND LOWER-CASE LETTERS

You saw in Chapter 1 that pressing the ATARI logo key will change to reverse video. Pressing it again will change back to normal video. You can include reverse video characters in a string in a PRINT statement. For example, type the statement

```
10 ? "THIS IS REVERSE VIDEO"
   press ATARI key here
```

as shown in Figure 3.18. Note that all the letters are displayed in reverse video.

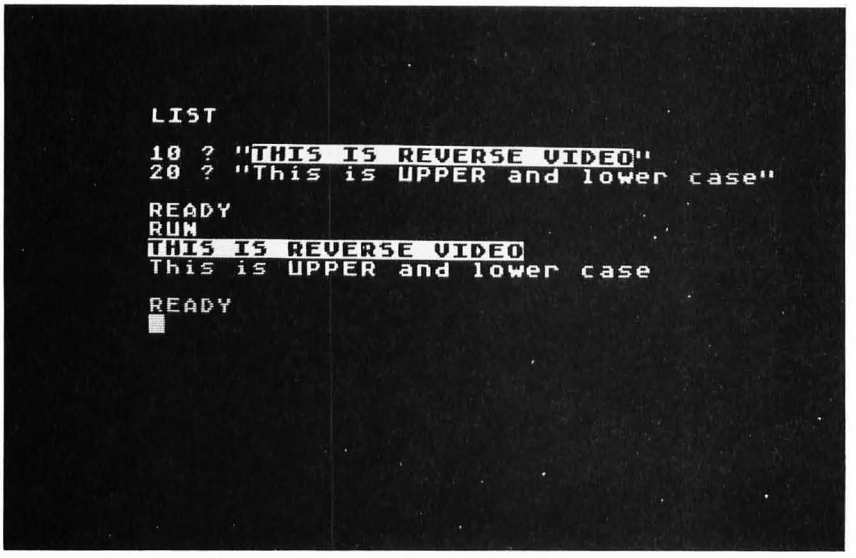
You can type lower-case letters by pressing the CAPS/LOWR key on the right of the keyboard. The SHIFT key can then be used to type capital letters as on a regular typewriter. To return to all caps, press the CAPS/LOWR key while holding down the SHIFT key.

As an example, type the statement

```
20 ? "This is UPPER and lower case"
     Press CAPS/LOWR key here
```

as shown in Figure 3.18. Note that when the program in Figure 3.18 is run, the two strings including the reverse video and lower-case letters are printed on the screen.

FIGURE 3.18 Reverse video and lower-case letters can be included in PRINT statements.



SOME BUILT-IN FUNCTIONS

The ATARI has a number of built-in functions that simplify many calculations. You may use any of these you care to in your programs.

The Functions ABS, INT, and SGN

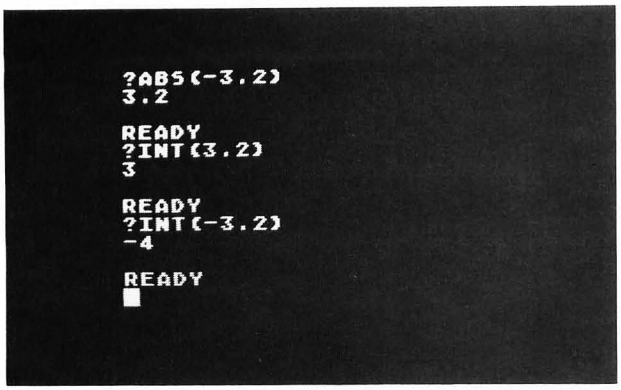
The *absolute value* of a number is the magnitude of a number without regard to its sign. The absolute value of a number can be found by using the built-in function ABS(X). Thus, for example, if X = −7 the value of ABS(X) will be 7.

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 ATARI will round to the next lower *signed number*. Thus, if X = −3.25, the value of INT(X) will be − 4.

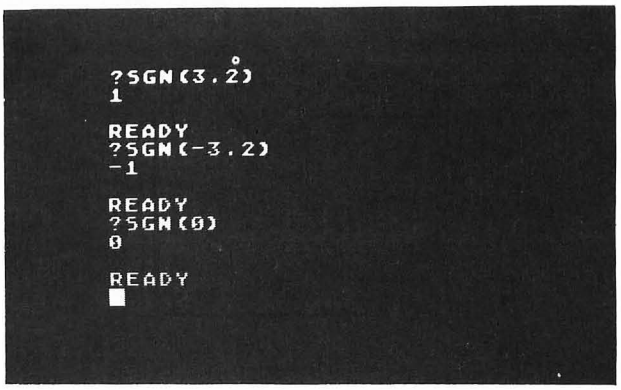
The function SGN(X) can be used to determine the *sign of a number*. It can have the following three values:

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

Examples using ABS, INT, and SGN are shown in Figure 3.19.



(a)



(b)

FIGURE 3.19 Finding the absolute value ABS, the integer part INT, and the sign SGN of a number.

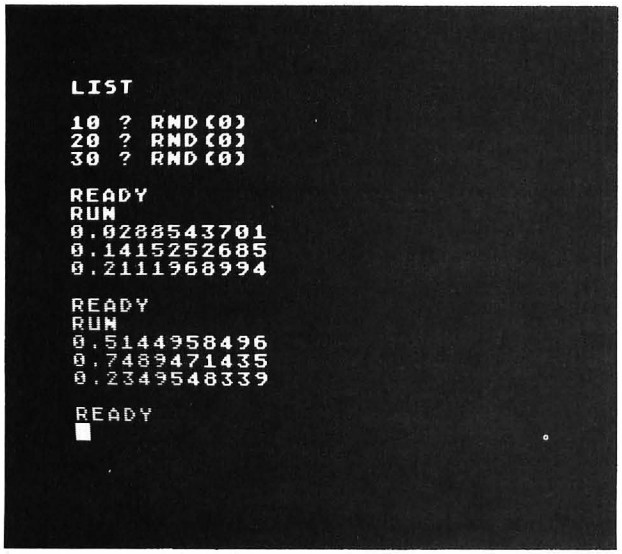
Random Numbers

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

Type in and run the following program twice as shown in Figure 3.20.

```
10 ? RND(0)
20 ? RND(0)
30 ? RND(0)
```

FIGURE 3.20 The function RND(0) produces a random number between 0 and 1.



The function RND(0) will return a pseudorandom number between 0 and 1. Each time RND(0) is called it produces a different number between 0 and 1. The argument 0 is of no particular significance. Any number could be used.

Square Root

The square root of a number can be found by using the BASIC function

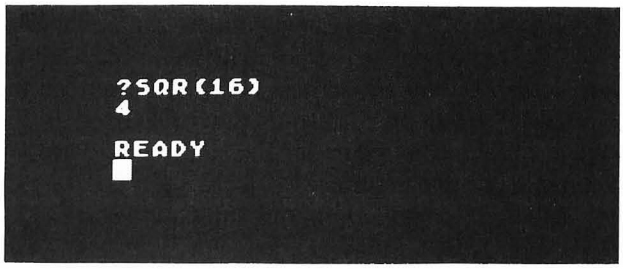
SQR(X)

where X is a positive number. For example, to find the square root of 16, type

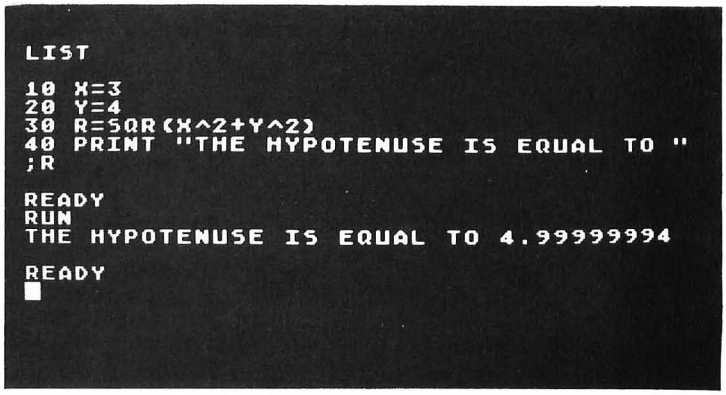
?SQR(16)

as shown in Figure 3.21a. To find the hypotenuse R of the right triangle shown in Figure 3.22, you could use the program shown in Figure 3.21b. Note that the use of the exponentiation operator ^ resulted in a truncation error. The exact value of the hypotenuse is 5.

FIGURE 3.21 Use of the square root function SQR.



(a)



(b)

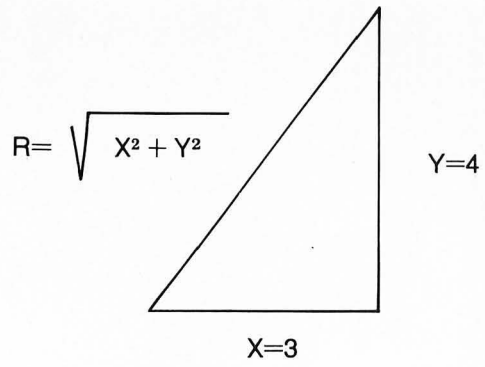


FIGURE 3.22 Finding the hypotenuse of a right triangle.

Trigonometric Functions

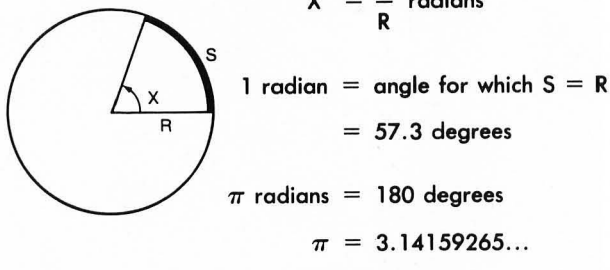
The ATARI contains the following built-in trigonometric functions:

ATARI Function	*Value of Function*
SIN(X)	sine of X
COS(X)	cosine of X
ATN(Y)	arctangent of Y

In these expressions, X is a numeric constant, variable, or expression that represents the value of an angle. The angle X is normally expressed in *radians*. You can change this so that X is expressed in degrees by executing the statement DEG. Once DEG has been executed, you must execute the statement RAD in order to go back to radians. The value of ATN(Y) is expressed in either radians (with RAD set) in the range ± 1.57, or in degrees (with DEG set) in the range ± 90. The argument Y is a numeric constant, variable, or expression.

The definition of a radian is shown in Figure 3.23.

FIGURE 3.23 Definition of a radian.



To convert degrees to radians, multiply by π/180. Examples using the trigonometric functions are shown in Figure 3.24.

```
PI=3.1415926

READY
?SIN(45*PI/180)
0.7071067759

READY
DEG

READY
?SIN(45)
0.70710678

READY
?COS(60)
0.50000001

READY
?ATN(2)
63.43494902

READY
```

FIGURE 3.24 Using the trigonometric functions SIN, COS, and ATN.

Natural Logarithms and the Exponential Function

Consider the equation

$$y = b^x$$

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 $e = 2.718281\ldots$, we say that y is the exponential function $y = e^x$ and x is the natural logarithm of y:

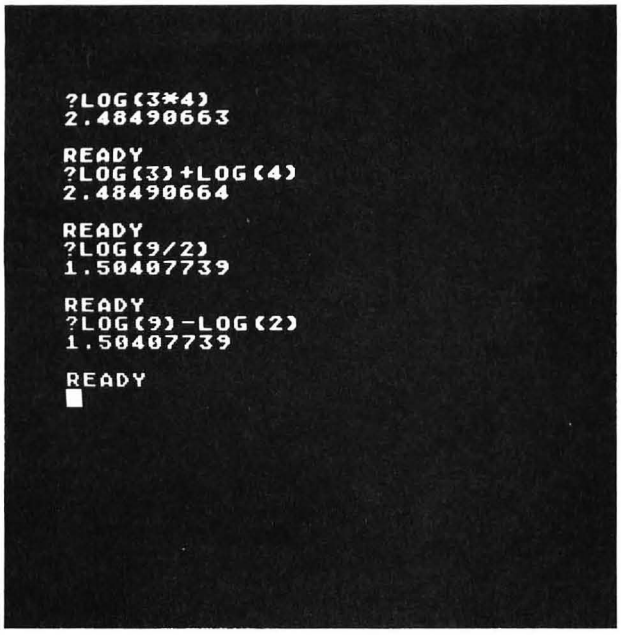
$$x = ln\ y$$

In BASIC e^x can be computed using the function EXP(X), and *ln X* can be computed using the function LOG(X). If the base b is 10, the logarithm to the base 10 can be computed using the function CLOG(X).

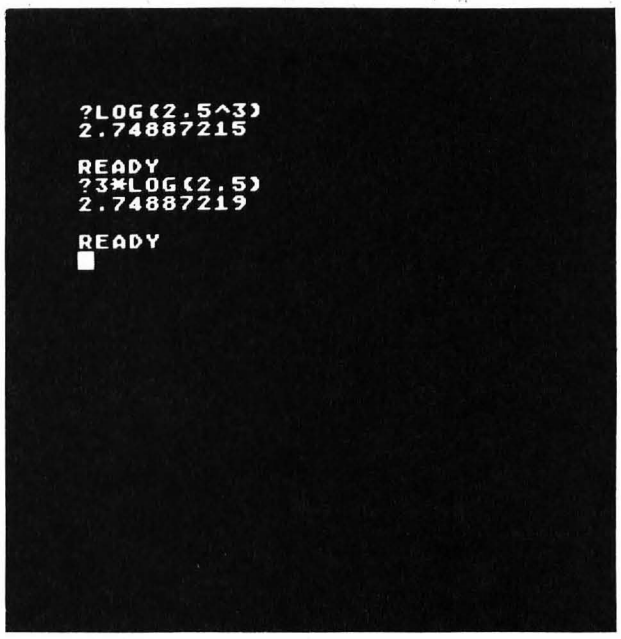
The following properties of logarithms are illustrated in the examples shown in Figure 3.25:

```
LOG(A*B)=LOG(A)+LOG(B)
LOG(A/B)=LOG(A)-LOG(B)
LOG(A^K)=K*LOG(A)
```

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



(a)



(b)

FIGURE 3.25 Properties of logarithms.

$$X = De^{PT/100}$$

For example, to find the amount of money you would earn in 7 years by investing $3,000 at 9.5 percent interest compounded continuously, type

?3000*EXP(9.5*7/100)

as shown in Figure 3.26.

FIGURE 3.26 Examples related to the exponential function.

Note that the answer is more than $5,833 or almost double your original investment. A characteristic of exponential growth is a constant doubling time T_d. From the equation for X we see that X will be equal to $2D$ in the time T_d, where

$$2D = De^{\frac{PT_d}{100}}$$

or

$$2 = e^{\frac{PT_d}{100}}$$

Taking the natural logarithm of both sides of this equation and using the third property of logarithms illustrated in Figure 3.25, we obtain

$$ln(2) = \frac{PT_d}{100} ln(e)$$

$$= \frac{PT_d}{100}$$

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 3.26. We therefore 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 percent inflation rate will double prices every 7 years.

EXERCISE 3.1

Let the variables A, B, C, and D have the following values:

$$A = 2,\ B = 3,\ C = 4,\ D = 5$$

Use the ATARI to evaluate the following expressions:

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

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

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

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

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

4

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 ATARI output various forms of data on the screen. In this chapter you will learn how to make the ATARI 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 sample programs that will show you how to

1. add two numbers
2. compute the area of a rectangle
3. compute the area of a circle
4. calculate gas mileage
5. display your name and address
6. make sounds with the ATARI.

THE INPUT STATEMENT

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

```
10 INPUT R
10 INPUT A,B
10 INPUT A$
```

When the first INPUT statement is executed, the ATARI 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 ATARI will expect you to enter *two* numerical values, separated by a comma. If you press RETURN after entering only one value, the ATARI will print another 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 store whatever you type on the screen in the string variable A$. The use of the INPUT statement will be illustrated in the following sample programs.

SUM OF TWO NUMBERS

Figure 4.1 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.

Lines 20 and 25 print 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 in Figure 4.1b, the two number 5 and 9 were entered from the keyboard. Line 40 then printed 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 of 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 in Figure 4.1b, the value 8 was entered for the first number. But then the RETURN key was pressed. Note that the ATARI responded with another question mark asking you to enter the second number. In this example − 3 was then entered.

In the third example the RETURN key was pressed without entering any data. This caused the ERROR- 8 message. All error codes are given in Appendix C.

This program will continue to ask you for two more numbers. To stop the program, press the BREAK key.

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

FIGURE 4.1 Sample program to add two numbers.

```
10 REM PROGRAM TO SUM TWO NUMBERS
20 PRINT "ENTER 2 NUMBERS"
25 PRINT "SEPARATED BY A COMMA"
30 INPUT A,B
40 PRINT A;"+";B;"=";A+B
50 PRINT
60 GOTO 20
```

(a)

```
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
?

ERROR-   8 AT LINE 30
```

(b)

AREA OF A RECTANGLE

Figure 4.2 shows the listing and a sample run of a program that computes the area of a rectangle, where the lengths of the two sides 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 in line 20 ends with a semicolon. Note that when you do this, the question mark follows the prompt messsage and the cursor remains on the *same line* as the message. Thus, you enter the data on the same line as the prompting message.

FIGURE 4.2 Program to calculate the area of a rectangle.

```
10 REM PROGRAM TO COMPUTE THE
15 REM AREA OF A RECTANGLE
20 ? "ENTER 2 SIDES OF RECTANGLE ";
30 INPUT X,Y
40 ? "THE AREA OF A RECTANGLE"
45 ? "WITH SIDES ";
50 ? X;" AND ";Y
55 ? "IS EQUAL TO ";X*Y
60 ?
70 GOTO 20
```

```
RUN
ENTER 2 SIDES OF RECTANGLE ?4,5
THE AREA OF A RECTANGLE
WITH SIDES 4 AND 5
IS EQUAL TO 20

ENTER 2 SIDES OF RECTANGLE ?3.6,7.5
THE AREA OF A RECTANGLE
WITH SIDES 3.6 AND 7.5
IS EQUAL TO 27

ENTER 2 SIDES OF RECTANGLE ?6,

ERROR-   8 AT LINE 30
```

AREA OF A CIRCLE

The area of a circle of radius r is given by

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

where π (pi) is approximately equal to 3.14159265. Figure 4.3 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.

Line 35 calculates the area of the circle. The value of pi has been defined in line 15. Note that in the second example after RUN two values, 6 and 3, were entered. But the ATARI was expecting only one value. It therefore used only the first value (6) and ignored the second value (3).

In the third example after RUN a value of 2.5E51 was entered. But this results in a value of the area A that is larger than 1E98; therefore, an overflow error message occurred. (See Figure 3.7.)

FIGURE 4.3 Program to calculate the area of a circle.

```
10 REM PROGRAM TO COMPUTE THE AREA OF A CIRCLE
15 PI=3.14159265
20 ? "ENTER A RADIUS ";
30 INPUT R
35 A=PI*R^2
40 ? "THE AREA OF THE CIRCLE IS ";A
50 ?
60 GOTO 20
```

```
RUN
ENTER A RADIUS ?35.2
THE AREA OF THE CIRCLE IS 3892.558831

ENTER A RADIUS ?6,3
THE AREA OF THE CIRCLE IS 113.097228

ENTER A RADIUS ?2.5E51

ERROR-   11 AT LINE 35
```

GAS MILEAGE

The program shown in Figure 4.4 computes gas mileage in miles per gallon (MPG). The reading of the odometer (the device that displays the mileage on the dashboard) at the last fillup is stored in memory cell M1 in line 25. The odometer reading at the present fillup is stored in memory cell M2 in line 35. The number of gallons it takes to fill the tank is stored in memory cell G in line 45. The total miles traveled since the last fillup is equal to M2 − M1. Therefore, the number of miles per gallon is given by (M2 − M1) / G. This is calculated in line 50 and stored in the memory cell MPG. It is printed on the screen in line 60.

FIGURE 4.4 Program for computing gas mileage.

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

A sample run is shown in Figure 4.5a. The answer is printed as 19.55696202 MPG. This answer contains many more digits after the decimal point than are meaningful. After all, because of variations in filling the tank it probably only makes sense to compute the MPG to the nearest tenth. How can we have the ATARI display the MPG to the nearest tenth? The following steps will do it:

1. Multiply the present value by 10:
19.55696202 × 10 = 195.5696202

2. Add 0.5:
195.5696202 + 0.5 = 196.0696202

3. Take the *interger part* of the result:
INT(196.0696202) = 196

4. Divide by 10:
196/10 = 19.6

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

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

Note that the result is stored back in memory cell MPG. Therefore, if you add this statement to the program shown in Figure 4.4 and run the program with the same values used in Figure 4.5a, the result will be as shown in Figure 4.5b.

The example shown in Figure 4.5c shows that if you mistakenly press RETURN when the INPUT statement is waiting for a value for the gallons G in line 45, the ATARI will produce the ERROR- 8 message.

```
RUN
ENTER PREVIOUS ODOMETER READING
?12345
ENTER NEW ODOMETER READING
?12654
ENTER GALLONS SINCE LAST FILLUP
?15.8
GAS MILEAGE: 19.55696202 MPG

READY
```

```
RUN
ENTER PREVIOUS ODOMETER READING
?12345
ENTER NEW ODOMETER READING
?12654
ENTER GALLONS SINCE LAST FILLUP
?15.8
GAS MILEAGE: 19.6 MPG

READY
```

```
RUN
ENTER PREVIOUS ODOMETER READING
?12345
ENTER NEW ODOMETER READING
?12654
ENTER GALLONS SINCE LAST FILLUP
?

ERROR-   8 AT LINE 45
```

FIGURE 4.5 Sample runs of gas mileage program.

NAME AND ADDRESS

The INPUT statement can be used to enter *string* data as well as numerical data into the computer. The statement

```
INPUT A$
```

will assign whatever characters you type to the string variable A$. As an example, consider the program shown in Figure 4.6. Line 15 contains the statement PRINT "ESC CTRL CLEAR", which clears the screen. Line 30 will assign whatever you type for your name to the string variable N$. Line 50 will assign whatever you type for your street address to the string variable S$. Line 70 will assign whatever you type for your city, state, and zip code to the string variable C$. Lines 80–100 will then print these three strings on three separate lines.

A sample run of this program is shown in Figure 4.7. Note that only one string variable can be used in an INPUT statement. This is because a comma, which is normally used to separate two numerical inputs, will just be assigned to the first string variable, as is shown for the city and state in Figure 4.7. Quotation marks will also be assigned to the string variable and can therefore be printed on the screen.

FIGURE 4.6 Program to display your name and address.

```
10 REM NAME AND ADDRESS
12 DIM N$(30),S$(30),C$(30)
15 ? "ESC CTRL CLEAR"
20 ? "ENTER YOUR NAME"
30 INPUT N$
40 ? "ENTER YOUR STREET ADDRESS"
50 INPUT S$
60 ? "ENTER YOUR CITY, STATE, AND ZIP"
70 INPUT C$
80 ? N$
90 ? S$
100 ? C$
```

You can make a single tone on your TV speaker by using the statement

SOUND V,P,D,L

In this expression V is a voice number between 0 and 3, P is a pitch value between 0 and 255, D is a distortion number (an even number between 0 and 14), and L is a volume or loudness value between 1 and 15.

A value of P = 1 gives the lowest tone and a value of P = 255 gives the highest tone. A value of L = 1 gives the weakest tone and a value of L = 15 gives the loudest tone. The tone will stay on until another SOUND statement (or an END statement) is executed. To turn a sound off, execute a SOUND statement with the same voice value V used to create the sound and a loudness value of 0.

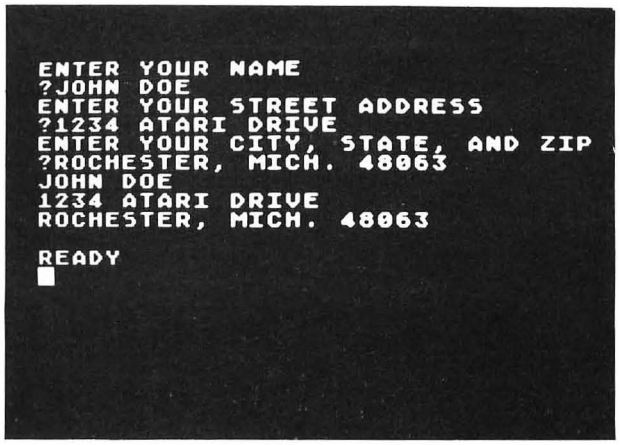
In order to try out some notes, turn up the volume on your TV set and type in and run the following program:

```
10 REM MAKING SOUNDS
20 PRINT "ENTER PITCH AND LOUDNESS"
30 INPUT P,L
40 SOUND 0, P, 10, L
50 INPUT A$
60 SOUND 0, 0, 0, 0
70 GOTO 20
```

After the sound is turned on in line 40, the program will wait at line 50 until you press RETURN. Line 60 will then turn the sound off.

Try several different values for P and L. The value of 10 used for the distortion D produces a "pure" tone. Try changing this to other values.

FIGURE 4.7 Sample run of program shown in Figure 4.6.



EXERCISE 4.1

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 input a temperature in °F and print on the screen the temperature in both °F and °C.

EXERCISE 4.2

Four different sounds corresponding to the four voice numbers 0–3 can be played at the same time. Write a program that will ask you to enter a pitch, distortion, and loudness value for each of four voices and then play the four sounds simultaneously. Have one sound at a time stop each time you press the RETURN key.

5

A REPETITION LOOP—LEARNING ABOUT FOR . . . NEXT

In Chapter 2 you learned how to use the GOTO statement to form a continuous loop. There is a looping structure available in BASIC, called a FOR . . . NEXT loop, which is particularly useful when you know the number of 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 dashed lines using the FOR . . . NEXT loop
3. how to use nested FOR . . . NEXT loops.

THE FOR . . . NEXT LOOP

The general form of a FOR . . . NEXT loop is shown in Figure 5.1. When statement 10 is executed, the value of I 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 5.1 General form of the FOR . . . NEXT loop.

```
10 FOR I=M1 TO M2 STEP M3
20 _____
30 _____
40 _____
50 NEXT I
60 _____
```

Immediate Mode Execution of the FOR . . . NEXT loop

In order to see how the FOR . . . NEXT loop works, try typing in the following examples in the immediate mode.

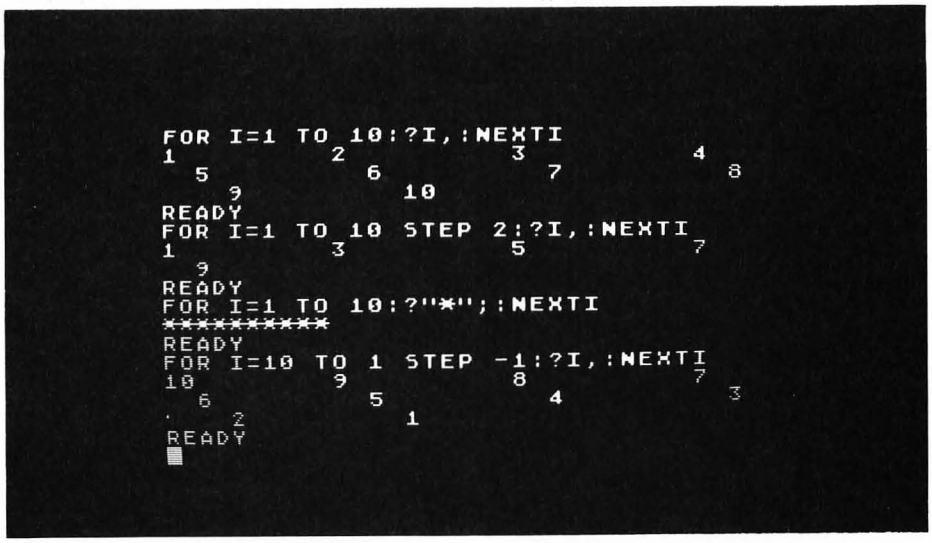


FIGURE 5.2 Using the FOR . . . NEXT loop in the immediate mode.

```
FOR I=1 TO 10:?I, :NEXTI
FOR I=1 TO 10 STEP 2:?I, :NEXTI
FOR I=1 TO 10:?"*", :NEXTI
FOR I=10 TO 1 STEP -1:?I, :NEXTI
```

These examples are shown in Figure 5.2.

Drawing Lines Using POSITION X,Y

Review the use of the POSITION statement in Chapter 3. By including the statements

```
POSITION X,Y
PRINT "X";
```

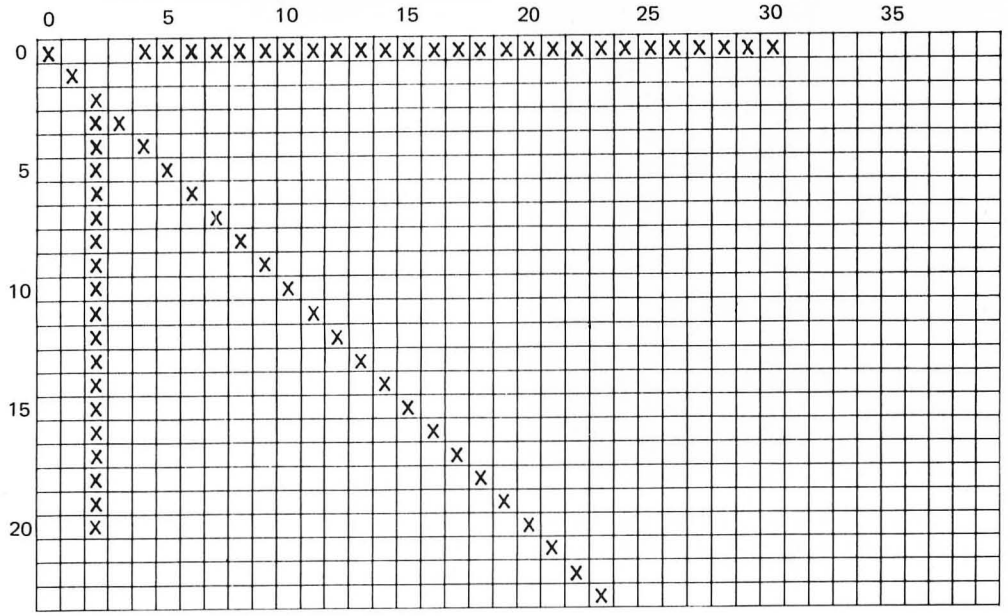
in a FOR . . . NEXT loop, we can print a series of "X"s on the screen. The relative position of the "X"s will depend upon how the values of X and Y are incremented in the FOR . . . NEXT loop.

For example, Figure 5.3 shows a horizontal, vertical, and diagonal line drawn on a screen layout. The horizontal line can be drawn by letting X vary from 4 to 30 in steps of 1 as shown in Figure 5.4. The statement

```
70 GOTO 70
```

is used to prevent the ready message from returning to the screen. Press BREAK to stop the program.

FIGURE 5.3 A horizontal, vertical, and diagonal line to be drawn using the POSITION statement.



```
10 REM HORIZONTAL LINE
20 ? "{CLR}"
30 FOR X=4 TO 30
40 POSITION X,0
50 ? "X"
60 NEXT X
70 GOTO 70
```

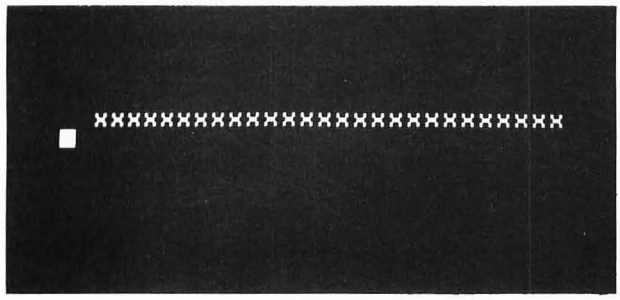
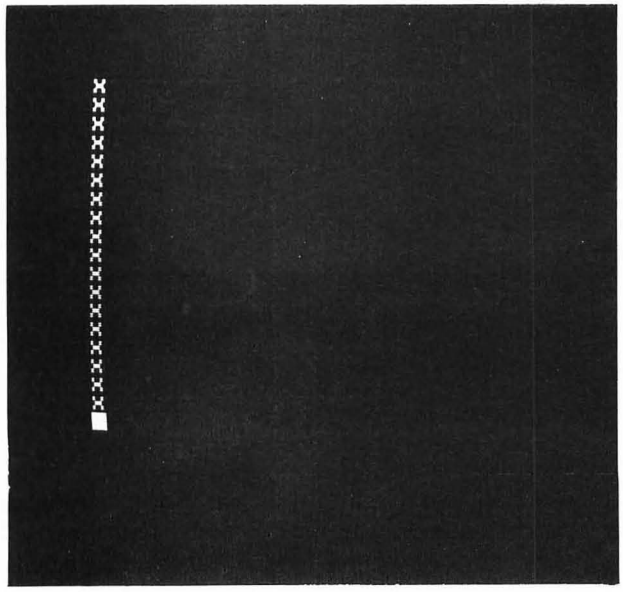


FIGURE 5.4 Drawing a horizontal line using POSITION X,0.

The vertical line can be drawn by letting Y vary from 3 to 20 in steps 1 as shown in Figure 5.5.

FIGURE 5.5 Drawing a vertical line using POSITION 2,Y.

```
10 REM VERTICAL LINE
20 ? "{CLR}"
30 FOR Y=3 TO 20
40 POSITION 2,Y
50 ? "X"
60 NEXT Y
70 GOTO 70
```



The diagonal line can be drawn by letting X vary from 0 to 23 in steps of 1 with Y = X, as shown in Figure 5.6.

Vertical and horizontal lines can be combined to form a border, as shown in Figure 5.7.

FIGURE 5.6 Drawing a diagonal line using POSITION X,Y.

```
10 REM DIAGONAL LINE
20 ? "{CLR}"
30 FOR X=0 TO 23
35 Y=X
40 POSITION X,Y
50 ? "X"
60 NEXT X
70 GOTO 70
```

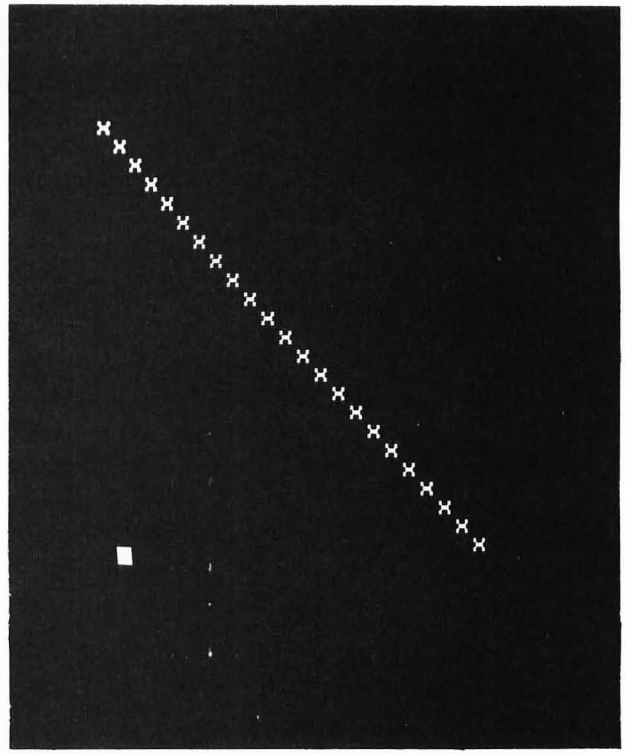
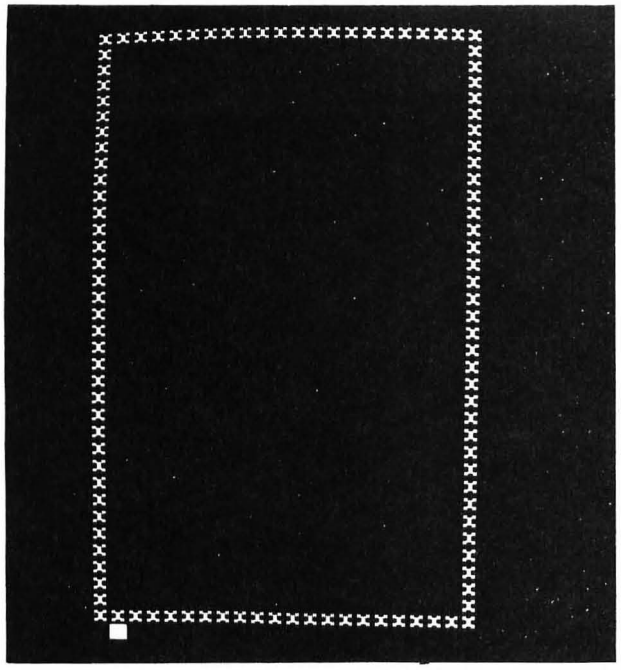


FIGURE 5.7 Drawing a border using POSITION X,Y.

```
10 REM BORDER
20 ? "{CLR}"
30 FOR X=2 TO 37
40 POSITION X,1
50 ? "X";
60 POSITION X,22
70 ? "X";
80 NEXT X
90 FOR Y=2 TO 21
100 POSITION 2,Y
110 ? "X";
120 POSITION 37,Y
130 ? "X";
140 NEXT Y
150 GOTO 150
```



EXERCISE 5.1

Draw a border around the message

WELCOME
TO THE
ATARI COMPUTER

shown in Figure 3.16.

EXERCISE 5.2

Draw your name in block letters using asterisks and the POSITION statement.

FIGURE 5.7 (cont.)

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 an Array of Points

Clear the screen and type, in the immediate mode,

```
Y=20
FOR X=2 TO 32 STEP
5:POSITION X,Y:?"*":NEXTX
```

A line of seven asterisks spaced five positions apart should appear near the bottom of the screen, as shown in Figure 5.8.

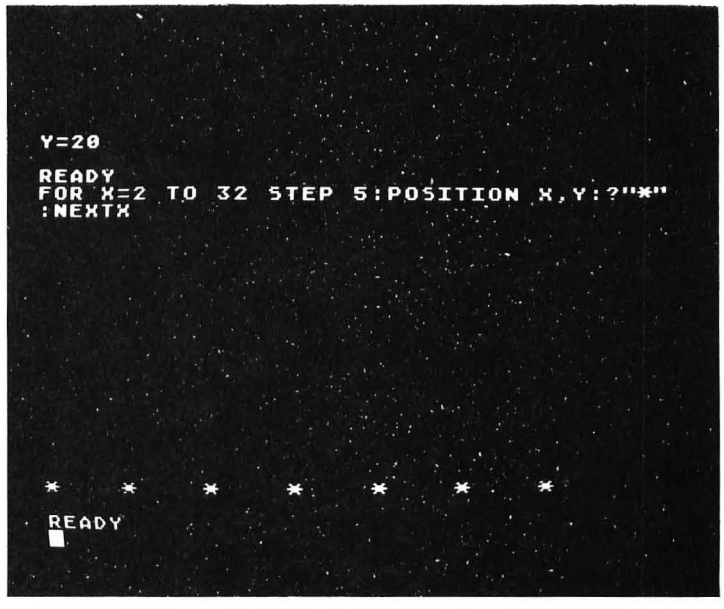
If you now let Y vary from 2 to 20 in steps of 3 you can plot seven rows, each containing seven asterisks. The program shown in Figure 5.9 will do this. Lines 30–60 form the FOR . . . NEXT loop used to plot a single row of asterisks, as shown in Figure 5.8. The outer FOR . . . NEXT loop starting at line 20 plots seven of these rows as Y varies from 2 to 20 in steps of 3.

Note that every time through the outer FOR . . . NEXT loop (lines 20–70) the inner FOR . . . NEXT loop (lines 30–60) is executed completely. That is, the inner loop loops seven times (and therefore plots seven asterisks) every time the outer loop loops once. Since the outer loop also loops seven times, a total of $7 \times 7 = 49$ asterisks will be plotted on the screen.

FIGURE 5.9 Program to plot an array of points.

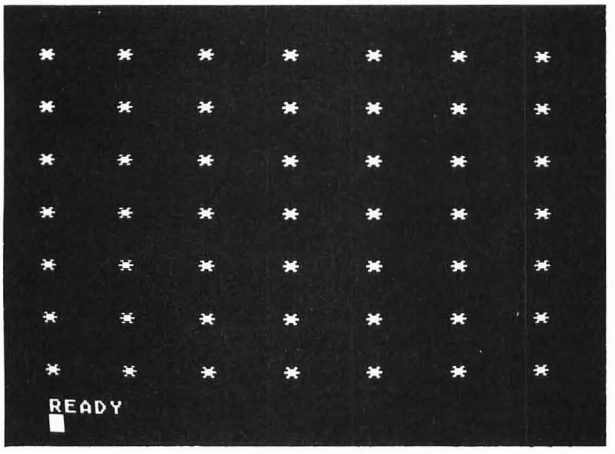
```
10 REM ARRAY OF POINTS
15 ? "{CLEAR}"
20 FOR Y=2 TO 20 STEP 3
30 FOR X=2 TO 32 STEP 5
40 POSITION X,Y
50 ? "*"
60 NEXT X
70 NEXT Y
```

FIGURE 5.8 Plotting a single row of seven asterisks.



Type in this program and run it. You should obtain the array of asterisks shown in Figure 5.10. Modify this program by changing the number of rows, the number of points plotted in each row, and the spacing between the points.

FIGURE 5.10 Array of points plotted using the program in Figure 5.9.



Changing Screen Colors

As another example of nested FOR . . . NEXT loops, consider the program shown in Figure 5.11. This program will continually change the color of the screen every second.

```
10 REM CHANGING COLORS
15 ? "{CLEAR}"
20 FOR C=0 TO 15
30 SETCOLOR 2,C,10
40 FOR I=1 TO 350:NEXT I
50 NEXT C
60 GOTO 20
```

FIGURE 5.11 Program to change the color of the screeen every second.

The statement SETCOLOR 2,C,10 changes the background screen color to the hue value C, where C is a number between 0 and 15. We will discuss the statement SETCOLOR in Chapter 7. For now, type in the program in Figure 5.11 and run it.

Line 15 clears the screen. The outer FOR . . . NEXT loop (lines 20–50) changes the screen color C from 0 to 15. Line 30 actually changes the screen to the color C. Line 40 is an inner FOR . . . NEXT loop that just uses up some time. Letting I increment from 1 to 350 will use up about 1 second. To make a longer delay just change 350 to a larger number. To make a shorter delay, change 350 to a smaller number. Line 60 branches back to line 20, which runs the program again.

Press the SYSTEM RESET key to stop the program. Change the length of time that each color is displayed.

SOUND EFFECTS

Type in the following line in the immediate mode:

```
FOR P=1 TO 255:SOUND 0,P,10,10:NEXTP:END
```

You will hear the speaker produce all possible pitch values one after the other. This should give you an idea of how to add sound effects to your programs. Let's look at some examples.

Producing Multiple Clicks

The program shown in Figure 5.12 will produce N clicks with pitch P and a separation of S seconds. Lines 70–100 loop N times; a click is produced each time line 80 is executed. Line 90 is a delay equal to about S seconds.

Run the program and set P = 150, S = 0.5, and N = 10. You should hear 10 clicks occurring every half-second. Enter different values of N, S, and P to

FIGURE 5.12 Program to produce N clicks with pitch P and a separation of S seconds.

```
10 REM PRODUCE N CLICKS WITH
20 REM SPACING S AND PITCH P
25 ? "ENTER PITCH P ";
30 INPUT P
35 ? "ENTER SEPARATION (SEC) ";
40 INPUT S
45 ? "ENTER NUMBER OF CLICKS ";
50 INPUT N
60 L=10
70 FOR I=1 TO N
80 SOUND 0,P,10,L
85 SOUND 0,0,0,0
90 FOR J=1 TO 350*S:NEXT J
100 NEXT I
110 ?
120 GOTO 25
```

produce different clicking effects. Press the BREAK key to stop the program. The loudness of the clicks can be changed by changing the value of L in line 60.

Producing a Phaser Noise

If you repeatedly execute SOUND 0,P,10,10 with different pitch values P, you can produce a variety of effects. For example, the program shown in Figure 5.13 produces a "phaser" noise 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, type in the program and run it for values of NC = 6, P1 = 10, P2 = 220, and DP = 5. Try a variety of different values for NC, P1, P2, and DP. Press the BREAK key to stop the program.

FIGURE 5.13 Program for making a "phaser" noise.

```
10 REM PHASER NOISE
20 REM NC=# CYCLES
30 REM P1=STARTING PITCH
40 REM P2=ENDING PITCH
50 REM DP=PITCH INCREMENT
60 ? "ENTER # OF CYCLES ";
65 INPUT NC
70 ? "ENTER STARTING PITCH ";
75 INPUT P1
80 ? "ENTER ENDING PITCH ";
85 INPUT P2
90 ? "ENTER PITCH INCREMENT ";
95 INPUT DP
100 L=10
110 FOR J=1 TO NC
120 FOR P=P1 TO P2 STEP DP
125 SOUND 0,P,10,L
130 NEXT P
135 NEXT J
140 SOUND 0,0,0,0
145 ?
150 GOTO 60
```

Producing a Siren Sound

A siren noise can be produced by repeatedly executing SOUND 0,P,10,L first with increasing values of P and then with decreasing values of P, as shown in Figure 5.14. The outer FOR . . . NEXT loop from lines 110 to 160 produces NC complete cycles of the siren sound. The loop in lines 120–135 produces the increasing sound and the loop in lines 140–155 produces the decreasing sound. Each of these loops executes SOUND 0,P,10,10, and produces a tone of pitch P and duration T. The FOR . . . NEXT loops in lines 130 and 150 produce delays (while the tone is on) proportional to T.

Type in this program and run it for values of NC = 5, P1 = 20, P2 = 220, DP = 3, and T = 1. Try changing the values of NC, P1, P2, DP, and T to produce different siren sounds. Press the BREAK key to stop the program.

FIGURE 5.14 Program to produce a siren sound.

```
10 REM SIREN NOISE
20 REM NC=# CYCLES
30 REM P1=STARTING PITCH
40 REM P2=ENDING PITCH
50 REM DP=PITCH INCREMENT
60 ? "ENTER # OF CYCLES ";
65 INPUT NC
70 ? "ENTER STARTING PITCH ";
75 INPUT P1
80 ? "ENTER ENDING PITCH ";
85 INPUT P2
90 ? "ENTER PITCH INCREMENT ";
95 INPUT DP
100 ? "ENTER HOLDING TIME ";
105 INPUT T
110 FOR J=1 TO NC
120 FOR P=P1 TO P2 STEP DP
125 SOUND 0,P,10,10
130 FOR K=1 TO T:NEXT K
135 NEXT P
140 FOR P=P2 TO P1 STEP -DP
145 SOUND 0,P,10,10
150 FOR K=1 TO T:NEXT K
155 NEXT P
160 NEXT J
170 SOUND 0,0,0,0
175 ?
180 GOTO 60
```

PLOTTING GRAPHIC PATTERNS

As another example of using nested FOR . . . NEXT loops, consider the program shown in Figure 5.15. Line 15 dimensions the string variable G$, which contains a graphic symbol entered through the keyboard in line 30. This graphic symbol is printed as a 10 × 10 array using the nested FOR . . . NEXT loops

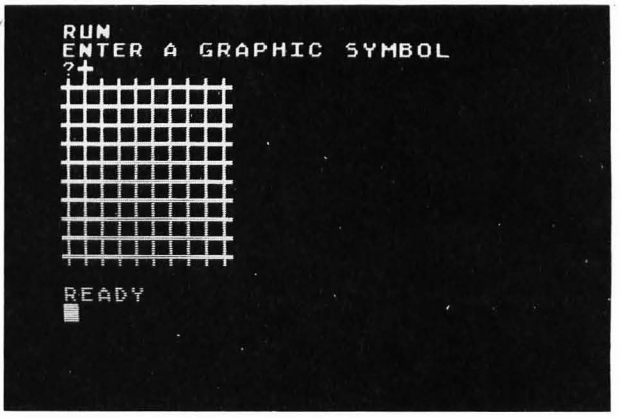
FIGURE 5.15 Program to produce graphic patterns.

```
10 REM GRAPHIC PICTURES
15 DIM G$(1)
20 ? "ENTER A GRAPHIC SYMBOL"
30 INPUT G$
40 FOR Y=1 TO 10
50 FOR X=1 TO 10
60 ? G$;
70 NEXT X
80 ?
90 NEXT Y
```

in lines 40–90. The inner loop in lines 50–70 prints 10 copies of the graphic symbol on a single row. The PRINT statement(?) in line 80 moves the cursor to the next screen line. The outer loop in lines 40–90 plots 10 rows of the graphic strip plotted by the inner loop.

Type in this program and run it. A sample run is shown in figure 5.16. Try different graphic symbols to produce a variety of patterns.

FIGURE 5.16 Sample run of program in Figure 5.15.



EXERCISE 5.3

Modify the program in Figure 5.15 so that G$ contains a string of three different graphic symbols entered from the keyboard. Have the resulting pattern contain 20 rows.

6

MAKING CHOICES—LEARNING ABOUT IF . . . THEN

Up to this point all of the programs that we have written have consisted of a sequence of instructions and simple loops. However, the thing that makes computers appear to be smart is their ability to make a decision based on the current state of affairs. The primary decision-making statement in BASIC is the IF . . . THEN statement. This statement allows the ATARI 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 ATARI's relational operators
3. the meaning of the ATARI's logical operators
4. about flowcharts and structured flowcharts.

THE IF . . . THEN STATEMENT

The IF . . . THEN statement in BASIC allows your program to conditionally execute some statements or to conditionally branch to some other statement. The following 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 a later section of this chapter.

In the first form of the IF . . . THEN statement, 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 behaves in a similar way to the first form. However, if the logical expression is *true*, all of the statements fol-

lowing the word THEN are executed. 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, if the logical expression is *true,* the program will branch to "line number." This form is equivalent to the first form, where the statement is a GOTO statement. Thus, for example, the following two statements are equivalent:

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

We will illustrate the use of the IF . . . THEN statement by adding some conditional statements to the programs we wrote in Chapter 4.

Gas Mileage Program

In the gas mileage program shown in Figure 4.4. of Chapter 4, M1 is the old odometer reading and M2 is the new odometer reading. Now to make any sense, M2 must be greater than M1 (M2 > M1). It is always a good idea when writing computer programs to check the data entered through the keyboard to try to detect any typing errors. For example, if after you have entered the value of M2 in line 35, M1 is greater than M2, then a typing error has probably been made. In any event M2 is too small to make sense. Thus, we could add the statements

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

to the program in Figure 4.4, as shown in Figure 6.1.

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 number 37 caught it, printed the message READING TOO SMALL, and then branched back to statement number 20, where the program started over again.

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

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

```
10 REM GAS MILEAGE PROGRAM
20 ? "ENTER PREVIOUS ODOMETER READING"
25 INPUT M1
30 ? "ENTER NEW ODOMETER READING"
35 INPUT M2
37 IF M1>M2 THEN ? "READING TOO SMALL":GOTO 20
40 ? "ENTER GALLONS SINCE LAST FILLUP"
45 INPUT G
50 MPG=(M2-M1)/G
55 MPG=INT(MPG*10+0.5)/10
60 ? "GAS MILEAGE: ";MPG;" MPG"
```

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

```
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 MILEAGE: 19.6 MPG

READY
```

Circle Program

In the circle program shown in Figure 4.3, the radius should obviously be positive. Actually, it you only want to calculate the area of the circle given by πr^2, a negative radius will give the same answer as the same positive radius. On the other hand, if you also calculate the circumference of the circle given by $2\pi r$, the radius must be positive. We can calculate the circumference by adding the two statements

```
45 C=2*PI*R
47 PRINT "CIRCUMFERENCE=";C
```

to the program in Figure 4.3. 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 in 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 4.3 that if the radius is too large an overflow error will occur when the area is computed in line 35. Since the value of the area A cannot be greater than 1.E97, the largest radius R that will not result in an overflow can be found as follows:

$$A = \pi r^2 < 1.E97$$
$$r^2 < 1.E97/\pi$$
$$r < \sqrt{1.E97/\pi}$$

Thus, if

$$R > SQR(1.E97/PI)$$

the area will be greater than 1.E97 and cause an overflow. We can test this by adding the following statement to the program:

```
33 IF R>SQR(1.E97/PI) 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* (in other words, if R is positive), the next IF . . . THEN statement on line 33 is executed. If R is not greater than SQR(1.E97/PI), then the program will continue on line 35.

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

```
10 REM PROGRAM TO COMPUTE THE AREA OF A CIRCLE
15 PI=3.14159265
20 ? "ENTER A RADIUS ";
30 INPUT R
32 IF R<0 THEN ? "RADIUS MUST BE POSITIVE":GOTO 20
33 IF R>SQR(1.0E+97/PI) THEN ? "RADIUS TOO LARGE":GOTO 20
35 A=PI*R^2
40 ? "THE AREA OF THE CIRCLE IS ";A
45 C=2*PI*R
47 ? "CIRCUMFERENCE=";C
50 ?
60 GOTO 20
```

FIGURE 6.4 Sample run of program in Figure 6.3.

```
RUN
ENTER A RADIUS ?-3
RADIUS MUST BE POSITIVE
ENTER A RADIUS ?2.5E51
RADIUS TOO LARGE
ENTER A RADIUS ?5.6
THE AREA OF THE CIRCLE IS 98.52034424
CIRCUMFERENCE=35.18583768

ENTER A RADIUS ?
```

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

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

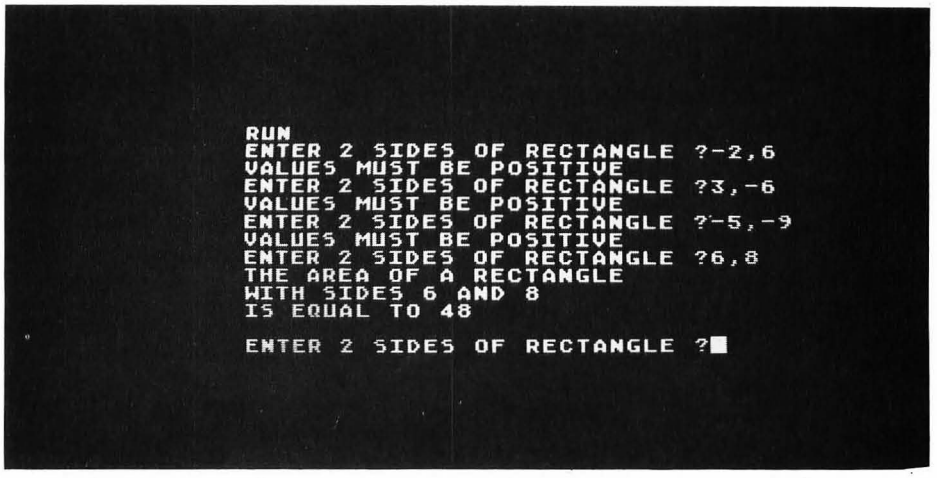


FIGURE 6.6 Sample run of program in Figure 6.5.

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 4.2 that computes the area of a rectangle. It is clear that *both* sides of a rectangle must be positive. Thus, if *either* of the two values entered in the INPUT statement on line 30 is negative, the program should print an error message and ask for new inputs. We can do this by adding the following single 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 ATARI 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 < 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 following two sections.

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.) Thus, for example,

$$A < 0$$

is a logical expression (it is either true or false) formed using the relational operator < ("less than"). If the contents of memory cell A are less than 0, this logical expression is true; otherwise, it is false.

The ATARI stores the logical value "false" as 0. It

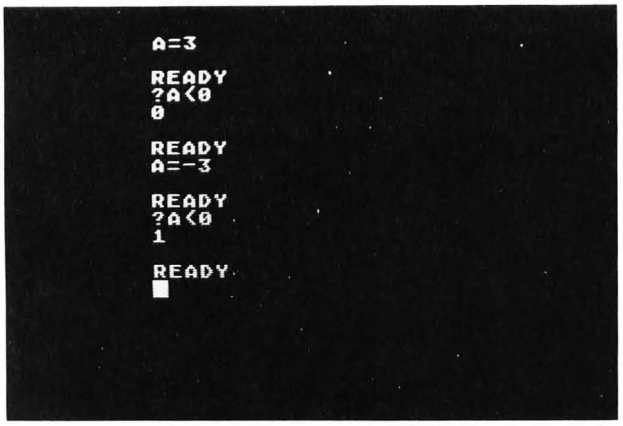


FIGURE 6.7 The ATARI stores "true" as 1 and "false" as 0.

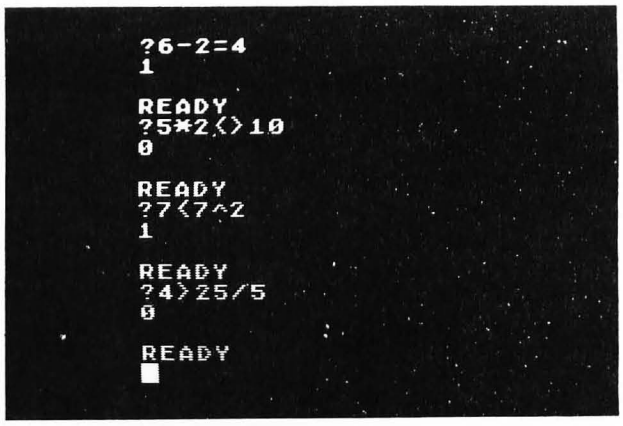


FIGURE 6.8 Examples of logical expressions formed using the relational operators.

stores the logical value "true" as 1. You can see this by typing

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

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.

The relational expressions used in the ATARI 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

LOGICAL OPERATORS

In addition to the relational operators =, <>, <, >, < =, and > =, the ATARI 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

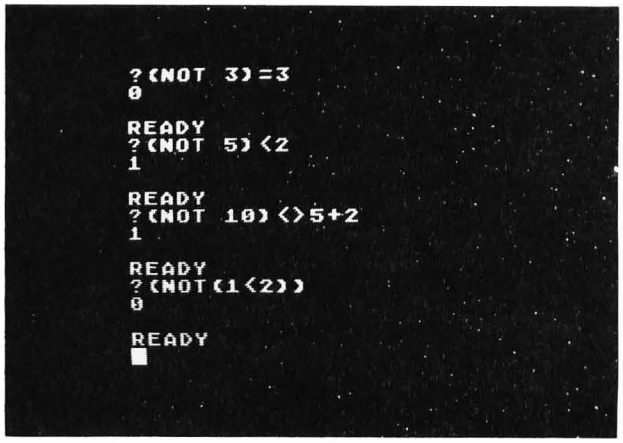
A	*NOT A*
true	false
false	true

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. Because of certain peculiarities in ATARI BASIC it is always a good idea to include a NOT operation in parentheses. Under certain rare conditions, your program may otherwise bomb out.

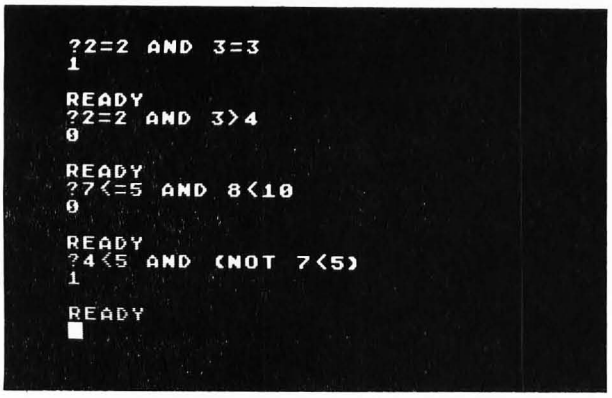
FIGURE 6.9 Using the logical operator NOT.



AND

The logical operator AND is a *binary* operator that operates on *two* logical expressions. Note from Table 6.2 that 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 6.10 Using the logical operator AND.



OR

The logical operator OR is, like AND, a binary operator. Note from Table 6.2 that 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 reason the fourth example is true is that the AND operation is performed *before* the OR operation. There is thus an order of precedence for logical and relational operators as well as arithmetic operators (see Chapter 3). When the ATARI 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.

FIGURE 6.11 Using the logical operator OR.

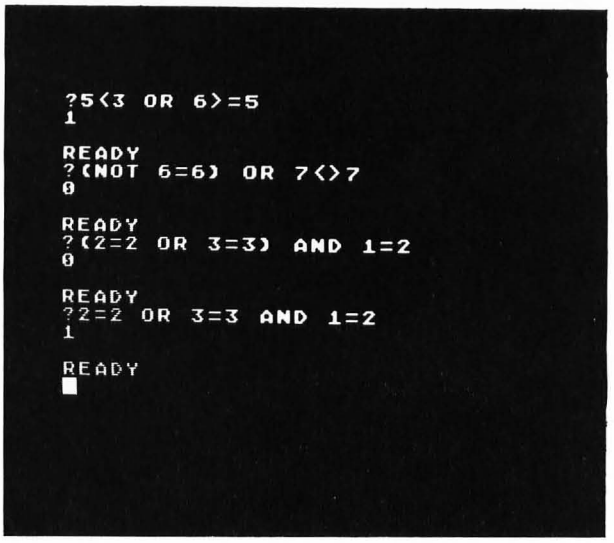


TABLE 6.3 Order of Precedence for Evaluating Expressions

Operator	*Meaning*
( )	Parenthesis
=, < >, <, >, < =, > =	Relational operators used with string variables
−, +	Unary negative or positive
^	Exponentiation
*, /	Multiplication and division
+, −	Addition and substraction
=, < >, <, >, < =, > =	Relational operators used with arithmetic expressions
NOT	Logical complement
AND	Logical AND
OR	Logical OR

WEEKLY PAY PROGRAM

As another example of the IF . . . THEN statement, consider the problem of calculating the weekly pay of an employee whose hourly rate is $4.00 per hour and who receives time and a half for overtime. Suppose that the total hours worked per week cannot exceed 60 hours. Thus, 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 60
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 40 hours and at $6.00 per hour for any hours over 40
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 INPUT; the value is stored in H. Line 40 checks to make

```
10 REM PROGRAM TO COMPUTE WEEKLY WAGES
20 ? "ENTER NUMBER OF HOURS WORKED"
30 INPUT H
40 IF H>60 THEN ? "TOO MANY HOURS":GOTO 20
50 IF H<0 THEN ? "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 ? "WEEKLY PAY= $ ";M
```

FIGURE 6.12 Listing of weekly pay program.

sure that H is not greater than 60. Line 50 checks to make sure that H 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 will branch to statement 90. Line 90 rounds the value of M to two places after the decimal point. 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 (to be paid at $6.00 per hour). Line 80 computes the total pay, M, consisting of the first 40 hours at $4.00 per hour plus the remaining overtime hours at $6.00 per hour. That is, M=40*4+OV*6. Line 90 and 100 will then round and print the total pay.

Sample runs of this program are shown in Figure 6.13. Note that trailing 0s are not printed on the screen. Thus, for example, $233.50 is printed at $233.5. In a later chapter (Chapter 12), we will see how to make the total cents always appear on the screen.

FIGURE 6.13 Sample runs of program in Figure 6.12.

```
RUN
ENTER NUMBER OF HOURS WORKED
?32
WEEKLY PAY= $ 128

READY
RUN
ENTER NUMBER OF HOURS WORKED
?52.25
WEEKLY PAY= $ 233.5

READY
RUN
ENTER NUMBER OF HOURS WORKED
?47.34
WEEKLY PAY= $ 204.04

READY
```

AREA OF 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

$$S = (A+B+C)/2$$

is the semiperimeter.

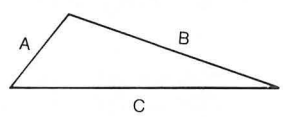
In BASIC the formula for the area can be written as

```
AREA=(S*(S-A)*(S-B)*(S-C))/\0.5
```

or

```
AREA=SQR(S*(S-A)*(S-B)*(S-C))
```

FIGURE 6.14 Finding the area of a triangle.



Semiperimeter, $S = (A + B + C)/2$

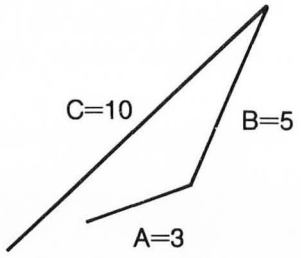
Area $= [S(S - A)(S - B)(S - C)]^{0.5}$

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

We want to write a program that will ask the user to enter the three sides of the 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 hav-

ing 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 the two sides A + B must be greater than C, where C is the longest side. This is equivalent to requiring C to be less than the semiperimeter S = (A + B + C)/2. Note that 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 and C < S = (A + B + C)/2.



Therefore, our program should check to make sure that the three numbers entered from the keyboard can really represent the sides of a triangle. Thus, we need to check to make sure that C < S. But which side is C? It is the longest side. But 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 semiperimeter S.

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

You should convince yourself that this *algorithm,* or step-by-step procedure, will, in fact, result in the memory cell L containing the largest value. This value of L can then be compared to the semiperimeter S to see if a triangle is possible.

The BASIC program to do all this is shown in Figure 6.16. Line 20 asks for the three sides of the traingle to be entered and line 30 stores these three values in A, B, and C. Line 40 compares A and B; if A is greater than 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; if C is greater than L, it stores the value of C in L. Therefore, by the time that line 70 is executed, L will contain the largest number stored in A, B, and C.

Line 70 computes the semiperimeter S, and line 80 compares L and S to see if a triangle is possible. If L is greater than S, the message NO TRIANGLE POSSIBLE is printed and the program branches back to line 20 and asks for three new sides. On the other hand, if L is not greater than S line 90 is executed, which computes the area of the triangle. Line 100 prints the result. Line 110 skips a line and line 120 branches back to line 20 to run the program again. A sample run of this program is shown in Figure 6.17.

FIGURE 6.16 Program to find the area of a triangle.

```
10 REM PROGRAM TO FIND THE
15 REM AREA OF A TRIANGLE
20 ? "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 ? "NO TRIANGLE POSSIBLE":GOTO 20
90 AREA=(S*(S-A)*(S-B)*(S-C))^0.5
100 ? "THE AREA OF THE TRIANGLE IS ";AREA
110 ?
120 GOTO 20
```

```
RUN
ENTER THE THREE SIDES OF A TRIANGLE
?5,10,3
NO TRIANGLE POSSIBLE
ENTER THE THREE SIDES OF A TRIANGLE
?10,3,5
NO TRIANGLE POSSIBLE
ENTER THE THREE SIDES OF A TRIANGLE
?3,4,5
THE AREA OF THE TRIANGLE IS 5.99999988

ENTER THE THREE SIDES OF A TRIANGLE
?
```

FIGURE 6.17 Sample runs of the program in Figure 6.16.

FLOWCHARTS AND PSEUDOCODE

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

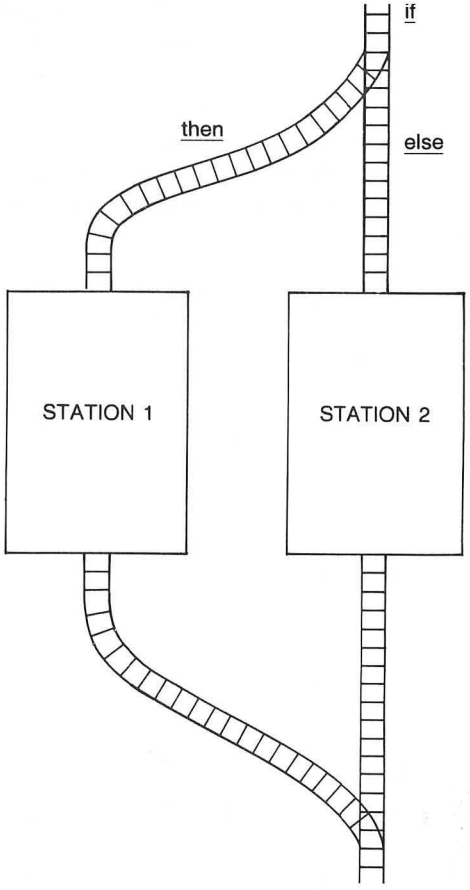
In Chapter 8 we will use the BASIC IF . . . THEN statement to form various loops. The *if . . . then . . . else* statement is one of these "good" statements that is available in structured programming languages such as PASCAL.

In Chapter 2 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 (like the name of the person sitting in the seat) or numerical values (like 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.

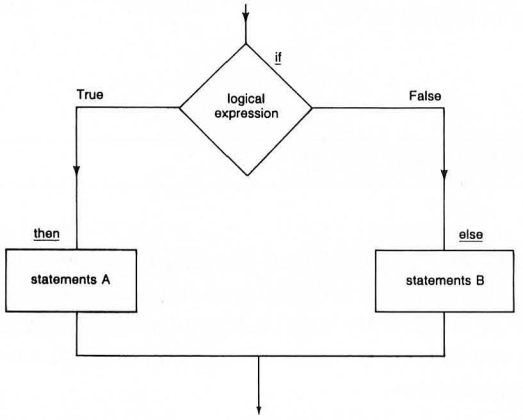
The *if . . . then . . . else* statement is like a *switch* in the track that allows the train to go on one of two different paths, as shown in Figure 6.18. These two paths lead to two different stations and then recombine on the other side of the stations. If the logical expression following *if* is true, the train will follow the track to station 1 where the *then* statements will be executed. If the logical expression following *if* is false, the train will follow the track to station 2 where the *else* statements will be executed. Note that the train can only go to station 1 *or* station 2. It *cannot* go to both stations.

FIGURE 6.18 The *if . . . then . . . else* statement takes the train to one of two possible stations.



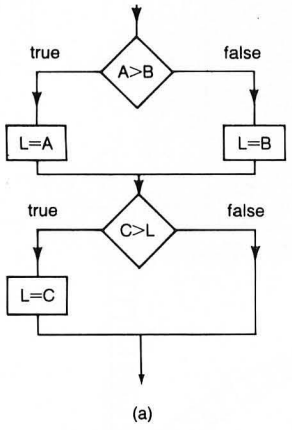
Flowcharts have traditionally been used to express a computer algorithm. The *if . . . then . . . else* statement illustrated in Figure 6.18 can be represented as a flowchart, as shown in Figure 6.19. The similarity to Figure 6.18 is obvious. If the logical expression in the diamond-shaped box is *true,* then the path to statements A will be followed. Otherwise, the path to statements B will be followed.

FIGURE 6.19 Flowchart representation of the *if . . . then . . . else* statement.



The algorithm for finding the largest value in A, B, and C is expressed as a flowchart and in *pseudocode* (that is, using *if . . . then . . . else*) in Figure 6.20. Many people find the pseudocode representation shown in Figure 6.20b to be simpler and just as easy to understand as the flowchart shown in Figure 6.20a. 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.

FIGURE 6.20 (a) Flowchart and (b) pseudocode for algorithm to find the largest value in A, B, and C.



(a)

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

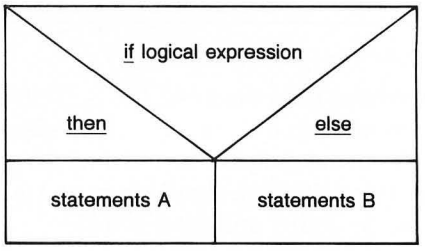
(b)

For those who still like to have some type of graphic representation of an algorithm without creating a "bowl of spaghetti" that is hard to understand, *structured flowcharts* are available.

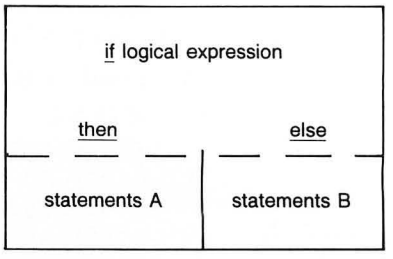
Structured Flowcharts

A structured flowchart, also called a *Nassi-Schneiderman chart,* after the people who introduced it, is an alternate representation of an algorithm that consists of various nested "boxes" without the connecting lines that are shown in Figure 6.20. Two alternate representations of the *if . . . then . . . else* statement are shown in Figure 6.21. We will use the form shown in Figure 6.21b. Using this structured flowchart, we can represent the algorithm given in Figure 6.20 as shown in Figure 6.22.

FIGURE 6.21 Two forms of a structured flowchart that represents the *if . . . then . . . else* statement.

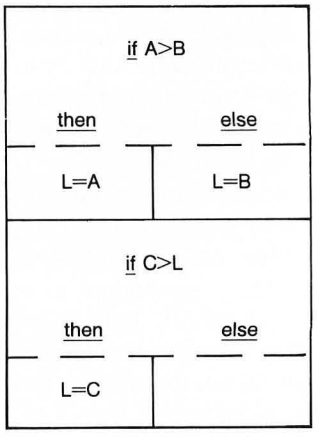


(a)

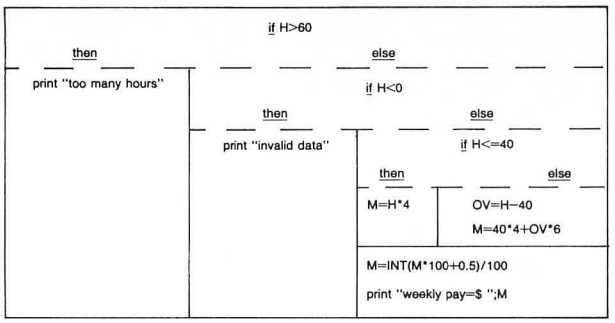


(b)

FIGURE 6.22 Structure flowchart representation of algorithm to find the largest value in A, B, and C.



Flowcharts and pseudocode are just different ways of representing an algorithm to try to make it easier to understand. When you are first developing a computer program it is generally easier to express the pro-



(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 ? "ENTER NUMBER OF HOURS WORKED"
30 INPUT H
40 IF H>60 THEN ? "TOO MANY HOURS":GOTO 20
50 IF H<0 THEN ? "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 ? "WEEKLY PAY= $ ";M
```

FIGURE 6.23 (a) Structured flowchart; (b) pseudocode of weekly pay program; (c) BASIC listing of weekly pay program.

gram in the form of a flowchart, structured flowchart, or pseudocode, and then to convert this algorithm to BASIC.

The structured flowchart and pseudocode for the weekly pay program discussed earlier in this chapter are shown in Figure 6.23a and 6.23b. The BASIC listing of this program is shown in Figure 6.23c. You should carefully compare these three representations of the same program.

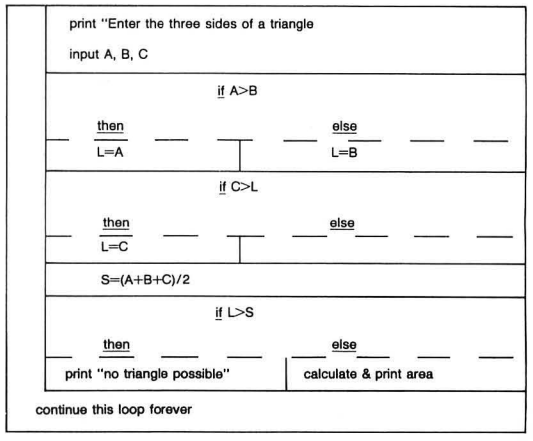
The advantage of the structured flowchart representation is that it clearly displays the logic of the program in a graphic form. The advantage of the pseudocode is that it describes the algorithm in a simple and straightforward manner. Note the importance of the indentation in the pseudocode description. The advantage of the BASIC representation is that it can be executed on the ATARI.

Some people have devised a variety of indentation conventions that will make a BASIC program easier to understand. You can use indentation in your program on the ATARI at the expense of using up more memory. You should always keep a written version of your programs on a piece of paper. This version can include indentation, pseudocode, structured flowcharts, or anything else that will help you to understand the program.

The complete structured flowchart for the program to find the area of a triangle is shown in Figure 6.24a. The BASIC listing of this program is shown in Figure 6.24b. You should compare the structured flowchart carefully with 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 by pressing the BREAK key).

In Chapter 8 we will take a closer look at loops. In particular you will learn how to stop a loop any time you want.

FIGURE 6.24 (a) Structured flowchart and (b) BASIC listing of program to find the area of a triangle.



```
10 REM PROGRAM TO FIND THE
15 REM AREA OF A TRIANGLE
20 ? "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 ? "NO TRIANGLE POSSIBLE":GOTO 20
90 AREA=(S*(S-A)*(S-B)*(S-C))^0.5
100 ? "THE AREA OF THE TRIANGLE IS ";AREA
110 ?
120 GOTO 20
```

FIGURE 6.4 (cont.)

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 percent of the amount over $20,200. Write a program that will input a taxable income, check that it is between $20,200 and $24,600, and then compute and print the income tax on the screen.

EXERCISE 6.2

Write a program to compute take-home pay. The program should input an hourly wage and the number of hours worked. Assume that 6.65 percent of the gross pay is deducted for Social Security taxes, 14.8 percent of the gross pay is deducted for federal income taxes, and 4 percent 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 deducted for Social Security, federal, and state income taxes, and the take-home pay.

EXERCISE 6.3

Write a program that will continuously input a 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.

7

LEARNING TO USE LOW-RESOLUTION GRAPHICS—DISPLAYING THE FLAG

In this chapter you will learn how to draw colored pictures on the screen using one of the low-resolution graphics modes of the ATARI. A high-resolution graphics mode that is also available will be described in Chapter 13.

In this chapter you will learn

1. how to plot various colored dots by using the statement PLOT X,Y

2. how to plot lines using the statement DRAWTO X,Y

3. to draw dashed lines using the FOR . . . NEXT loop

4. to draw areas and arrays of points

5. how to display the American flag on the TV screen.

PLOTTING DOTS AND LINES USING THE PLOT AND DRAWTO STATEMENTS

Type

GR. 5

This is an abbreviation for GRAPHICS 5, which sets the graphics mode 5. When you do this the screen will clear to black except for a four-line text window at the bottom of the screen.

Graphics mode 5 is a low-resolution graphics mode in which the screen is considered to be divided into a grid made up of 40 rows and 80 columns, as shown in Figure 7.1. The column positions of the grid are numbered 0 through 79 from left to right. This is called the X position or X coordinate. The row positions of the grid are numbered 0 through 39 from top to bottom. This is called the Y position or Y coordinate. Any one of 3,200 (40 × 80 = 3,200) small squares or blocks on the grid can be identified by giving its X and Y coordinates. For example, in Figure 7.1 the shaded block is located at the coordinates X = 25,Y = 15.

You can plot a colored spot at any of the 3,200 grid positions on the screen. These spots can be one of 16 different color hues. The possible color hues are given in Figure 7.2. In addition to the 16 color hues a

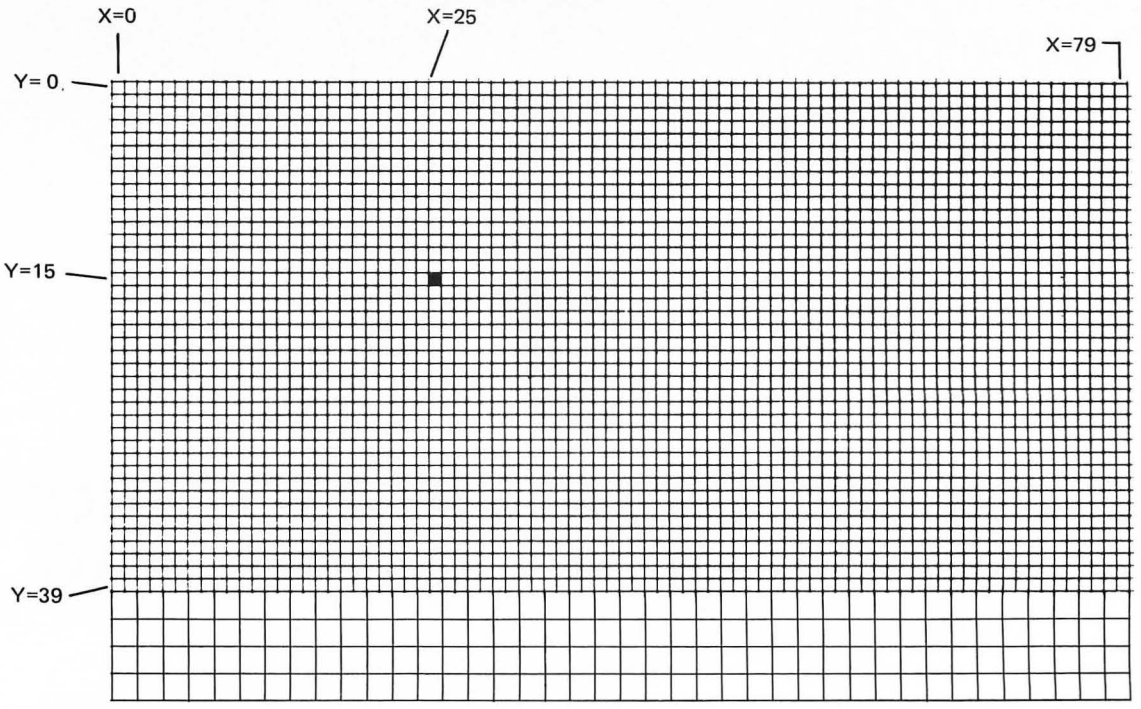
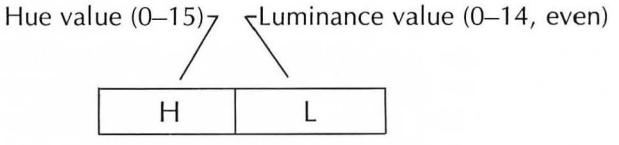


FIGURE 7.1 The low-resolution graphics mode 5 divides the screen into an 80×40 grid.

0 Gray	8 Light blue
1 Light orange(gold)	9 Dark blue
2 Orange	10 Turquoise
3 Red-orange	11 Green-blue
4 Pink	12 Green
5 Purple-blue	13 Yellow-green
6 Blue	14 Orange-green
7 Blue	15 Light orange

FIGURE 7.2 Sixteen colors numbered 0–15 can be plotted using low-resolution graphics.



Color register No. 0

FIGURE 7.3 A color register contains the hue and luminance values for a particular color.

spot can have one of 8 different values of color luminance. The luminance value is an even number between 0 and 14. A luminance value of 0 is very faint; a value of 14 is very bright. A color is determined by the combination of its hue value (0–15) and its luminance value (0–14). An odd luminance value has the same effect as the next lower even value.

The 16 hues and 8 luminances give rise to 128 (16 × 8) different colors that can be displayed. To display a particular color, the hue and luminance value must be stored in a *color register*. The color register contains both the hue value and the luminance value, as shown in Figure 7.3.

The ATARI contains five color registers that contain different colors (hue–luminance combinations). These resigers are numbered 0–4 as shown in Figure 7.4. The statement

SETCOLOR R,H,L

will store the color hue value H and the luminance L in color register R. For example,

SETCOLOR 0,0,14

FIGURE 7.4 SETCOLOR R,H,L stores the color hue H and luminance L in color register R.

SETCOLOR R,H,L *Register No.*		*COLOR Numbers (mode 5)*
0	H L	1
1	H L	2
2	H L	3
3	H L	—
4	H L	0 (Background and border)

will set the color in register 0 to gray with a luminance of 14.

To plot a spot on the screen we must tell the ATARI which color register to use to determine the color. Graphics mode 5 allows four different colors corresponding to the color registers 0, 1, 2, and 4. Color register 4 controls the color of the background. The default value of color register 4 is black. (SETCOLOR 4,0,0). The statement

```
COLOR N
```

tells the ATARI which color register to use. For graphics mode 5, the value of N must be 1, 2, 3, or 0 corresponding to color registers 0, 1, 2, and 4, as shown in Figure 7.4. Therefore, if you want to use the color stored in color register 0, you must execute the statement

```
COLOR 1
```

Now type

```
SETCOLOR 0,0,14:COLOR 1
```

This will set color register 0 to white and identify color register 0 (COLOR 1) as the color to use.

The text window at the bottom of the screen is controlled by the GR. 0 mode. In this mode color register 2 contains the color of the background and color register 1 controls the color of the text. The hue value in color register 1 (the text color) is always the same as the background (color register 2). Only the luminance can be different. Type

```
SETCOLOR 2,0,0
```

This will set the background of the text window to black (gray hue with 0 luminance). Now type

```
SETCOLOR 1,0,14
```

This will set the text color in the text window to white (the same gray hue as the background but with maximum luminance).

Once you have set the color, you can plot a spot located at coordinates X,Y by typing

```
PLOT X,Y
```

For example, if after typing

```
GR. 5
SETCOLOR 0,0,8:COLOR 1
```

you type

```
PLOT 25,15
```

then a white spot located at coordinates X = 25, Y = 15 will be plotted, as shown in Figure 7.5.

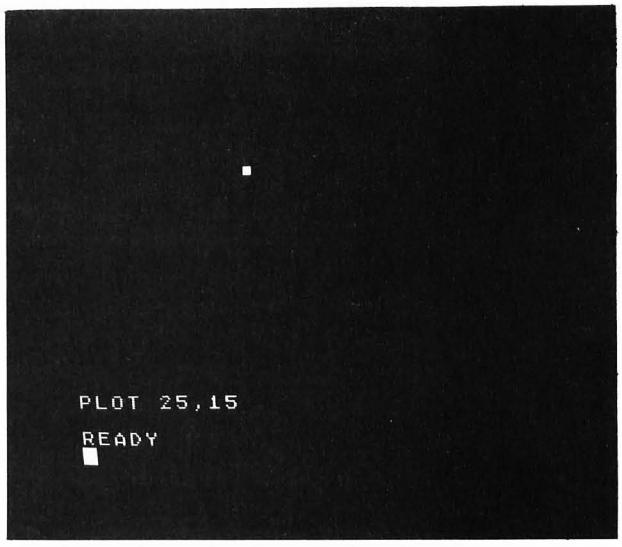


FIGURE 7.5 PLOT 25,15 will plot a spot at location X = 25, Y = 15.

In order to get out of the low-resolution graphics mode 5, type

```
GR. 0
```

This will cause the ATARI to return to the full-screen text mode (24 lines of 40 characters each), which is the same as graphics mode 0.

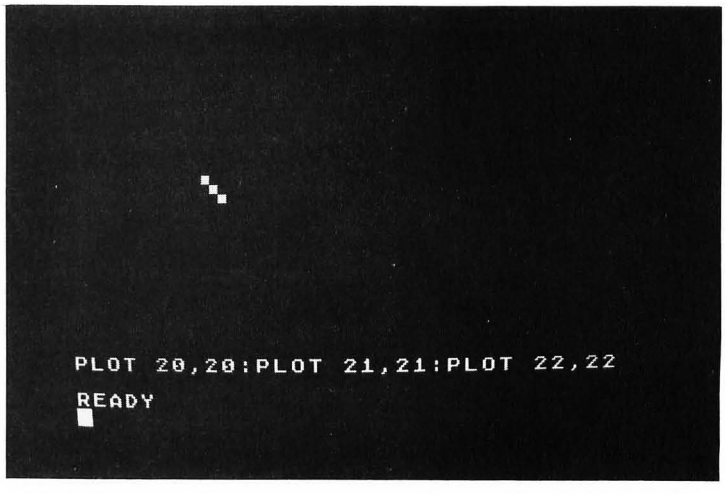
Whenever a GRAPHICS statement is executed the color registers are set to their default values. These default values are shown in Table 14.4 in Chapter 14.

Return to the low-resolution graphics mode 5 by typing GR. 5 again. Now type

```
SETCOLOR 0,0,14:COLOR 1
SETCOLOR 2,0,0:SETCOLOR 1,0,14
PLOT 20,20:PLOT 21,21:PLOT 22,22
```

The screen should display three spots located along a diagonal line, as shown in Figure 7.6.

FIGURE 7.6 Multiple spots can be plotted using multiple PLOT statements.



Now type

```
DRAWTO 24,20
```

Notice that the two spots at 23,21 and 24,20 are plotted as shown in Figure 7.7. The statement DRAWTO X,Y can be used to plot a line from the most recently plotted spot to the location X,Y.

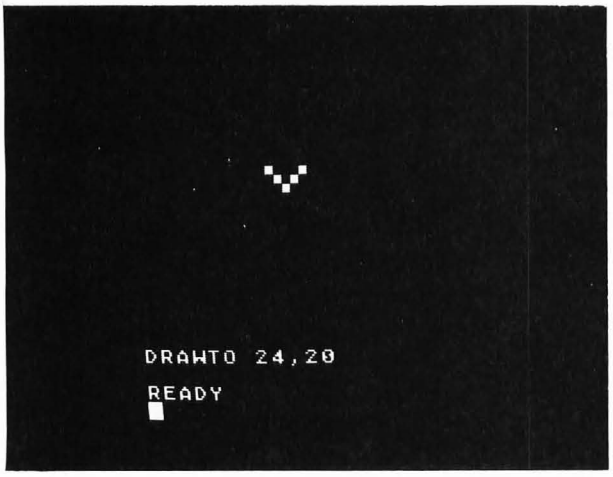


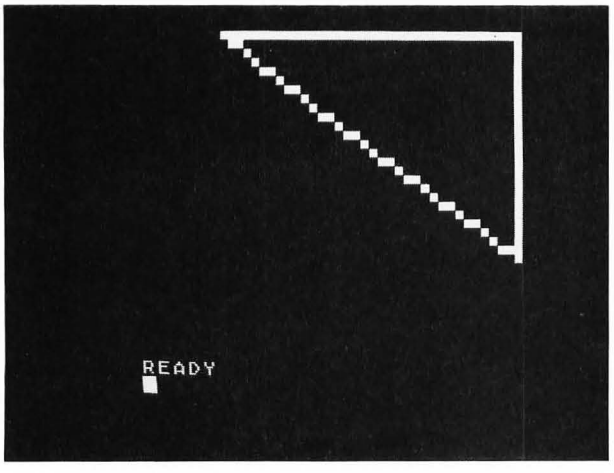
FIGURE 7.7 Plotting a line using DRAWTO X,Y.

The graphics commands can be used in the deferred mode of execution by including them in a BASIC program. For example, return to the text mode by typing GR. 0 and then type in the following program:

```
10 GRAPHICS 5
15 SETCOLOR 4,0,0
20 SETCOLOR 0,0,14:COLOR 1
25 SETCOLOR 2,0,0:SETCOLOR 1,0,14
30 PLOT 15,5:DRAWTO 50,5
40 DRAWTO 50,30
50 DRAWTO 15,5
```

This program should plot the triangle shown in Figure 7.8.

FIGURE 7.8 Triangle plotted using DRAWTO statements.



EXERCISE 7.1

Plot the following horizontal lines on the screen:

1. a blue line from X = 10 to X = 35 at Y = 3
2. a yellow line eight spots long starting at column number 10 on row number 12
3. a pink line all the way across the top of the screen.

EXERCISE 7.2

Plot the following vertical lines on the screen:

1. a green line from Y = 3 to Y = 15 at X = 2
2. a purple line 15 blocks high with the top at row 10 and located in column 18
3. a blue line along the entire right edge of the screen.

Drawing Your Name

Suppose that you want to draw your name in large block letters on the screen. The first step is to draw your name on quadrille paper the way you want it to appear on the 80×40 grid on the screen. For example, Figure 7.9 shows the name JEFF sketched on a grid. Some of the column and row numbers are written next to each letter.

From Figure 7.9 you can see that to plot the letter J the computer must execute the statements

```
PLOT 2,19
PLOT 2,20:DRAWTO 8,20
DRAWTO 8,10
```

Similarly, to plot the letter E the statements

```
PLOT 11,10:DRAWTO 11,20
DRAWTO 17,20
PLOT 12,15:DRAWTO 15,15
PLOT 12,10:DRAWTO 17,10
```

must be executed. The statements

```
PLOT 20,20:DRAWTO 20,10
DRAWTO 26,10
PLOT 21,14:DRAWTO 24,14
```

will plot the first F; the second F can be plotted with the statements

```
PLOT 29,20:DRAWTO 29,10
DRAWTO 35,10
PLOT 30,14:DRAWTO 33,14
```

You can type these statements in the immediate mode and watch each letter being plotted one seg-

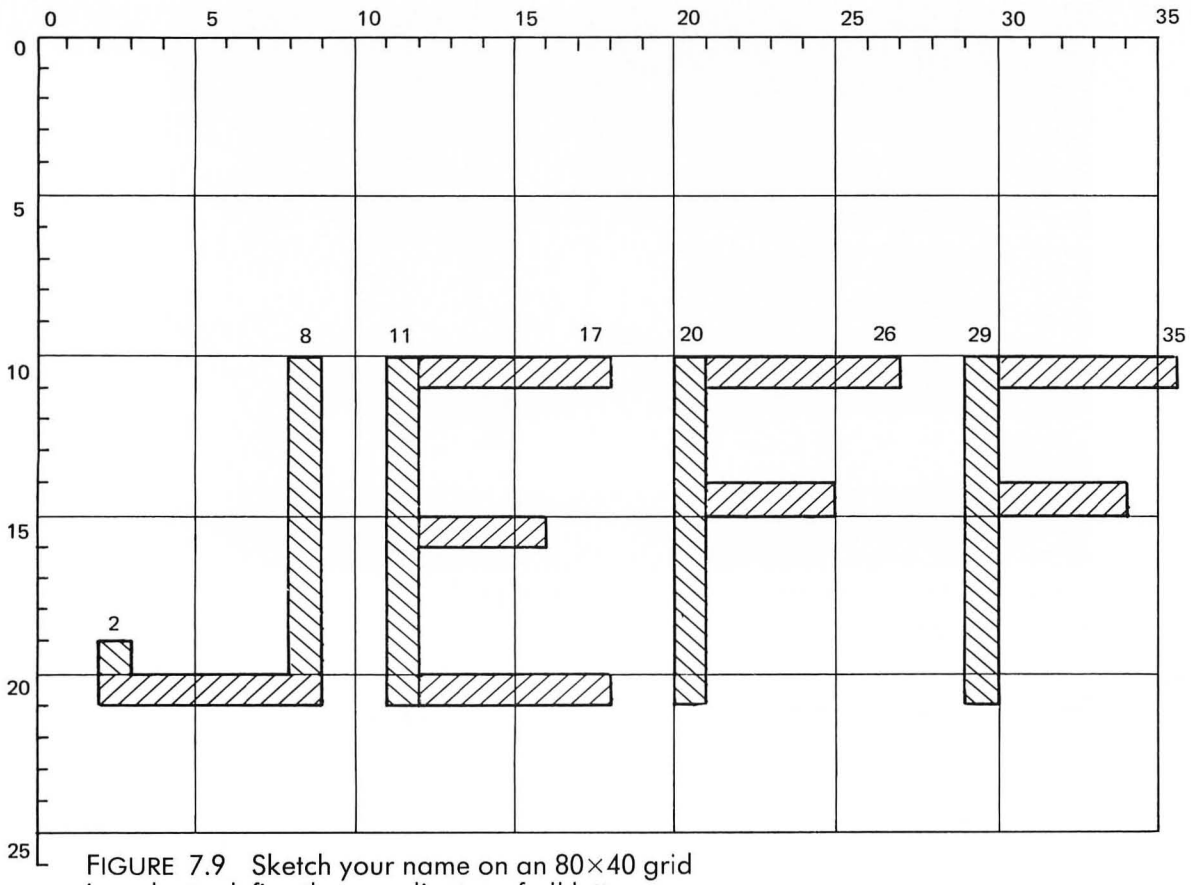


FIGURE 7.9 Sketch your name on an 80×40 grid in order to define the coordinates of all letter segments.

ment at a time. Alternatively, you can return to the GR. 0 mode and type in the entire program using line numbers. Then you can execute the program by typing RUN.

A listing of this program, to be run in the deferred mode, is shown in Figure 7.10. Line 20 enters the low-resolution graphics mode 5. Each different letter is plotted in a different color. Lines 30–60 plot a green J. Lines 70–110 plot a yellow E. Lines 120–150 plot a pink F, and lines 170–190 plot another pink F. The result of running this program is shown in Figure 7.11.

Note that to plot three different colors on the screen we must store three different colors in the three *different* color registers 0, 1, and 2. (Recall that color register 4 stores the color of the background.) This is because each spot that we plot on the screen has associated with it the color number (1–3) that we used when we plotted the spot. Color number 1 always points to color register 0. If you change the color (H and L values) stored in color register 0 the color of the J will change. To see this, type

```
SETCOLOR 0,3,8
```

The J should change from green to red. Similarly, if you type

```
SETCOLOR 2,9,8
```

the two Fs will change from pink to dark blue. Try this. Change the values stored in the various color registers (by executing the SETCOLOR statement) and watch the colors of the four letters and background change.

FIGURE 7.10 Listing of program to plot the name JEFF in block letters.

```
20 GRAPHICS 5
30 SETCOLOR 0,12,8:COLOR 1
40 PLOT 2,19
50 PLOT 2,20:DRAWTO 8,20
60 DRAWTO 8,10
70 SETCOLOR 1,13,8:COLOR 2
80 PLOT 11,10:DRAWTO 11,20
90 DRAWTO 17,20
100 PLOT 12,15:DRAWTO 15,15
110 PLOT 12,10:DRAWTO 17,10
120 SETCOLOR 2,4,8:COLOR 3
130 PLOT 20,20:DRAWTO 20,10
140 DRAWTO 26,10
150 PLOT 21,14:DRAWTO 24,14
170 PLOT 29,20:DRAWTO 29,10
180 DRAWTO 35,10
190 PLOT 30,14:DRAWTO 33,14
```

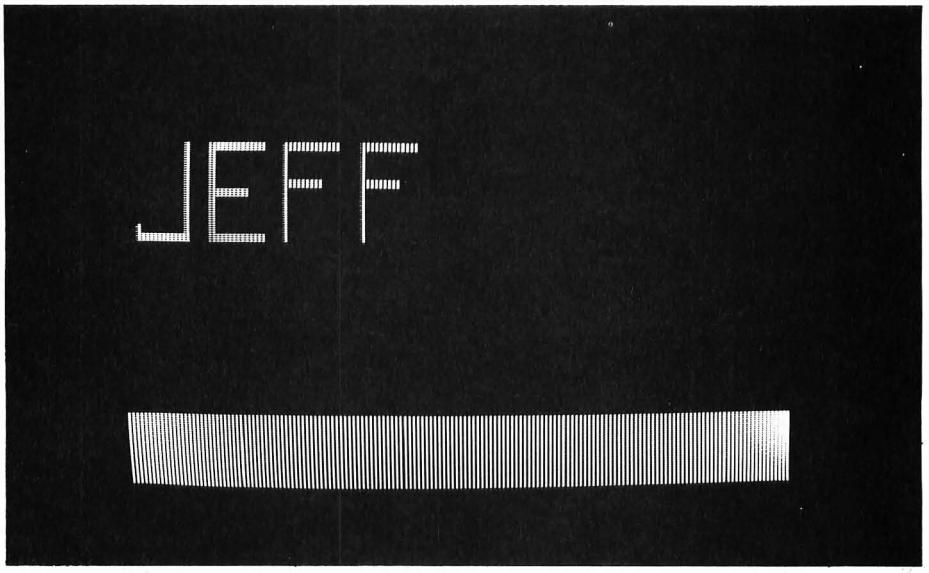


FIGURE 7.11 Result of running the program shown in Figure 7.10.

EXERCISE 7.3
Write a program that will plot your name in block letters on the screen. Use three different colors for the letters.

Drawing Dashed Lines

Enter the low-resolution graphics mode 5 by typing

```
GR. 5
SETCOLOR 0,0,14:COLOR 1
SETCOLOR 2,0,0:SETCOLOR 1,0,14
```

Now type

```
FOR X=12 TO 40 STEP 2:PLOT X,5:NEXTX
```

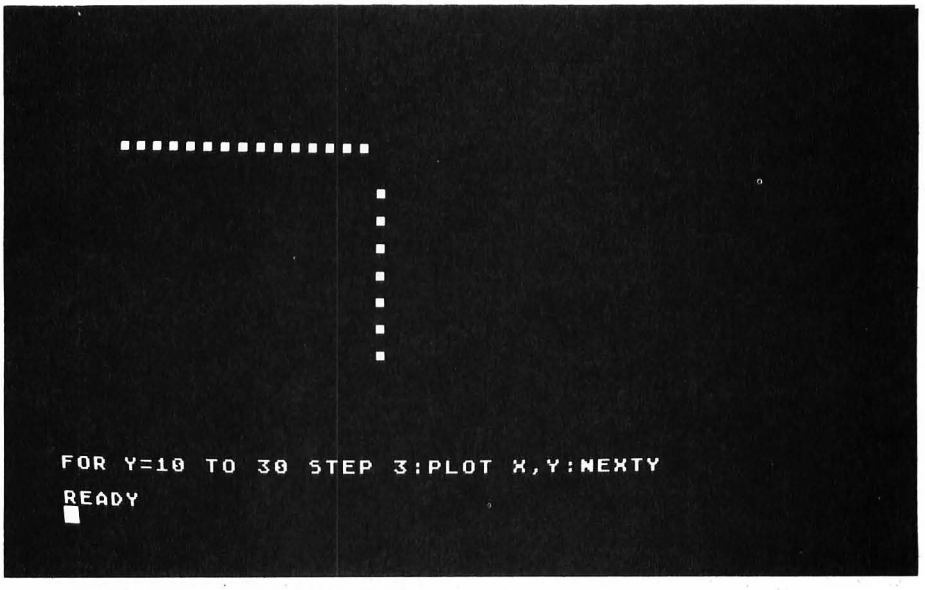
This will plot the horizontal dashed line shown in Figure 7.12.

The vertical line shown in Figure 7.12 can be plotted by typing

```
FOR Y=10 TO 30 STEP 3:PLOT X,Y:NEXTY
```

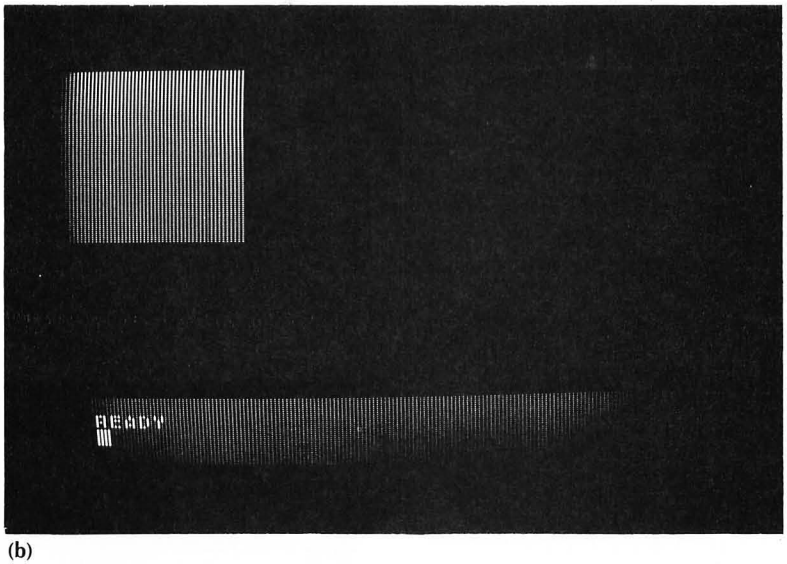
Note that seven spots are plotted in this vertical line corresponding to Y values of 10, 13, 16, 19, 22, 25, and 28. Another step of 3 would produce a value of Y equal to 31, which is greater than 30. Therefore, the FOR . . . NEXT loop is exited.

FIGURE 7.12 Plotting dashed lines using the FOR . . . NEXT loop.



```
10 REM PLOT AREA
20 GRAPHICS 5:SETCOLOR 2,9,8:COLOR 3
30 FOR ROW=0 TO 20
40 PLOT 0,ROW:DRAWTO 24,ROW
50 NEXT ROW
```

(a)



(b)

FIGURE 7.13 Program shown in (a) will plot area shown in (b).

Drawing Areas

The program shown in Figure 7.13a will plot the blue area shown in Figure 7.13b. This area is plotted by drawing 21 rows (0–20) of horizontal lines, each 25 units long.

Type in this program and run it. Modify the program so that it will draw a square area 20 units on a side with the upper-left-hand corner of the square at the coordinates X = 10, Y = 10.

Plotting an Array of Points

Earlier you saw (see Figure 7.12) that in the low-resolution graphics mode the FOR . . . NEXT loop

```
FOR X = 12 TO 40 STEP 2:PLOT X,5:NEXTX
```

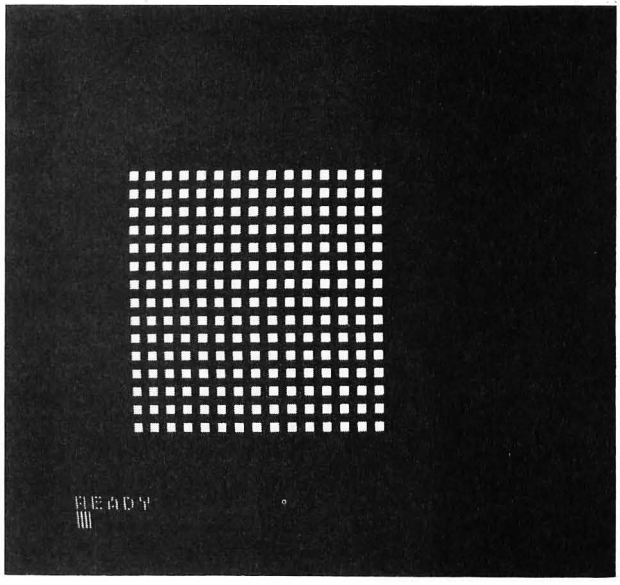
will plot 15 spots in a horizontal row with a blank space between adjacent spots. If you change the statement PLOT X,5 to PLOT X,Y and then let Y change in an outer FOR . . . NEXT loop, you can produce several rows of these dashed lines. The program shown in Figure 7.14 will do this.

Line 20 enters the low-resolution graphics mode 5 and sets the color to white. The inner FOR . . . NEXT loop starting at line 40 produces one row of 15 spots at line number Y. The outer FOR . . . NEXT loop starting at line 30 plots 15 rows of these dashed lines as Y varies from 6 to 34 in steps of 2.

FIGURE 7.14 Program to plot an array of points.

```
10 REM ARRAY OF POINTS
20 GRAPHICS 5:SETCOLOR 0,0,14:COLOR 1
30 FOR Y=6 TO 34 STEP 2
40 FOR X=12 TO 40 STEP 2
50 PLOT X,Y
60 NEXT X:NEXT Y
```

FIGURE 7.15 Array of plots plotted using the program in Figure 7.14.

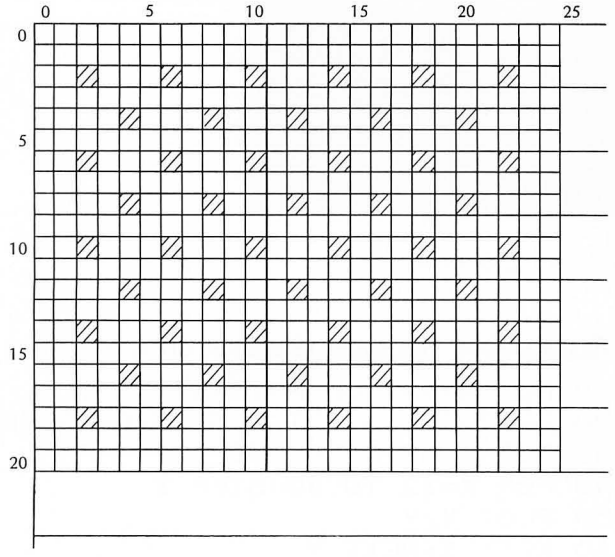


Type in this program and run it. You should obtain the array of spots shown in Figure 7.15. Modify this program by changing the number of rows, the number of points plotted in each row, and the spacing between the spots.

Plotting the Star Field

When we display the flag later in this chapter we will need to plot the star field. We will do this by plotting an array of low-resolution graphic spots. These will be arranged according to the pattern shown in Figure 7.16.

FIGURE 7.16 Pattern used to display the star field in the flag.



If you look carefully at this pattern you will see that it consists of two rectangular arrays of points: a 5×6 array and a 4×5 array. These two rectangular arrays will be plotted separately.

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

```
230 FOR Y=2 TO 18 STEP 4
240 FOR X=2 TO 22 STEP 4
250 PLOT X,Y:NEXT X:NEXT Y
```

The second rectangular array can be plotted using the following statements:

```
260 FOR Y=4 TO 16 STEP 4
270 FOR X=4 TO 20 STEP 4
280 PLOT X,Y:NEXT X:NEXT Y
```

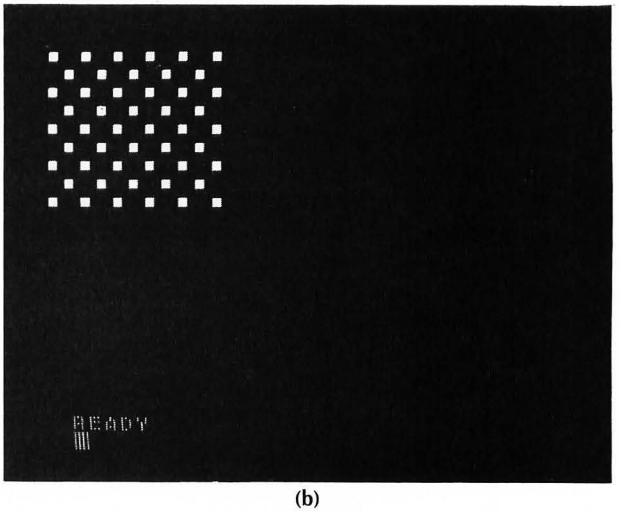
You should convince yourself that these two sets of statements will, in fact, produce the pattern shown in Figure 7.16.

A program that will plot this star field is shown in Figure 7.17a. The result of running this program is shown in Figure 7.17b.

FIGURE 7.17 (a) BASIC program to display star field; (b) star field displayed by executing program in (a).

```
215 REM PLOT STAR FIELD
217 GRAPHICS 5:C2=0
220 SETCOLOR 0,C2,8:COLOR 1
230 FOR Y=2 TO 18 STEP 4
240 FOR X=2 TO 22 STEP 4
250 PLOT X,Y:NEXT X:NEXT Y
260 FOR Y=4 TO 16 STEP 4
270 FOR X=4 TO 20 STEP 4
280 PLOT X,Y:NEXT X:NEXT Y
290 END
```

(a)



(b)

Making Stripes

The one further 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 two colors. 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 colors, C1 and C2, from which the stripes will be formed.

Given these variables, Figure 7.18 shows an algorithm that will display N stripes, each of width W and length L, starting with the C1 color in color register 0.

In this algorithm the inner NL *for . . . next* loop will plot one stripe consisting of W rows of lines, each with a length L. The color of the first stripe will be C1

(color number R=1). After the NL *for . . . next* loop is completed the value of R is changed to the other color register number using the *if . . . then . . . else* statement. The outer NS *for . . . next* loop 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 7.19. You should type in this program and run it. A sample run of the program is shown in Figure 7.20. You should try making different kinds of stripes using this program. Another sample run of this program is shown in Figure 7.21. We will use the values shown in this example to help display our flag.

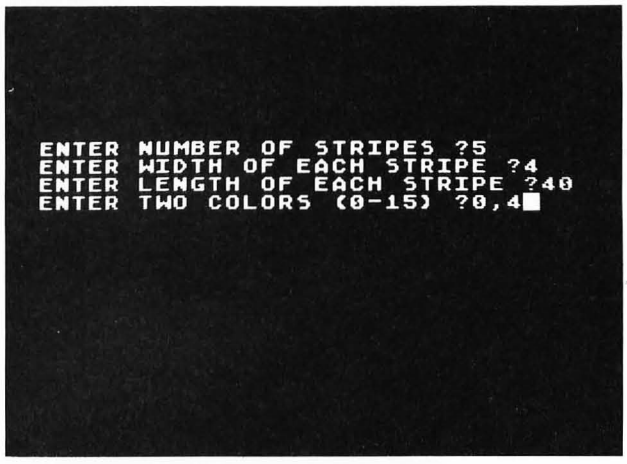
FIGURE 7.18 Algorithm for displaying N stripes, each of width W and length L, starting with the color C1 in color register 0.

```
clear screen
SETCOLOR 0,C1,8: SETCOLOR 1,C2,8
ROW = 0:R = 1
for NS = 1 to N
    COLOR R
    for NL = 1 to W
        PLOT 0,ROW: DRAWTO L-1, ROW
        ROW = ROW + 1
    next NL
    if R = 1
    then R = 2
    else R = 1
next NS
```

FIGURE 7.19 BASIC listing of program to make stripes.

```
10 REM PROGRAM TO MAKE STRIPES
15 ? "}"
20 ? "ENTER NUMBER OF STRIPES ";
25 INPUT N
30 ? "ENTER WIDTH OF EACH STRIPE ";
35 INPUT W
40 ? "ENTER LENGTH OF EACH STRIPE ";
45 INPUT L
50 ? "ENTER TWO COLORS (0-15) ";
55 INPUT C1,C2
57 GRAPHICS 5
60 SETCOLOR 0,C1,8:SETCOLOR 1,C2,8
65 ROW=0:R=1
70 FOR NS=1 TO N
75 COLOR R
80 FOR NL=1 TO W
90 PLOT 0,ROW:DRAWTO L-1,ROW
100 ROW=ROW+1
110 NEXT NL
120 IF R=1 THEN R=2:GOTO 140
130 R=1
140 NEXT NS
```

FIGURE 7.20 Sample run of program shown in Figure 7.19.



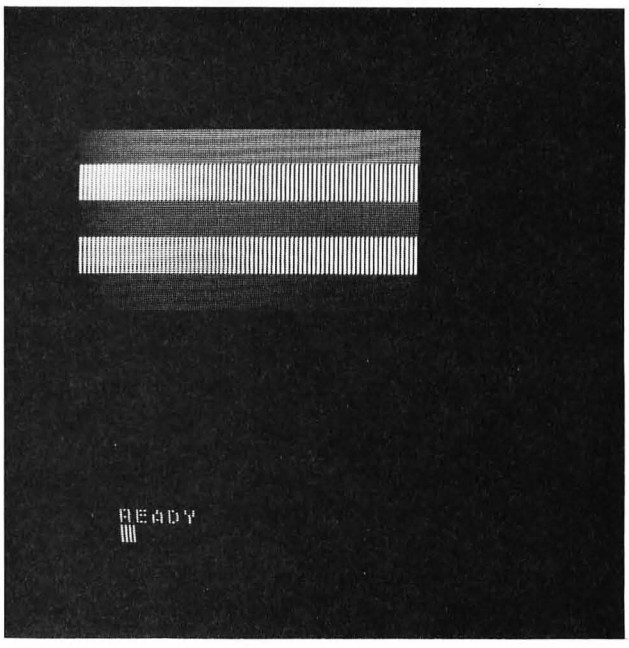
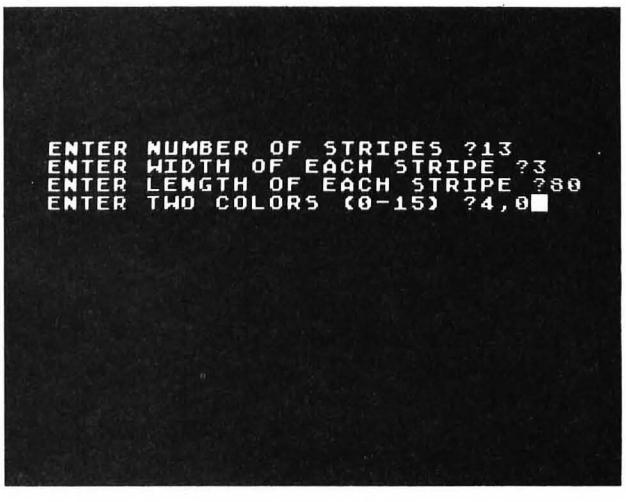


FIGURE 7.21 A second run of program shown in Figure 7.19.



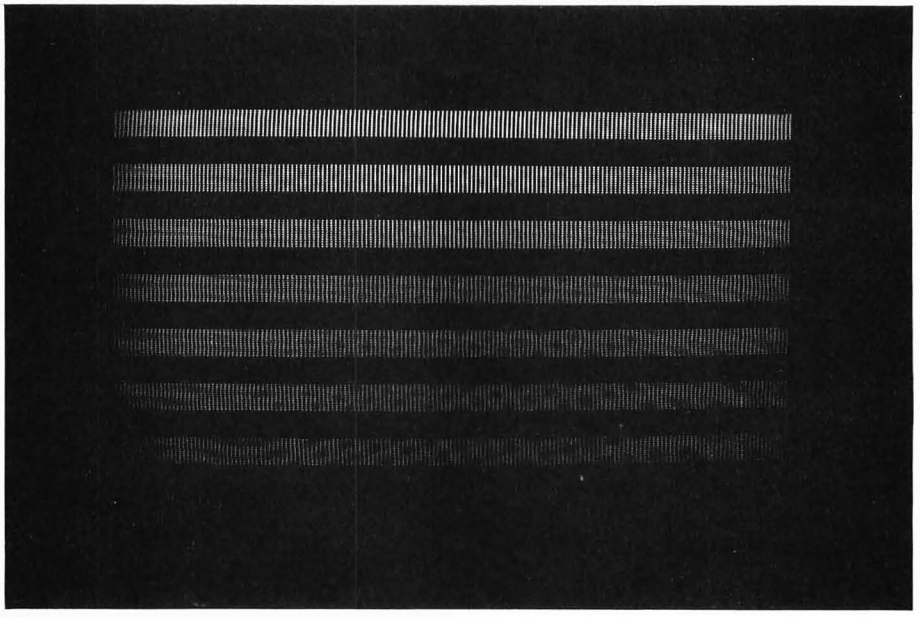


FIGURE 7.21 (cont.)

DISPLAYING THE FLAG

The American flag has 13 stripes. If we use 3 graphics 5 lines for each stripe, we will require 39 lines. Since there are 40 rows in the graphics 5 mode, this will work out well. The star field shown in Figure 7.16 is 21 rows high, which corresponds to the top 7 stripes of the flag. This is the correct size of the star field.

The BASIC program shown in Figure 7.22 will display the 13 stripes of the flag. Lines 50–150 are just the algorithm shown in Figure 7.18 with N = 13, W = 3, and L = 71. This will produce a somewhat shortened version of the 13 stripes shown in Figure 7.21b.

We now need to add the blue field to Figure 7.21b. This will be done by plotting a 21×25 blue area in the upper-left-hand corner of the screen. The following algorithm will do this:

set color to blue
for ROW = 0 to 20
    Plot from 0 to 24 at ROW
next ROW

This algorithm is accomplished by adding lines 170–210 to our program, as shown in Figure 7.23. The result of executing this new program is shown in Figure 7.24.

We now need to add the star field. This is done by adding lines 215–290 as shown in Figure 7.25. (See Figure 7.17a.) The resulting flag is shown in Figure 7.26.

FIGURE 7.22 Program to display the 13 stripes of the flag.

```
10 REM PROGRAM TO DISPLAY FLAG
20 SETCOLOR 0,4,6:REM RED
30 SETCOLOR 1,0,14:REM WHITE
40 SETCOLOR 2,9,6:REM BLUE
50 GRAPHICS 5
60 ROW=0:R=1
70 FOR NS=1 TO 13:REM RED & WHITE STRIPES
80 COLOR R
90 FOR NL=1 TO 3
100 PLOT 0,ROW:DRAWTO 70,ROW
110 ROW=ROW+1
120 NEXT NL
130 IF R=1 THEN R=2:GOTO 150
140 R=1
150 NEXT NS
```

FIGURE 7.23 Subroutine to add the blue field to the flag.

```
170 REM PLOT BLUE FIELD
180 COLOR 3
190 FOR ROW=0 TO 20
200 PLOT 0,ROW:DRAWTO 24,ROW
210 NEXT ROW
```

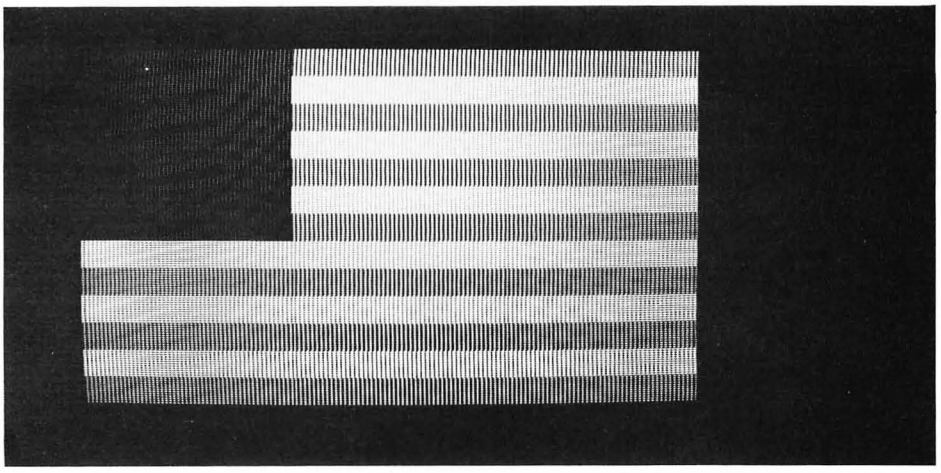


FIGURE 7.24 Result of executing program shown in Figures 7.22 and 7.23.

FIGURE 7.25 Program to add the star field to the flag.

```
215 REM PLOT STAR FIELD
220 COLOR 1:REM WHITE STARS
230 FOR Y=2 TO 18 STEP 4
240 FOR X=2 TO 22 STEP 4
250 PLOT X,Y:NEXT X:NEXT Y
260 FOR Y=4 TO 16 STEP 4
270 FOR X=4 TO 20 STEP 4
280 PLOT X,Y:NEXT X:NEXT Y
290 END
```

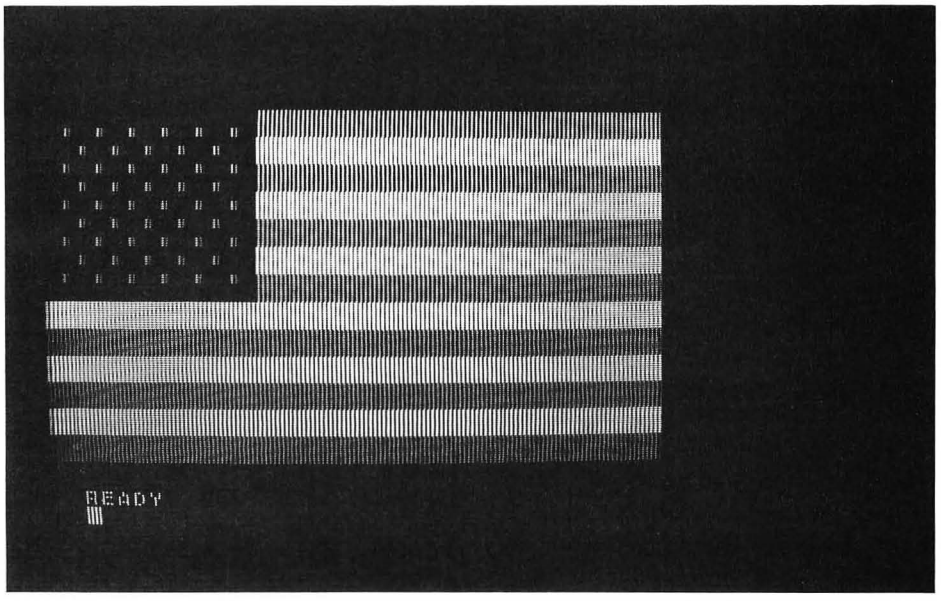
EXERCISE 7.4

Write a program that will draw a large 8×8 red and blue checkerboard on the screen suitable for playing a game of checkers.

EXERCISE 7.5

Write a program that will plot a large 3×3 red and blue checkerboard for playing tic-tac-toe. Each square of the checkerboard should contain 12×12 low-resolution plotting spots. Plot a yellow X on square I,J if you input I,J,X. Plot a green O on square I,J if you input I,J,O. For example, if you input 1,2,X a large yellow X should be plotted in the second square of the first row. If you input 2,2,O, a large green circle should be plotted in the center square. Make the large circle occupy 8×8 squares. Make the large X occupy 7×7 squares.

FIGURE 7.26 Flag produced by program shown in Figures 7.22, 7.23, and 7.25.



8

LEARNING MORE ABOUT LOOPS—ANOTHER LOOK AT IF . . . THEN

In Chapters 5 and 7 we used the FOR and NEXT statements to form loops. In this chapter we will form different types of loops by using the IF . . . THEN statement. In Chapter 6 we used the IF . . . THEN statement to make simple choices 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. Since you are using the same IF . . . THEN statement, you may think that there is no difference between using IF . . . THEN to form loops and using it to form an *if . . . then . . . else* construct. But this is not so. 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 repeat a loop while an affirmative answer is given to a question
2. to use the IF . . . THEN statement to form a *repeat while* loop
3. to make nested loops using the IF . . . THEN statement
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* construct 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:

```
30_____
40_____
50_____
60_____
repeat lines 30–60 while A > 0
```

You can do this with the following statement:

```
70 IF A>0 THEN 30
```

Lines 30–70 form a *loop* that is exited only when A > 0 becomes false—that is, when A < = 0. Obviously, in order to get out of the loop there must be something in lines 30–60 that will eventually cause A to become less than or equal to 0.

Later in this chapter we will look at other types of loops. For now, let's look at some examples.

TRIANGLE PROGRAM

The program to find the area of a triangle was discussed in Chapter 6; the BASIC listing is given in Figure 6.16. Because of the GOTO statement in line 120, this program executes over and over again until the BREAK key is pressed in response to the INPUT statement. 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:

```
17  DIM A$(5)
120 ? "DO YOU WANT TO CONTINUE (Y,N)";
125 INPUT A$
130 IF A$(1,1)="Y" THEN 20
140 END
```

Line 120 displays the message "DO YOU WANT TO CONTINUE (Y,N) and line 125 then waits for a response to be entered from the keyboard. This response is stored in the string A$, which is dimensioned to a length of 5 in line 17. Line 130 compares the first letter of this string to "Y" and if A$(1,1) = "Y" the program branches back to line 20 and the area of another triangle is found. Any other response will terminate the program.

The string A$(I,J) is the substring of A$ that starts at character number I and continues through character number J. Thus, A$(1,1) is the first letter in A$. Therefore, line 120 will branch to line 20 if either "Y" or "YES" was typed in line 125. (The string A$(I) is the substring consisting of the characters in A$ starting at location I and continuing to the end of the string.)

The BASIC listing of this modified program is shown in Figure 8.1 and a sample run is shown in Figure 8.2. Remember that if the response to an INPUT statement is expected to be a nonnumeric value, then a string variable must be used in the INPUT statement. If the INPUT statement contains a numerical variable and the user types in a letter or other nonnumeric value, the ATARI will respond with the message ERROR- 8 and exit the program.

An INPUT statement containing a string variable will accept any input but will treat it as a string. Thus, in line 130 in Figure 8.1 the substring A$(1,1) must be compared to the *string* "Y".

FIGURE 8.1 BASIC listing of modified triangle program.

```
10 REM PROGRAM TO FIND THE
15 REM AREA OF A TRIANGLE
17 DIM A$(5)
20 ? "ENTER THE THREE SIDES OF A TRIANGLE"
30 INPUT A,B,C
40 IF A>B THEN L=A:GOTO 60
50 L=B
60 IF C>L THEN L=C
70 S=(A+B+C)/2
80 IF L>S THEN ? "NO TRIANGLE POSSIBLE":GOTO 20
90 AREA=(S*(S-A)*(S-B)*(S-C))^0.5
100 ? "THE AREA OF THE TRIANGLE IS ";AREA
110 ?
120 ? "DO YOU WANT TO CONTINUE (Y,N)";
125 INPUT A$
130 IF A$(1,1)="Y" THEN 20
140 END
```

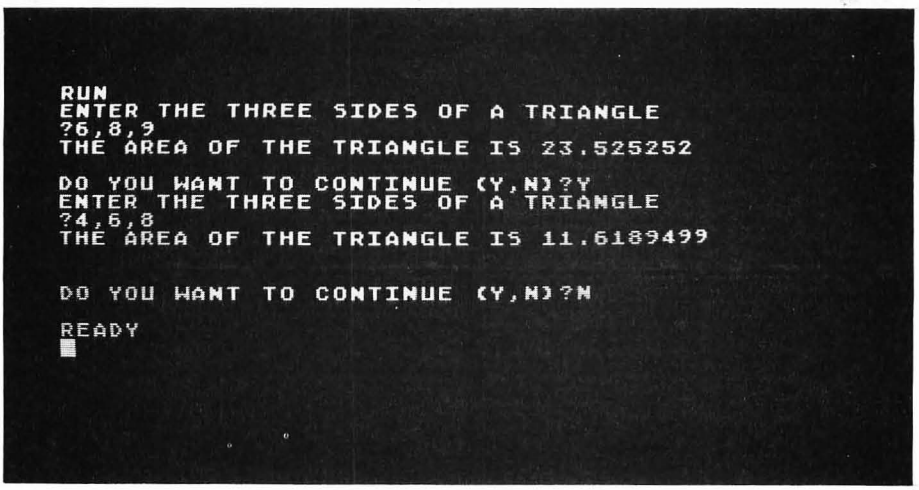


FIGURE 8.2 Sample run of program shown in Figure 8.1.

RANDOM STRIPE PATTERNS

In this section we will write a program that will draw a random horizontal stripe pattern. The pattern will contain 40 horizontal lines each 80 spaces long. In other words the picture will take up the entire 80×40 screen area in the low-resolution graphics mode 5. Each stripe that is plotted will have a 50/50 chance of being one of two possible colors. These two colors can be specified by the user.

A pseudocode description and a structured flowchart for this program are shown in Figure 8.3. After clearing the screen the user specifies two color numbers C1 and C2 in the INPUT statement. The variable Y is used to specify the line number (0–39) at which a particular horizontal line is drawn.

Each time through the inner *repeat while* loop a single stripe is drawn. The line number Y is increased by 1 each time through this loop. The color of each stripe is determined by the value of a random number R. The value of this random number is between 0 and 1. If it is less than 0.5 (there will be a 50/50 chance of this), then the color in color register 0 (COLOR 1) is used for the next stripe. Otherwise, the color in color register 1 (COLOR 2) is used. This loop is repeated *while* Y< = 39. After the stripe pattern is plotted, the user is asked if another picture is wanted. If so, the screen is cleared and the entire program is executed again. Otherwise, the screen is cleared and the program terminates.

A BASIC listing of this program is shown in Figure 8.4. Compare this listing carefully with the pseudocode and structured flowchart representations of the program shown in Figure 8.3. Note in particular how the *repeat while* and *if . . . then . . . else* constructs are implemented in BASIC. Line 120 will cause the screen to be cleared of the previous stripe pattern.

You should type in this program and run it. A sample run is shown in Figure 8.5.

EXERCISE 8.1

Use a FOR . . . NEXT loop to implement the random stripe algorithm in Figure 8.3. Run the program and compare the result with Figure 8.5.

FIGURE 8.3 (a) Pseudocode of program to draw random stripes; (b) structured flowchart for program to draw random stripes.

```
clear screen
loop:   input 2 colors C1, C2
        GR. 5: SETCOLOR 0,C1,8
        SETCOLOR 1,C1,8
        Y = 0
        loop:   R = RND(0)
                if R < 0.5
                then COLOR 1
                else COLOR 2
                Draw horizontal line at Y
                Y = Y + 1
        repeat while Y < = 39
        input "another picture?";A$
        clear screen to green
repeat while A$(1,1) = "Y"
```

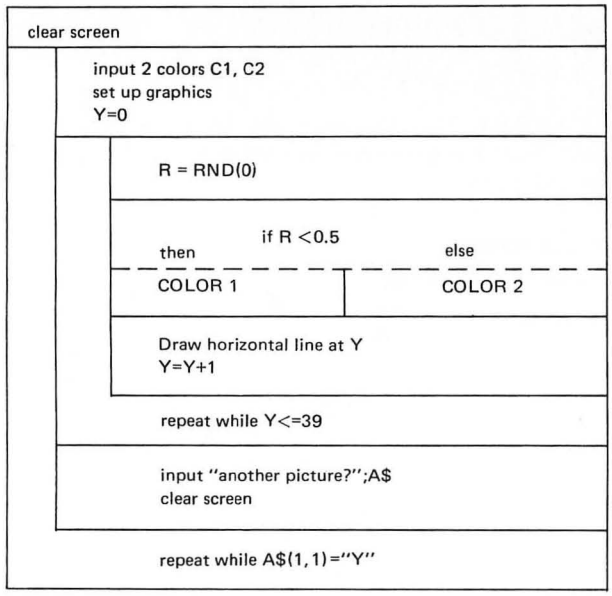
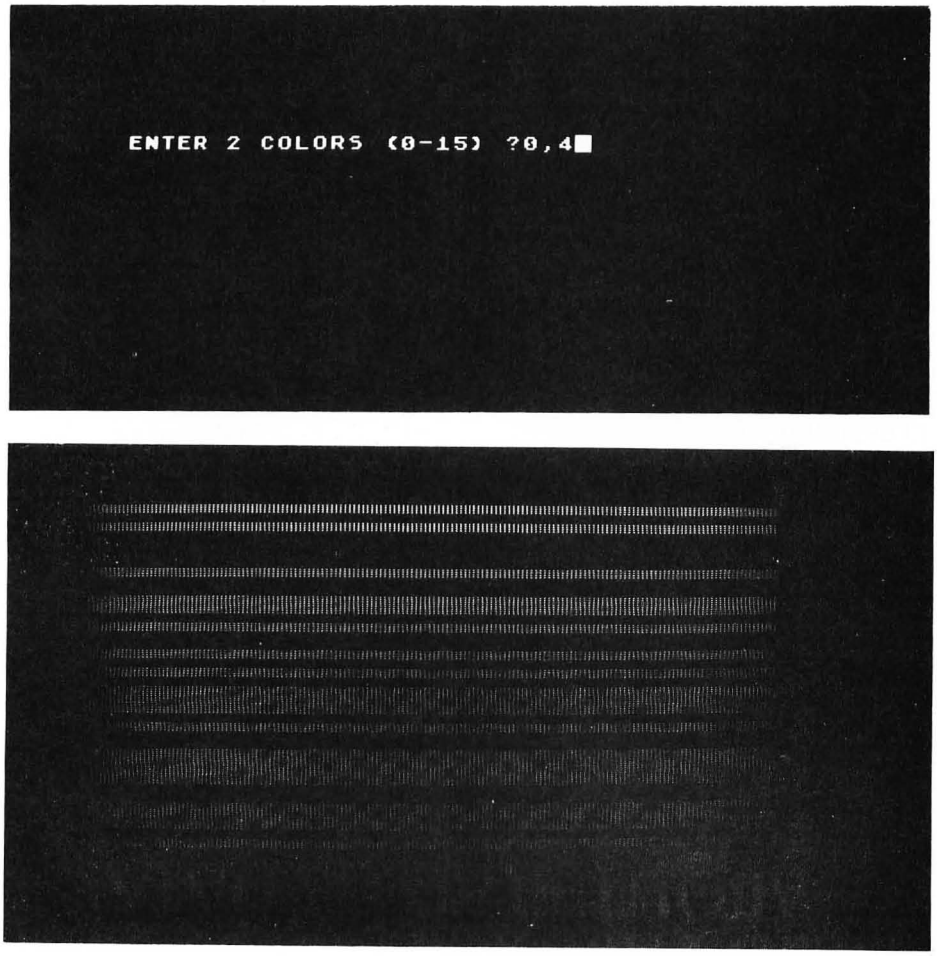


FIGURE 8.3 (cont.)

```
10 REM RANDOM STRIPES
15 DIM A$(5)
20 ? "{CLR}"
30 ? "ENTER 2 COLORS (0-15) ";
35 INPUT C1,C2
40 GRAPHICS 5:SETCOLOR 0,C1,8:SETCOLOR 1,C2,8
45 Y=0
50 R=RND(0)
60 IF R<0.5 THEN COLOR 1:GOTO 70
65 COLOR 2
70 PLOT 0,Y:DRAWTO 79,Y
80 Y=Y+1
90 IF Y<=39 THEN 50
100 ? "ANOTHER PICTURE (Y,N)";
110 INPUT A$
120 GRAPHICS 0
130 IF A$(1,1)="Y" THEN 30
140 END
```

FIGURE 8.4 BASIC listing of program to produce a random stripe pattern.

FIGURE 8.5 Sample run of program shown in Figure 8.4.



RANDOM CHECKERBOARD PATTERN

In this section we will modify the program in Figure 8.4 to plot a random checkerboard pattern rather than a stripe pattern. This can be done by adding another inner loop that will plot a single spot rather than a horizontal line. Each spot will have a 50/50 chance of having one of two possible colors.

Pseudocode and structured flowchart representations of this program are shown in Figure 8.6. Compare these algorithms with the corresponding program descriptions given in Figure 8.3 for the random stripe program. Note that for each line plotted on the screen (which occurs within the *repeat while* Y< = 39 loop) there is another nested *repeat while* X< = 79 loop. This inner loop will plot 80 spots (with random color) on each line. Note that the value of X must be initialized to 0 at the beginning of this inner loop (that is, at the beginning of each new line).

The BASIC listing of this program is shown in Figure 8.7. Compare this listing carefully with the program descriptions shown in Figure 8.6. Make sure you understand clearly how each of the nested *repeat while* loops is implemented in BASIC and what its function is in the execution of the program.

Type in this program and run it. A sample run is shown in Figure 8.8. This program is a good example of how "slow" BASIC is. It will take over a minute to plot one random checkerboard.

EXERCISE 8.2

Use FOR . . . NEXT loops to implement the random checkerboard algorithm given in Figure 8.6. Run the program and compare your result with Figure 8.8.

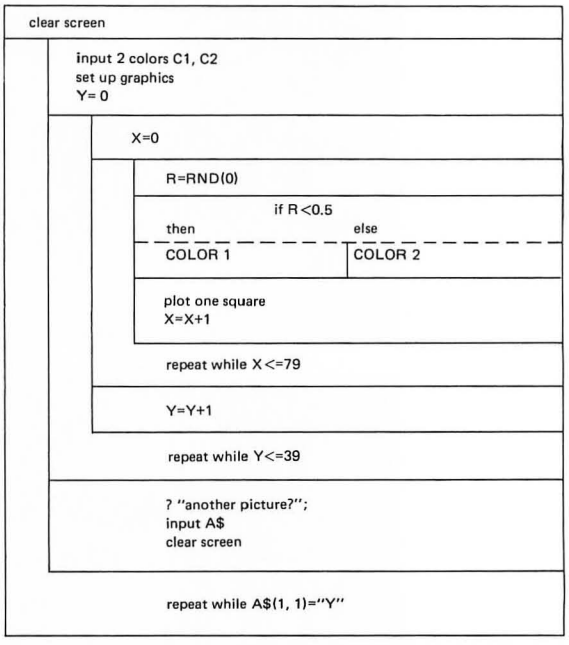
FIGURE 8.7 BASIC listing of program to plot random checkerboard pattern.

```
10 REM RANDOM CHECKERBOARD
15 DIM A$(5)
20 ? "}"
30 ? "ENTER 2 COLORS (0-15) ";
35 INPUT C1,C2
40 GRAPHICS 5:SETCOLOR 0,C1,8:SETCOLOR 1,C2,8
45 Y=0
47 X=0
50 R=RND(0)
60 IF R<0.5 THEN COLOR 1:GOTO 70
65 COLOR 2
70 PLOT X,Y
72 X=X+1
75 IF X<=79 THEN 50
80 Y=Y+1
90 IF Y<=39 THEN 47
100 ? "ANOTHER PICTURE (Y,N)";
110 INPUT A$
120 GRAPHICS 0
130 IF A$(1,1)="Y" THEN 30
140 END
```

FIGURE 8.6 (a) Pseudocode and (b) structured flowchart for program to plot a random checkerboard pattern.

```
clear screen
loop:          input 2 colors C1, C2
               set up graphics
               Y = 0
               loop: X = 0
                     loop: R = RND(0)
                           if R < 0.5
                           then COLOR 1
                           else COLOR 2
                           plot one square
                           X = X + 1
                     repeat while X < = 79
                     Y = Y + 1
               repeat while Y < = 39
               ? "another picture?";
               input A$
               clear screen
repeat while A$(1,1) = "Y"
```

(a)



(b)

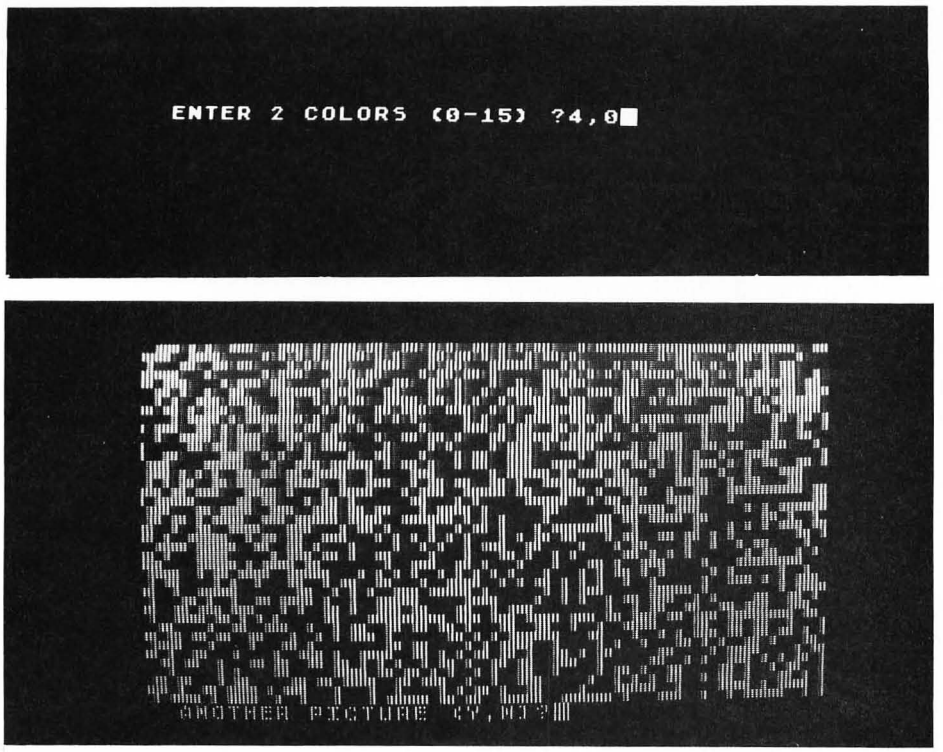


FIGURE 8.8 Sample run of program shown in Figure 8.7.

DIFFERENT TYPES OF LOOPS

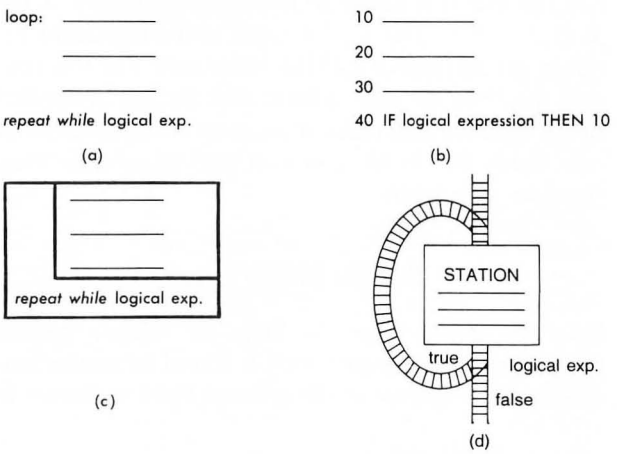
There are really four different elementary loop structures that are available. You can test the logical expression at the beginning of the loop or at the end of the loop. In addition, you can branch out of the loop when the logical expression is either *true* or *false*. We will call the two loops with the test at the end of the loop the *repeat while* and *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. In addition to these elementary loops, it is possible to use a more general loop structure in which the test of the logical expression is done in the middle of the loop. Depending upon whether the loop is exited when the logical expression is true or false we will call these two general loop structures the *loop . . . exit if . . . endloop* and the *loop . . . continue if . . . endloop* loops.

All of these loop structures can be implemented in BASIC. As we will see, some are easier to implement than others. Most good programmers use only two or three of these loop structures in all of their programs. The choice of which ones to use 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 in all of the programs in this chapter. Its general form is shown in Figure 8.9. 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

FIGURE 8.9 The *repeat while* loop: (a) pseudocode; (b) BASIC implementation; (c) structured flowchart; (d) train track equivalent.

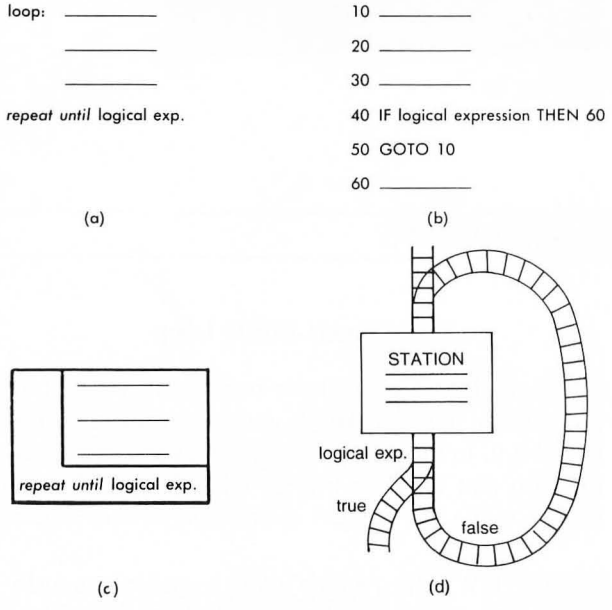


8.9d shows what this loop looks like in our train track model of a computer program. Note that the train continues to loop around through the station as long as the logical expression is true.

The *Repeat Until* Loop

The general form of the *repeat until* loop is shown in Figure 8.10. Note that 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 to avoid logical errors because you will always be thinking either *while* or *until*. Many people prefer the *repeat until* and some languages implement this loop directly.

FIGURE 8.10 The *repeat until* loop: (a) pseudocode; (b) BASIC implementation; (c) structured flowchart; (d) train track equivalent.

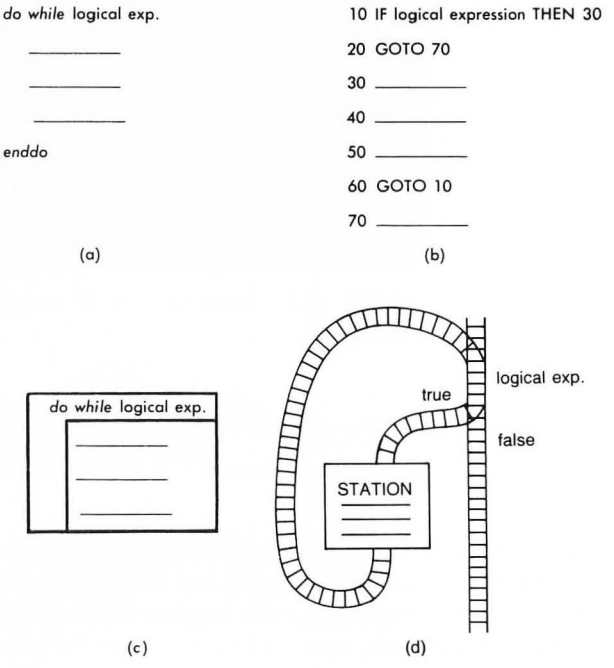


However, by comparing Figures 8.9 and 8.10 you can see that it is easier to implement a *repeat while* loop in BASIC. The *repeat until* implementation requires an additional GOTO statement. For this reason, any time we form a loop with the test at the end of the loop we will make it a *repeat while* loop. After you finish this book you can use whichever loop structure you want.

The *Do While* Loop

The *do while* loop is one of those "good" programming statements that is found in newer languages such as PASCAL. Its general form is shown in Figure 8.11. In this loop the test of the logical expression is done at the *beginning* of the loop. This means that if the logical expression is initially *false,* the train will *never* go to the station. That is, the statements within the loop will never be executed. Note that 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.

FIGURE 8.11 The *do while* loop: (a) pseudocode; (b) BASIC implementation; (c) structured flowchart; (d) train track equivalent.



The *Do Until* Loop

The fourth elementary loop structure is the *do until* loop, whose general structure is shown in Figure 8.12. In this loop the test of the logical expression is also done at the beginning of the loop. However, the statements within the loop are only executed if the logical expression is *false*—that is, *until* the logical expression is *true*. Note that if the logical expression is initially *true,* the train will never get to the station and the statements within the loop will never be executed.

Note also that the BASIC implementation of the *do until* loop requires only one GOTO statement rather than the two needed for the *do while* loop. 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 the loop.

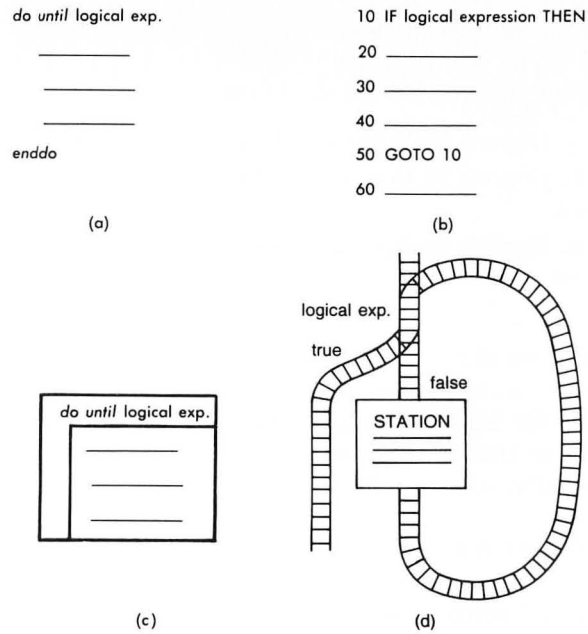


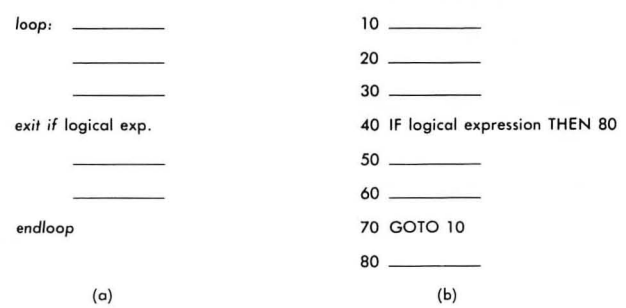
FIGURE 8.12 The *do until* loop: (a) pseudocode; (b) BASIC implementation; (c) structured flowchart; (d) train track equivalent.

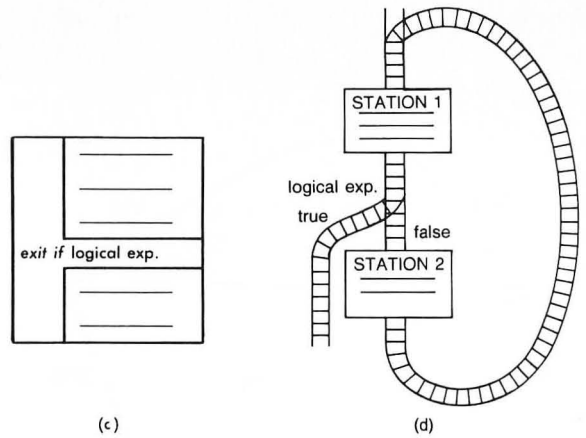
People who write structured programs using a "good" structured programming language use the *do while* and the *repeat until* loops. As we have seen it will save us some code (and therefore some memory) if instead we use the *do until* and the *repeat while* loops. However, any of these loops can be used without much difficulty.

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* construct, whose general form is shown in Figure 8.13. This is really a generalized *until* loop. That is, if the *exit if* statement is at the top of the loop, it reduces to the *do until* loop. If the *exit if* statement is at the bottom of the loop, it reduces to the *repeat until* loop.

FIGURE 8.13 The *loop . . . exit if . . . endloop* loop: (a) pseudocode; (b) BASIC implementation; (c) structured flowchart; (d) train track equivalent.



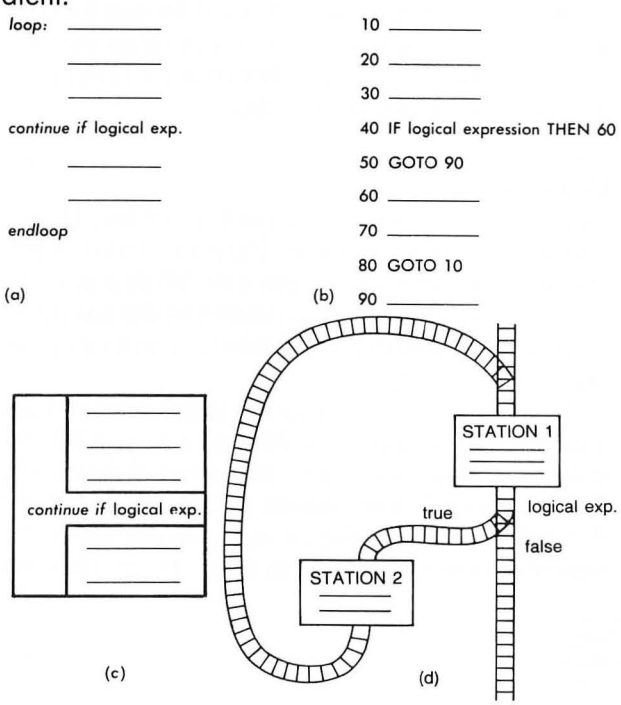


The *Loop . . . Continue If . . . Endloop* Loop

In order to complete the discussion of loops, we will show the general form of the *loop . . . continue if . . . endloop* in Figure 8.14. This is really a generalized *while* loop. That is, if the *continue if* statement is at the top of the loop, it reduces to the *do while* loop. If the *continue if* statement is at the bottom of the loop, it reduces to the *repeat while* loop.

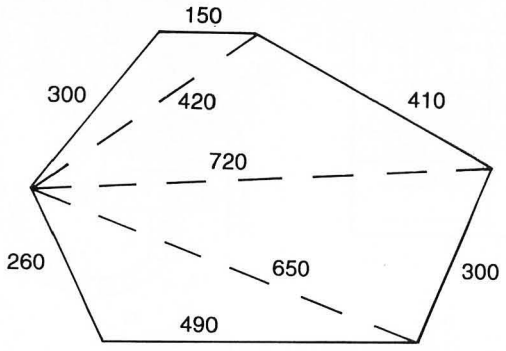
Any of the loops that have been described can be used in your programs, but as we have seen, the easiest ones to implement in BASIC are the *repeat while,* the *do until,* and the *loop . . . exit if . . . endloop* structures.

FIGURE 8.14 The *loop . . . continue if . . . endloop* loop: (a) pseudocode; (b) BASIC implementation; (c) structured flowchart; (d) train track equivalent.



EXERCISE 8.3

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



Modify the program shown in Figure 8.1 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, adding these results, and using the fact that 1 acre = 43,560 square feet.

EXERCISE 8.4

Suppose that the tract of land shown in Exercise 8.3 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 8.5

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 immediately preceding numbers. Write a program that will display on the screen all numbers in the Fibonacci sequence that are less than 1,000.

EXERCISE 8.6

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 input D and P; 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, where 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 case:

1. Deposit $500 per month at 8 percent interest.
2. Deposit $1,000 per month at 10 percent interest.
3. Deposit $1,000 per month at 12 percent interest.

EXERCISE 8.7

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

EXERCISE 8.8

If you deposit $100 each year in a bank account paying 5 percent interest compounded annually, how much money will you have after 10 years?

EXERCISE 8.9

Population growth. In 1974 the U.S. birth rate was 14.9 births per 1,000 population, the death rate was 9.1 deaths per 1,000 population, and the net migration rate was 1.7 per 1,000 population. Assume that these rates will remain constant in the future and that the population of the United States at the beginning of 1976 was 214,398,000. Also assume for the purpose of simulating this process on the computer that all births, deaths, and migrations take place on the last day of each year. Write and run a program that will determine in which year the population of the United States will reach 300,000,000.

EXERCISE 8.10

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

$$H = -16.2T^2 + VT$$

Write a program that will

1. input a value of V
2. print the letters T and H for a table heading
3. compute H for values of T starting at 0 and increasing by 1 second until the rocket hits the ground
4. print the values of T and H in the form of a table.

Run the program using a value of V = 200 ft/s.

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 same 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 that function needs to be performed. The ATARI screen can display a maximum of only 22 program lines (leaving two lines at the bottom for the cursor); therefore, if you can keep your main program and all subroutines less than 22 screen lines long, you will be able to read and study a complete program segment without having to scroll the screen. This technique of modularizing your program will 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 with which you write short programs.

In this chapter you will learn

1. how to use the GOSUB and RETURN statements

2. to plot the same figure at different locations on the screen

3. to plot figures of varying sizes

4. how to display your name anywhere on the screen

5. to use the game paddles and joysticks.

THE GOSUB AND RETURN STATEMENTS

The general form of the GOSUB statement is

GOSUB line number

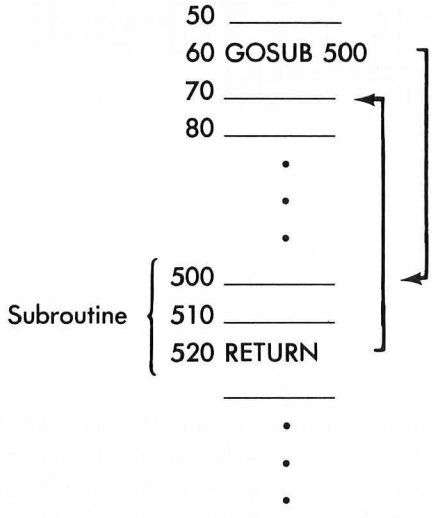
When this statement is executed, the program branches to the statement at line "line number." For example, the statement GOSUB 500 will cause the program to branch to line 500. It looks as if GOSUB 500 behaves the same way as GOTO 500. However, there is an important difference. The ATARI *remembers* where the statement GOSUB 500 is located in

the program. Line number 500 is the first line of a *subroutine* that is just a collection of BASIC statements that perform a particular task. At the end of this subroutine you must include the statement

RETURN

When the RETURN statement is executed, the program will then branch back to the next statement following GOSUB 500. This process is shown in Figure 9.1

FIGURE 9.1 Forming a subroutine using GOSUB and RETURN.



Now it looks as if you would accomplish the same result in Figure 9.1 by using the two statements

60 GOSUB 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. Recall that the ATARI always remembers the point from which it branched to a subroutine and it will always return to that point.

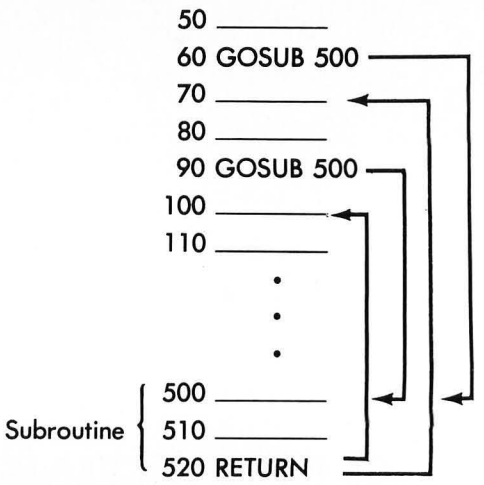
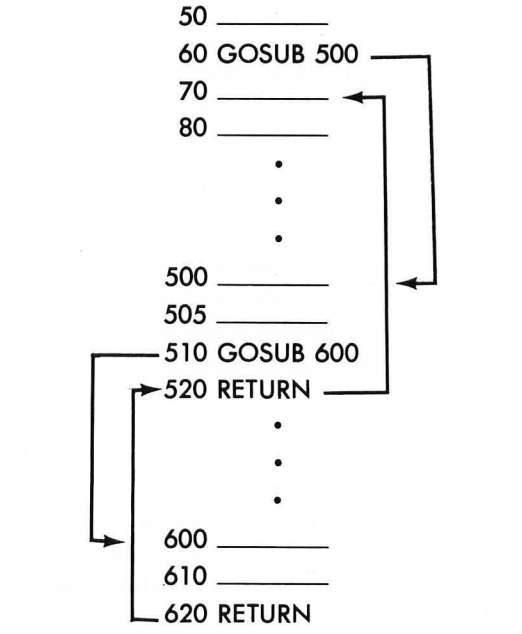


FIGURE 9.2 Calling a subroutine from two different locations within a program.

You can even call a subroutine from within another subroutine. The ATARI will always find its way back by retracing its steps as shown in Figure 9.3.

FIGURE 9.3 One subroutine can call another subroutine.



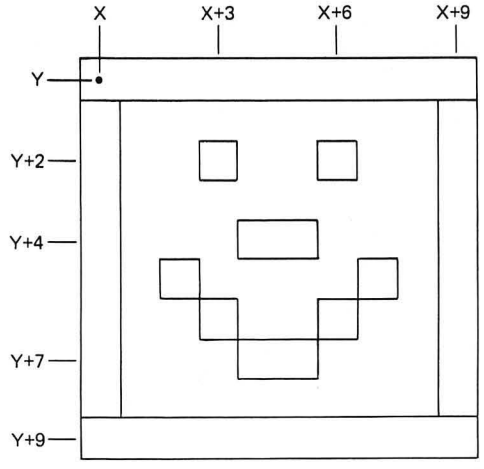
Line 60 branches to the subroutine at line 500. Line 510, which is within this subroutine, branches to a second subroutine at line 600. The RETURN statement on line 620 will branch back to line 520, the statement following the GOSUB 600 statement. This happens to be the RETURN statement of the subroutine that begins at line 500. It will then branch back to line 70, the statement following the GOSUB 500 statement.

PLOTTING MULTIPLE FIGURES

The graphic figure shown in Figure 9.4a can be plotted using the subroutine in Figure 9.4b. All points in this figure are defined relative to the X,Y coordinate of the upper-left-hand corner of the figure. Lines 105–130 in Figure 9.4b draw the box in a clockwise fashion starting at the position X,Y. Line 140 plots the two eyes. Line 150 plots the nose and the mouth is drawn in lines 160–170. Note that line 180 is the RETURN statement. Study the subroutine in Figure 9.4b carefully and make sure you understand how it draws the face in Figure 9.4a.

FIGURE 9.4 (a) Definition of graphic figure; (b) subroutine to plot the figure in (a).

(a)



(b)

```
100 REM PLOT FACE
105 PLOT X,Y:DRAWTO X+9,Y
110 DRAWTO X+9,Y+9
120 DRAWTO X,Y+9
130 DRAWTO X,Y
140 PLOT X+3,Y+2:PLOT X+6,Y+2
150 PLOT X+4,Y+4:PLOT X+5,Y+4
160 PLOT X+2,Y+5:DRAWTO X+4,Y+7
170 PLOT X+5,Y+7:DRAWTO X+7,Y+5
180 RETURN
```

This subroutine must have valid values for X and Y before it is called. In order to test this subroutine, type it in as shown in Figure 9.4b. Then in the immediate mode, type

GR. 5:SETCOLOR 0,0,8:COLOR 1

This will put you into the low-resolution graphics mode 5. Now type

X=10:Y=10:GOSUB 100

This should display the figure shown in Figure 9.5.

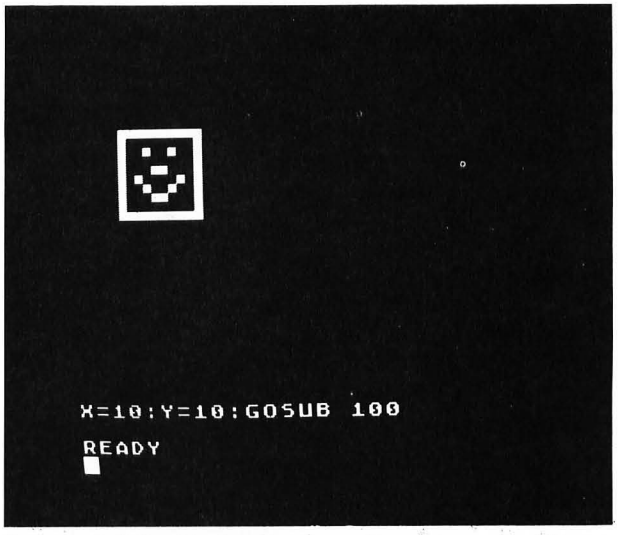


FIGURE 9.5 Test of subroutine given in Figure 9.4b.

Note that you can execute the statement GOSUB 100 in the immediate mode. This is very useful for testing subroutines.

Now that you know that the subroutine works, you can plot multiple faces by simply calling this subroutine several times with different values for X and Y. The program shown in Figure 9.6 calls the statement GOSUB 100 nine times using a nested FOR . . . NEXT loop. This loop will produce the following nine pairs of values for X and Y:

X	Y
0	0
15	0
30	0
0	15
15	15
30	15
0	30
15	30
30	30

These nine pairs of values will correspond to the coordinates of the upper-left-hand corner of the nine faces.

You should type in and run the program shown in Figure 9.6. (If you already have the subroutine typed in you only need to add lines 10–60.) The result of executing this program is shown in Figure 9.7.

Modify this program to plot only four faces. Note that the END statement in line 60 is required to prevent the subroutine at line 100 from being executed an extra time without being called from a GOSUB statement.

```
10 REM MULTIPLE FIGURES
20 GRAPHICS 5:SETCOLOR 0,0,14:COLOR 1
25 SETCOLOR 2,0,0:SETCOLOR 1,0,14
30 FOR Y=0 TO 30 STEP 15
40 FOR X=10 TO 50 STEP 20
50 GOSUB 100:NEXT X:NEXT Y
60 END
100 REM PLOT FACE
105 PLOT X,Y:DRAWTO X+9,Y
110 DRAWTO X+9,Y+9
120 DRAWTO X,Y+9
130 DRAWTO X,Y
140 PLOT X+3,Y+2:PLOT X+6,Y+2
150 PLOT X+4,Y+4:PLOT X+5,Y+4
160 PLOT X+2,Y+5:DRAWTO X+4,Y+7
170 PLOT X+5,Y+7:DRAWTO X+7,Y+5
180 RETURN
```

FIGURE 9.6 Program to plot nine faces on the screen.

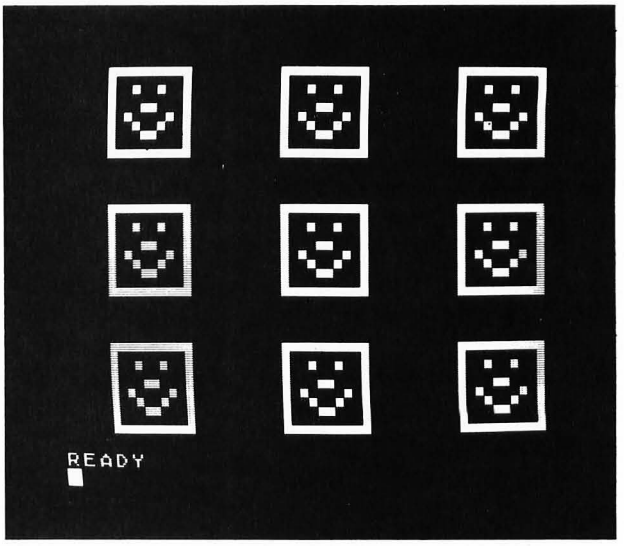


FIGURE 9.7 Result of executing the program given in Figure 9.6.

Plotting Different-Sized Figures

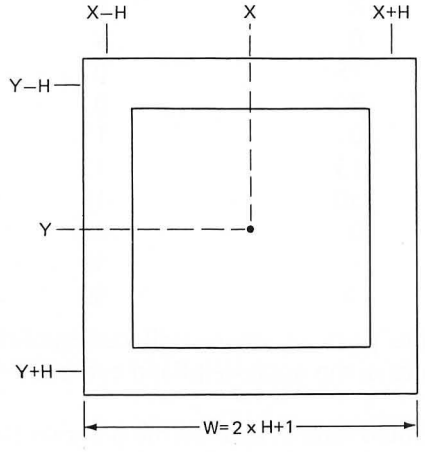
In addition to making the location of a figure variable, you can also change the size of a figure. For example, Figure 9.8a shows a square centered at X,Y whose width is 2 * H + 1. The subroutine shown in Figure 9.8b will plot this square.

Type in this subroutine and then test it by typing

GR. 5:SETCOLOR 0,0,8:COLOR 1

FIGURE 9.8 (a) Definition of square of width 2 * H + 1; (b) subroutine to plot the square in (a).

(a)



(b)

```
200 REM SQUARE OF WIDTH 2*H+1
210 PLOT X-H,Y-H:DRAWTO X+H,Y-H
220 DRAWTO X+H,Y+H
230 DRAWTO X-H,Y+H
240 DRAWTO X-H,Y-H
250 RETURN
```

to enter the low-resolution graphics mode 5, followed by

X=40:Y=20:H=10:GOSUB 200

This should display the figure shown in Figure 9.9.

You can now plot multiple squares of different sizes by calling the subroutine in Figure 9.8b with different values of H. For example, the program shown in Figure 9.10 will plot seven concentric squares, all centered at X = 40, Y = 20.

You should type in this program by adding lines 10–60 to the "square" subroutine in Figure 9.8b. If you run the program you should obtain the figure shown in Figure 9.11. Try running this program after changing the step size in line 40 to 2, 4, 6, and 1.

FIGURE 9.9 Test of subroutine given in Figure 9.8b.

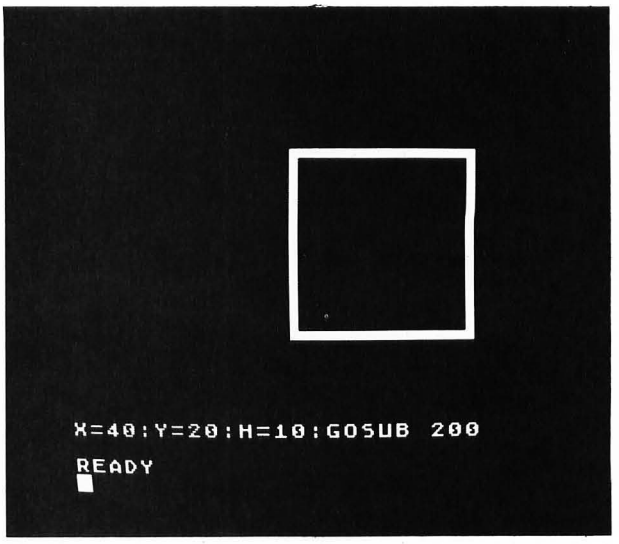


FIGURE 9.10 Program to plot concentric squares.

```
10 REM CONCENTRIC SQUARES
20 GRAPHICS 5:SETCOLOR 0,0,14:COLOR 1
25 SETCOLOR 2,0,0:SETCOLOR 1,0,14
30 X=40:Y=20
40 FOR H=1 TO 19 STEP 3
50 GOSUB 200:NEXT H
60 END
200 REM SQUARE OF WIDTH 2*H+1
210 PLOT X-H,Y-H:DRAWTO X+H,Y-H
220 DRAWTO X+H,Y+H
230 DRAWTO X-H,Y+H
240 DRAWTO X-H,Y-H
250 RETURN
```

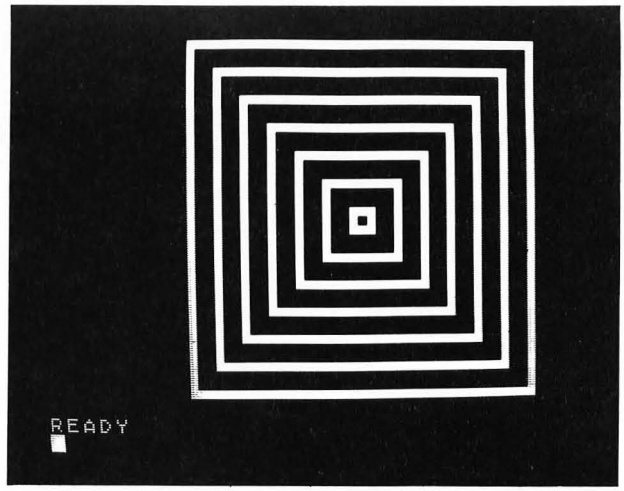


FIGURE 9.11 Result of running the program shown in Figure 9.10.

PLOTTING YOUR NAME

You can use the ideas described in the previous sections of this chapter to plot your name anywhere on the screen using letters of varying sizes. The trick is to define each letter in terms of the X,Y coordinate of its upper-left-hand corner and the width, W, and height, H, of the letter. Then you can plot each letter wherever you want by using subroutines.

As an example, Figure 9.12 shows a subroutine that will plot the letter J. In this figure W is the width of the letter and H is its height. The upper-left-hand corner of the H×W rectangle containing the letter defines the position X,Y of the letter. Study the subroutine shown in Figure 9.12 and make sure you understand how lines 210–230 plot each part of the J.

In a similar way, Figure 9.13 shows a subroutine that will plot the letter F. Note that the short horizontal bar in the F is located at row number $Y + (H - 1)/2$. If H is an odd number the horizontal bar will be placed at the middle of the F. Also note that the length of this bar is $(W - 1)/2$.

Figure 9.14a defines the letter E. The subroutine in Figure 9.14b begins by plotting an F in line 410. Line 420 then adds the bottom horizontal bar to produce an E.

In order to test these subroutines, type in the lines shown in Figures 9.12b, 9.13b, and 9.14b. Then enter the low-resolution graphics mode 5 by typing

```
GR. 5:SETCOLOR 0,0,8:COLOR 1
```

You can then plot a J by typing

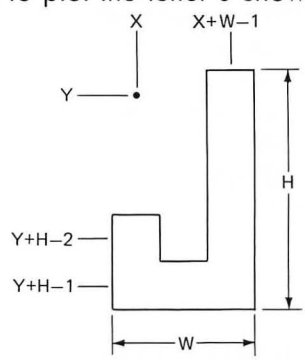
```
X=10:Y=10:W=3:H=5:GOSUB 200
```

as shown in Figure 9.15. In order to test the subroutines for plotting the F and the E, type

```
X=20:GOSUB 300
```

FIGURE 9.12 (a) Definition of the letter J; (b) subroutine to plot the letter J shown in (a).

(a)

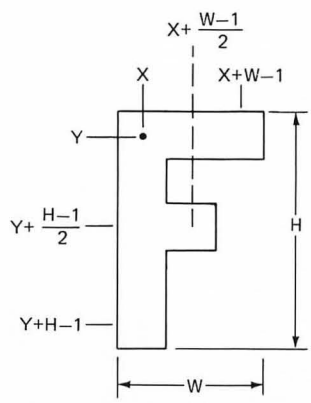


(b)

```
200 REM PLOT A J  (H X W)
210 PLOT X+W-1,Y:DRAWTO X+W-1,Y+H-1
220 DRAWTO X,Y+H-1
230 PLOT X,Y+H-2
240 RETURN
```

FIGURE 9.13 (a) Definition of the letter F; (b) subroutine to plot the letter F shown in (a).

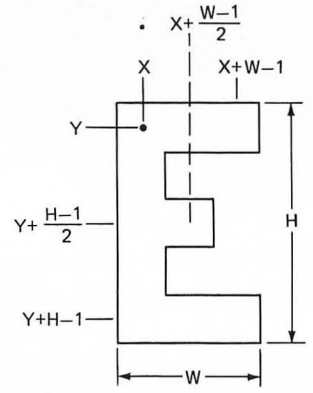
(a)



```
300 REM PLOT AN F   (H X W)
310 PLOT X+W-1,Y:DRAWTO X,Y
320 DRAWTO X,Y+H-1
330 PLOT X+1,Y+(H-1)/2:DRAWTO X+(W-1)/2,Y+(H-1)/2
340 RETURN
```

FIGURE 9.14 (a) Definition of the letter E; (b) subroutine to plot the letter E shown in (a).

(a)



(b)

```
400 REM PLOT AN E   (H X W)
410 GOSUB 300:REM PLOT F
420 PLOT X+1,Y+H-1:DRAWTO X+W-1,Y+H-1
430 RETURN
```

FIGURE 9.15 Testing the subroutine shown in Figure 9.12.

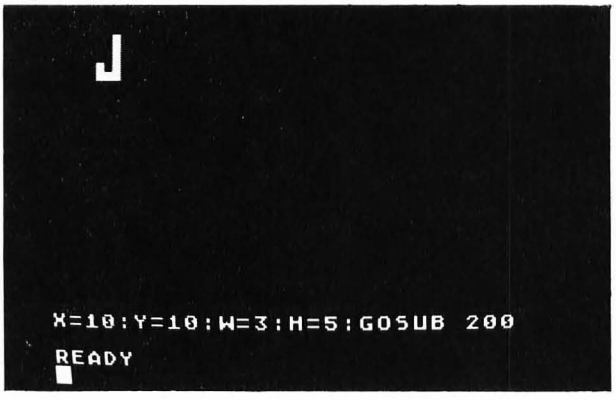
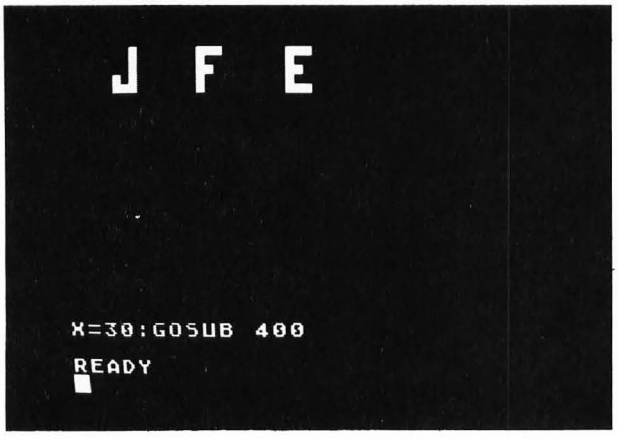


FIGURE 9.16 Testing the subroutine to plot individual letters.



and

```
X=30:GOSUB 400
```

as shown in Figure 9.16.

Figure 9.17 shows a subroutine that will combine the letters J, E, and F to plot the name JEFF. Before calling this subroutine, X, Y, W, and H must have been assigned values. Line 510 plots a J at location X,Y. Line 520 plots an E where the X location has been increased by W + 2. This will leave a blank column between the J and the E. Similarly, lines 530–540 plot the two Fs.

A main program that plots three JEFFs of different sizes is shown in Figure 9.18. Lines 20–30 will plot the name with 5×3 letters. Lines 40–50 will plot the name at a different location using 7×5 letters. Finally, the name with 15×11 letters is plotted in lines 60–70. The result of running this program is shown in Figure 9.19.

FIGURE 9.17 Subroutine to plot the name JEFF.

```
500 REM PLOT JEFF (H X W)
510 GOSUB 200:REM J
520 X=X+W+2:GOSUB 400:REM E
530 X=X+W+2:GOSUB 300:REM F
540 X=X+W+2:GOSUB 300:REM F
550 RETURN
```

FIGURE 9.18 Main program that plots the name JEFF three times.

```
10 REM PLOT 3 JEFFS
15 GRAPHICS 5:SETCOLOR 0,0,14:COLOR 1
17 SETCOLOR 2,0,0:SETCOLOR 1,0,14
20 X=12:Y=2:W=3:H=5
30 GOSUB 500
40 X=15:Y=10:W=5:H=7
50 GOSUB 500
60 X=10:Y=20:W=11:H=15
70 GOSUB 500
80 END
```

EXERCISE 9.1

Write a program that will plot your name at two different locations on the screen. The size of the two names should be different.

EXERCISE 9.2

Write a program that will plot a set of seven concentric diamonds.

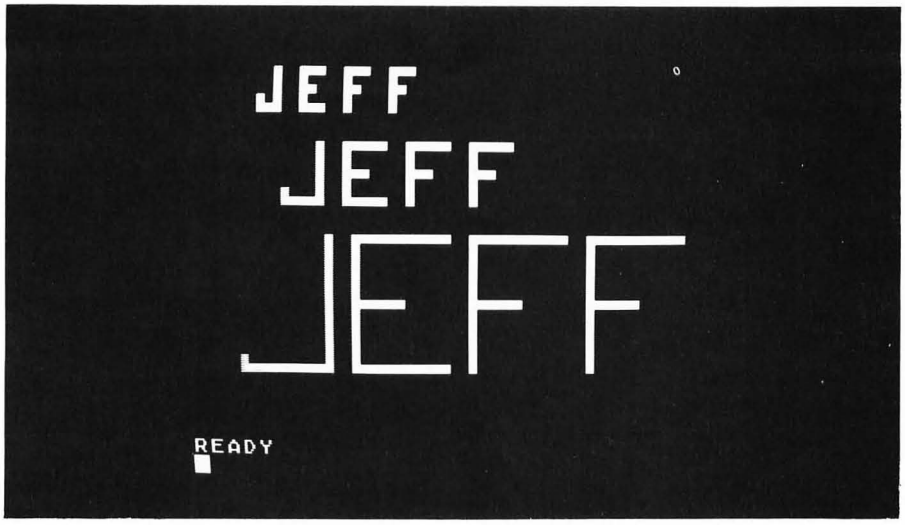


FIGURE 9.19 Result of running the program shown in Figure 9.18.

USING THE GAME PADDLES

The ATARI game paddles can be plugged into any of the four controller jacks on the front of the computer below the keyboard. The game paddles come in pairs with two paddles connected to a single plug. The plugs are numbered 1 to 4 from left to right.

The value of a paddle is read using the paddle function PADDLE(N). The argument N is a number between 0 and 7. This value will depend on which controller jack the paddles are connected to, as shown in Figure 9.20. For example, if we connect the paddles to controller jack 3, then the two functions PADDLE(4) and PADDLE(5) will read the two paddles.

To see this, connect the paddles to jack 3, type the following one-line program and run it.

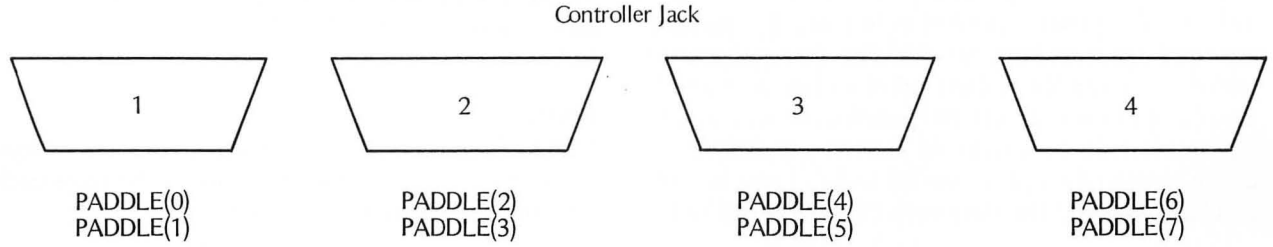
```
10 ?PADDLE(4),PADDLE(5):GOTO 10
```

Two columns of numbers will scroll off the screen. Turn the two paddles. Note that the values change from 0 to 228 as the paddles are rotated. The value of 228 occurs when the paddles are rotated completely counterclockwise.

Now type in the program shown in Figure 9.21. Line 20 sets the low-resolution graphics mode 5 with a white color. Line 30 assigns X an integer value between 0 and 76 depending upon the position of the knob on paddle 4. Note that the maximum possible value is 228/3 = 76. This is done so that in the PLOT statement in line 50 the maximum value of X will be 76. Similarly, line 40 assigns Y an integer value between 0 and 38 depending upon the position of the knob on paddle 5.

The PLOT statement in line 50 causes a white dot

FIGURE 9.20 Eight paddles can be connected to the ATARI through the four controller jacks.



```
10 REM USING THE PADDLES
20 GRAPHICS 5:SETCOLOR 0,0,14:COLOR 1
30 X=INT(PADDLE(4)/3)
40 Y=INT(PADDLE(5)/6)
50 PLOT X,Y
60 GOTO 30
```

FIGURE 9.21 Program to draw figures using the game paddles.

to be plotted on the screen. The exact position of the dot will depend on the position of the two paddle knobs. Line 60 branches back to line 30 and the two paddles are read again. Thus, this program should continually display new spots on the screen as the paddle knobs are turned.

A sample run of this program is shown in Figure 9.22. You should try running this program. Press the BREAK key to stop the program.

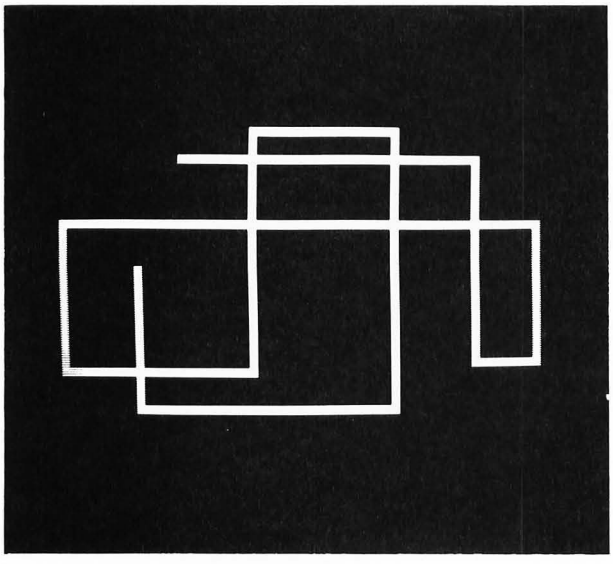


FIGURE 9.22 Sample run of program shown in Figure 9.21.

The function PTRIG(J) (J = 0 – 7) is used to tell if the paddle trigger button on paddle number J is being pressed. The value of PTRIG(J) will be 0 only if the trigger button on paddle number J is being pressed.

Suppose that you want to move a single spot around the screen in response to the game paddles. The program shown in Figure 9.23 will do this. Type it in and run it.

Lines 20–50 are the same as in Figure 9.21. After the first white spot is plotted in line 50, the paddles are read again in lines 60–70. The new values are stored in X1 and X2. If these are the same as X and Y measured in lines 30–40, the paddles are read again. This test is made in line 80. As soon as one of the paddles is rotated the test in line 80 will fail and line 90 will be executed. The statement COLOR 0 will point

```
10 REM MOVING SPOT WITH PADDLES
20 GRAPHICS 5:SETCOLOR 0,0,14:COLOR 11
30 X=INT(PADDLE(4)/3)
40 Y=INT(PADDLE(5)/6)
50 PLOT X,Y
60 X1=INT(PADDLE(4)/3)
70 Y1=INT(PADDLE(5)/6)
80 IF X1=X AND Y1=Y THEN 60
90 COLOR 0:PLOT X,Y
100 X=X1:Y=Y1
110 COLOR 1:GOTO 50
```

FIGURE 9.23 Program to move a single spot around the screen.

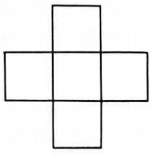
to color register 4, which contains the default background color black. The statement PLOT X,Y will therefore plot a black spot at the same location (X,Y) at which the white spot was plotted in line 50. This will erase the spot. Line 100 then assigns the new screen location X1,Y1 to the values X and Y. Line 110 resets the color to white by pointing to color register 0 and then branches back to line 50, where a new white spot is plotted at X,Y.

An alternate way of writing this program is to read the paddles, plot a white spot, erase the spot, and repeat this process. However, this will result in a blinking spot when the paddles are not being turned. The technique illustrated in Figure 9.23 in which a new spot is plotted only when the paddles are rotated will prevent the spot from blinking.

If the PLOT statements in lines 50 and 90 in Figure 9.23 are replaced with GOSUB statements, larger figures can be moved around the screen. (See Exercise 9.3.)

EXERCISE 9.3

Write a program that will plot the following 3×3 cross in the center of the screen:



Have the cross move around the screen in response to rotating the game paddles.

EXERCISE 9.4

Modify the program in Figure 9.21 so that the figure on the screen is erased each time the paddle trigger button is pressed.

EXERCISE 9.5

Modify Exercise 9.3 so that it uses a joystick instead of game paddles. Change the color of the cross each time the trigger button is pressed.

USING THE JOYSTICKS

An ATARI joystick can be plugged into one of the four controller jacks on the front of the computer (see Figure 9.20). The value of the joystick is read using the stick function STICK(N), where N is the joystick number 0–3. Stick number 0 is plugged into the leftmost controller jack and stick number 3 is plugged into the rightmost controller jack.

The function STICK(N) can only take on the values shown in Figure 9.24, depending on the position of the stick.

The program in Figure 9.25 shows how to produce the same type of figure as the one shown in Figure 9.22 by moving the joystick. The joystick is assumed to be plugged into the rightmost controller jack. If you press the stick trigger button, the value of the function STRIG(N) will be 0. Line 90 in Figure 9.25 checks to see if you are pressing the trigger button; if you are, the program branches to line 20, which erases the screen. Otherwise it will continue to plot a new point in line 40.

Type in this program and run it. Modify the program so that diagonal lines are plotted when the value of the function STICK(N) is 6, 5, 9, or 10 (see Figure 9.24).

FIGURE 9.24 Possible values of the function STICK(N).

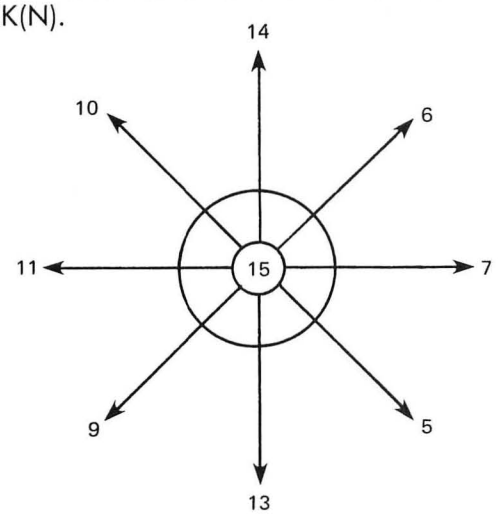


FIGURE 9.25 Program to draw figures using a joystick.

```
10 REM PLOTTING WITH A JOYSTICK
20 GRAPHICS 5:SETCOLOR 0,0,14:COLOR 1
30 X=20:Y=20
35 IF X>79 THEN X=79
36 IF X<0 THEN X=0
37 IF Y>39 THEN Y=39
38 IF Y<0 THEN Y=0
40 PLOT X,Y
50 IF STICK(3)=7 THEN X=X+1:GOTO 90
60 IF STICK(3)=13 THEN Y=Y+1:GOTO 90
70 IF STICK(3)=11 THEN X=X-1:GOTO 90
80 IF STICK(3)=14 THEN Y=Y-1
90 IF STRIG(3)=0 THEN 20
100 GOTO 35
```

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 second is to use an INPUT statement such as INPUT A. In the last 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.

In this chapter you will learn

1. to use the READ, DATA, and RESTORE statements
2. to make horizontal bar graphs
3. to make vertical bar graphs containing multiple bars
4. to 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 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 following statement:

```
10 DATA 5,10
```

and then type

```
RUN
READ A
?A
READ A
?A
READ A
```

as shown in Figure 10.1 The first time you typed READ A, the first data value in the DATA statement (5) was stored in A. The second time you typed READ

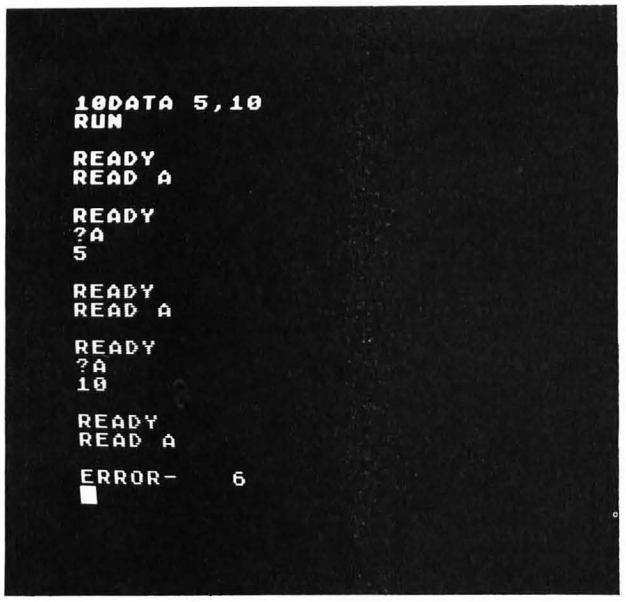


FIGURE 10.1 The READ statement reads successive values from a DATA statement.

A, the second data value in the DATA statement (10) was stored in A. The third time you typed READ A, an error message, ERROR- 6, was displayed. This is an *out of data* error because there were no more data values in the DATA statement to use.

When a program is executed, a *pointer* points to the first data value in the DATA statement. (More than one DATA statement in a program will be treated as a single long DATA statement.) As data values are "used up" by being read in READ statements, the pointer keeps moving 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 ERROR- 6 message will be displayed.

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:

```
10 DATA 5,10,15
   RUN
   READ A,B,C
   ?A,B,C
   RESTORE
   READ B,C
   ?A,B,C
```

as shown in Figure 10.2.

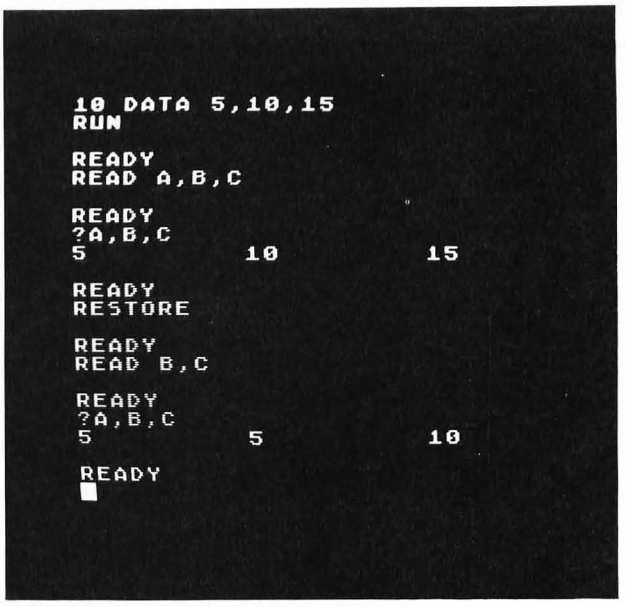


FIGURE 10.2 The RESTORE statement moves the pointer to the first data value in the DATA statement.

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 20, 25, 30

Type RUN, which 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,D
?A,B,C,D
READ A,B,C
```

Note that the two DATA statements are treated as one long DATA statement. DATA statements may occur anywhere in the program. They 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, although there are values for A and B (25 and 30), there is no value for C and therefore the ERROR- 6 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 that has been dimensioned. For example, change the DATA statement in line 20 to

20 DATA ACE,BOSTON

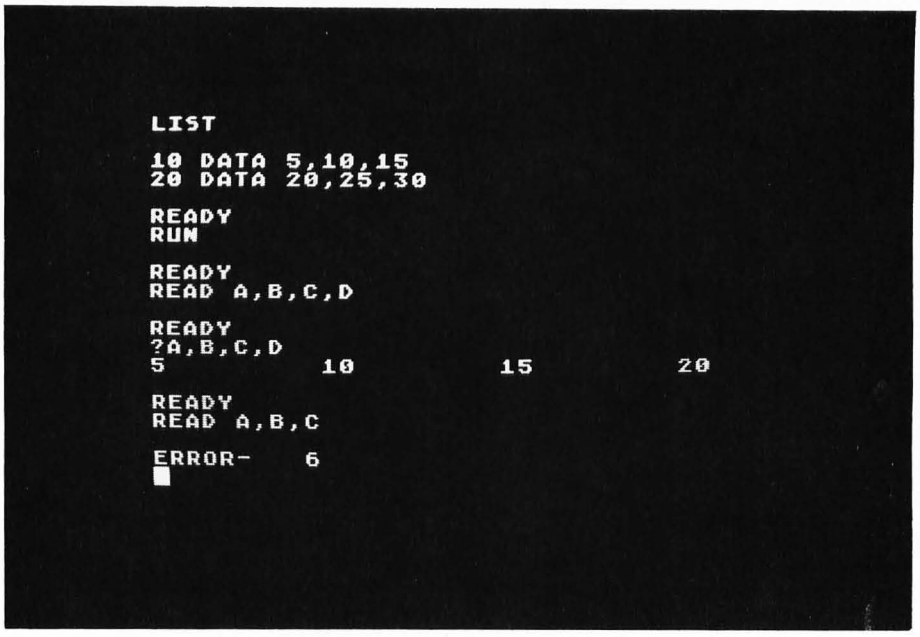


FIGURE 10.3 There must be data values for *all* variable names in a READ statement.

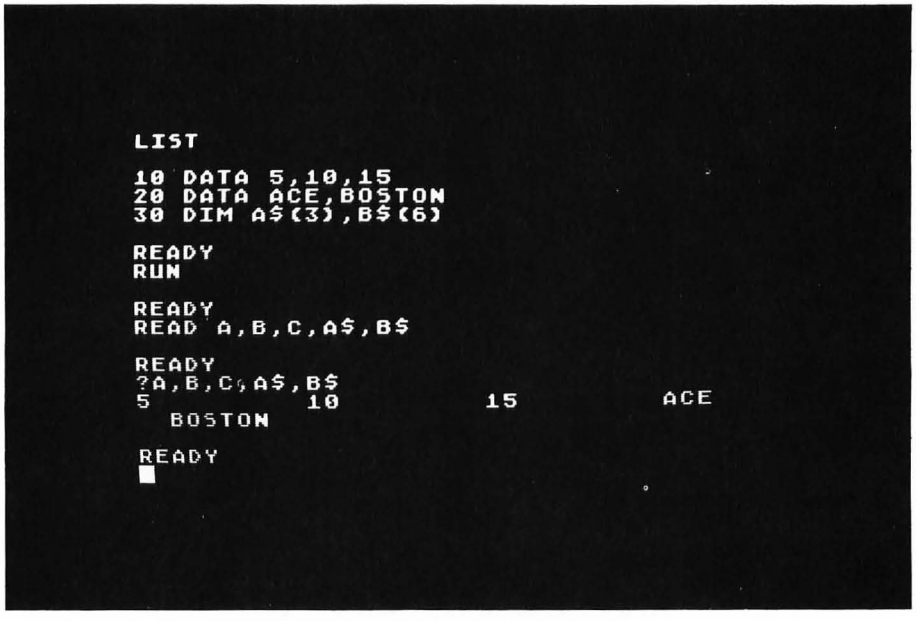


FIGURE 10.4 String variables can be used in a READ statement to read strings in a DATA statement.

and add

```
30 DIM A$(3),B$(6)
```

and then type

```
READ A,B,C,A$,B$
?A,B,C,A$,B$
```

as shown in Figure 10.4. Strings in a DATA statement may contain any characters (including quotation marks) except a comma. Commas are used to separate different entries in a DATA statement. Note also that the numerical variables A and B are completely different memory cells from the string variables A$ and B$. The ATARI will not get these mixed up.

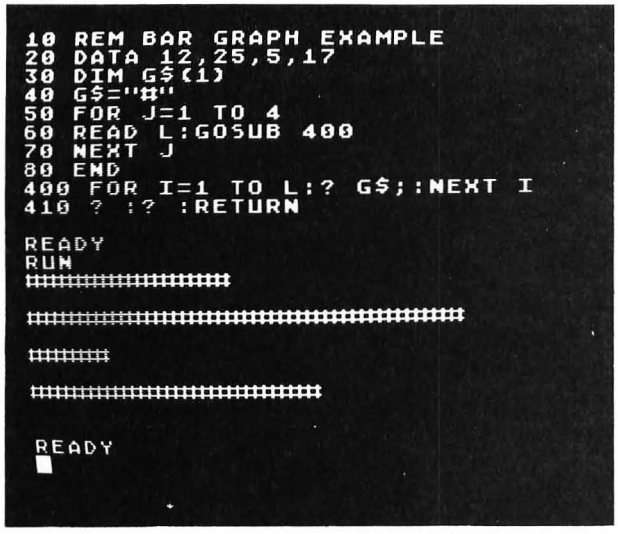
The READ and DATA statements are particularly useful when you have a list of data whose values do not change in the program and which are read by the same READ statement. Examples using the READ and DATA statements will be given in the following sections.

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 ATARI is one that plots a horizontal line whose length is proportional to the quantity of interest.

As an example, suppose that you want to compare graphically the four values 12, 25, 5, and 17. You can plot four lines with lengths 12, 25, 5, and 17 using the program shown in Figure 10.5.

FIGURE 10.5 Program to plot four lines of lengths 12, 25, 5, and 17.



In this program line 20 is a DATA statement that contains the lengths of the four bars to be plotted. Line 40 defines the character (the symbol # in this case) that will be used to draw the lines and stores this character in the string variable G$, which was dimensioned in line 30. Other characters such as hyphens or asterisks could also be used. Lines 50–70 form a FOR . . . NEXT loop that is executed once for each bar that is to be plotted (four, in this case).

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 symbol stored in G$ (#) right next to each other to form the bar. The first PRINT statement (?) in line 410 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 80 is necessary to prevent the program from executing line 400 again. This would produce the error message ERROR-16, which would mean that a RETURN statement was encountered without a corresponding GOSUB.

The basic idea shown in Figure 10.5 can be used to produce useful bar graphs of real data, as illustrated in the following 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 bar graphs of 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 symbol defined in line 30 will represent a certain numer of people. We're using a reverse video space (typed with the ATARI key) for G$. It is printed as a blank space in Figure 10.6. In order to determine how many people this should be you must choose a value that will ensure that the longest bar will fit on the screen. The state name starting in column 2 plus a space will use five columns of a screen line. Therefore, the longest possible bar is one 35 spaces long. The maximum population is that of Massachusetts, 5,737,037. Therefore, each symbol G$ must represent more than 5737037/35 = 163915 persons. We will therefore choose each symbol to represent a population of 200,000.

Given a population P, line 70 calculates the number of symbols L to be plotted (that is, the length of the bar). In the equation

$$L = \frac{P}{200000} + 0.5$$

```
10 REM POPULATION BAR GRAPH
15 DIM G$(1),S$(2)
20 N=6
30 G$=" "
35 ? "§"
40 ? " POPULATION OF NEW ENGLAND STATES"
45 ? :?
50 FOR J=1 TO N
60 READ S$,P
70 L=P/200000+0.5:GOSUB 400
80 NEXT J
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 ? S$;" ";:FOR I=1 TO L:? G$;:NEXT I:? :? :RETURN
```

FIGURE 10.6 Program to produce a bar graph of the data in Table 10.1.

the 0.5 will round to the nearest 200,000 persons, since this equation is equivalent to the equation

$$L = \frac{P + 100000}{200000}$$

Note that the number of symbols plotted in the subroutine in line 400 will be equal to the interger part of L.

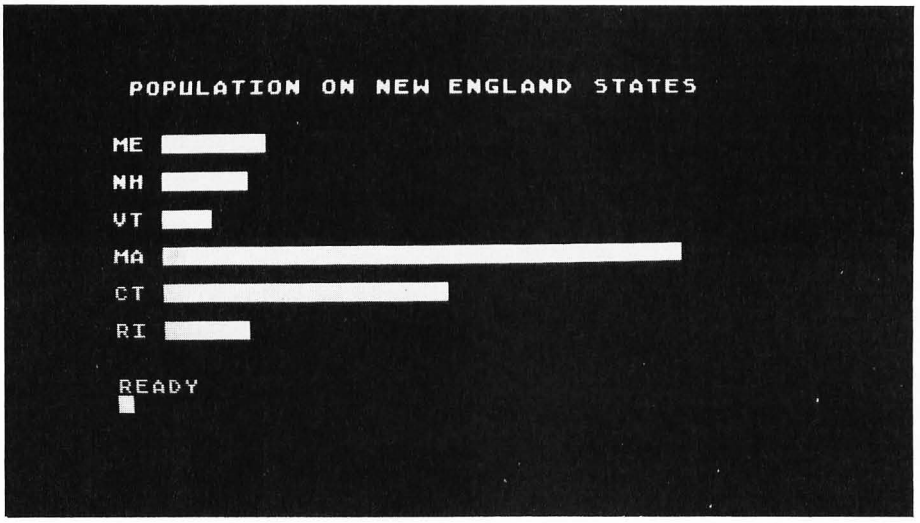
The subroutine in line 400 has been modified to print the state name, stored in S$, to the left of the each bar. Line 35 clears the screen; line 40 prints the title to the graph, and then line 45 skips two lines. The result of running this program is shown in Figure 10.7.

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 any information on the actual values of these populations. We can correct this by adding a scale to the bottom of the graph.

Since each symbol G$ represents a population of 200,000, five symbols will represent one million people. A subroutine that prints such a scale is shown in Figure 10.8. This subroutine is called in line 85 of the revised main program shown in Figure 10.9. The result of executing this new program is shown in Figure 10.10.

FIGURE 10.7 Result of running the program shown in Figure 10.6.



```
600 REM ADD SCALE
610 ? "   +";:FOR I=1 TO 6:? "-----+";:NEXT I:?
620 ? "   ";:FOR I=0 TO 6:? I;"     ";:NEXT I:? :?
630 ? "          MILLIONS OF PEOPLE"
640 RETURN
```

FIGURE 10.8 Subroutine to display scale.

```
10 REM POPULATION BAR GRAPH
15 DIM G$(1),S$(2)
20 N=6
30 G$=" "
35 ? "}"
40 ? " POPULATION OF NEW ENGLAND STATES"
45 ? :?
50 FOR J=1 TO N
60 READ S$,P
70 L=P/200000+0.5:GOSUB 400
80 NEXT J
85 GOSUB 600:REM ADD SCALE
90 END
```

FIGURE 10.9 Revised main program that calls subroutine to add a scale.

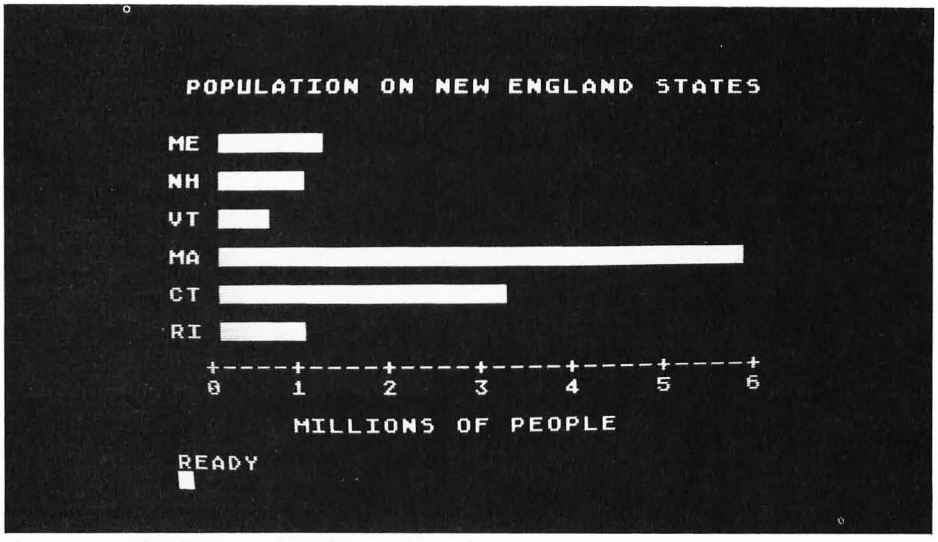


FIGURE 10.10 Result of running the program shown in Figure 10.9.

VERTICAL BAR GRAPHS

In addition to the horizontal bar graphs that have been described, you can draw vertical bars by using low-resolution graphics. As you already know, the statement

PLOT X,Y1:DRAWTO X,Y2

will plot a vertical bar from Y1 to Y2 at the horizontal location X. If Y2 is less than Y1, the bar will be plotted from bottom to top.

Suppose that you want to 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 negative direction. Figure 10.11 shows the ranges of values for which

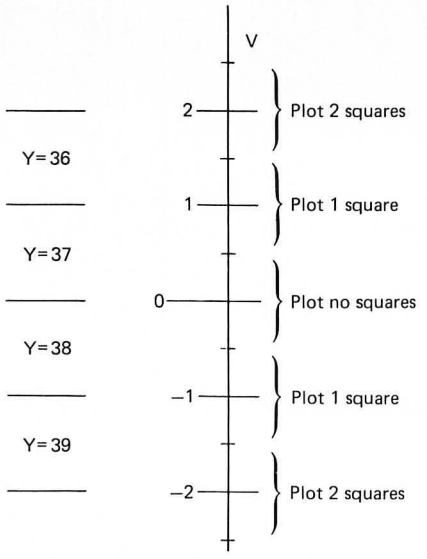


FIGURE 10.11 Screen layout for plotting vertical bar with length proportional to the value V.

various numbers of squares will be plotted. The bottom of row 37 on the screen will define the "zero" value of V. From Figure 10.11 you see that a value of V between 0.5 and 1.5 will result in one square being plotted in row Y = 37. Similarly, a value of V between − 0.5 and − 1.5 will result in one square being plotted in row Y = 38.

A positive bar of length L can be plotted using the statement

PLOT X,Y1:DRAWTO X,Y1−L+1

Note that if L = 1, then a single square will be plotted. The case of L = 0 must be tested for separately in order to plot no square at all.

A negative bar of length L (absolute value) can be plotted using the statement

PLOT X,Y1:DRAWTO X,Y1+L−1

From Figure 10.11 note that Y1 = 37 for a positive bar and Y1 = 38 for a negative bar. Also note that the number of squares to be plotted (that is, the length of the bar) is given by

L = INT(ABS(V) + 0.5)

These ideas are summarized in the algorithm shown in Figure 10.12. This algorithm will plot a vertical bar at position X with a length proportional to V.

A BASIC subroutine that implements this algorithm is shown in Figure 10.13. Lines 540–550 plot the bar for positive values of V and lines 570–580 plot the bar for negative values of V.

In order to test this subroutine, type it in as shown in Figure 10.13. Then enter the low-resolution graphics mode by typing

GR. 5:SETCOLOR 0,0,8:COLOR 1

```
L = INT(ABS(V) + 0.5)
if L = 0
then return
else if V < 0
    then Y1 = 38
        PLOT X,Y1:DRAWTO X,Y1 + L − 1
    else Y1 = 37
        PLOT X,Y1:DRAWTO X,Y1 − L + 1
```

FIGURE 10.12 Algorithm to plot a bar of length proportional to the value V.

FIGURE 10.13 Subroutine to plot a bar of length V.

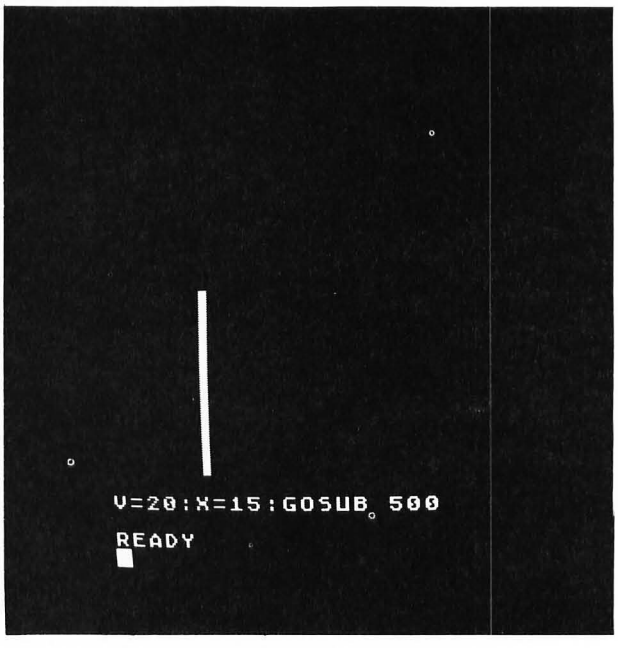
```
500 REM PLOT BAR OF LENGTH V
510 L=INT(ABS(V)+0.5)
520 IF L=0 THEN RETURN
530 IF V<0 THEN 570
540 Y1=37
550 PLOT X,Y1:DRAWTO X,Y1-L+1
560 GOTO 590
570 Y1=38
580 PLOT X,Y1:DRAWTO X,Y1+L-1
590 RETURN
```

Then type

V=20:X=15:GOSUB 500

You should obtain the result shown in Figure 10.14.

FIGURE 10.14 Testing subroutines given in Figure 10.13.



As another example, type

```
GR.5:SETCOLOR 0,0,8:COLOR 1
V=+3.0:X=10:GOSUB 500
```

The line beginning with V=+3.0 is "live" on the screen. This means that if you edit this line (using the CTRL ↑ → keys) 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 500
V=−0.6:X=12:GOSUB 500
V=+1.3:X=13:GOSUB 500
V=−1.3:X=14:GOSUB 500
```

You should obtain the bars shown in Figure 10.15. Try some other values.

This technique of using the immediate mode of execution to test subroutines is a good method because you can make quite a few tests without disturbing the program that you have stored in the computer.

We will now use this subroutine to plot a "multiple" bar graph that will display five-year economic data.

FIGURE 10.15 Further tests of subroutine given in Figure 10.13.

Multiple Bar Graph for the Economy

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. We will use the three colors given in Table 10.3. Note that we can only plot three *different* colors using graphics mode 5. The first and last bar of each group of four will therefore be the same color (orange). The main program for plotting this bar graph is shown in Figure 10.16.

FIGURE 10.16 Main program for plotting economy bar graph.

```
5 REM THE ECONOMY
10 GRAPHICS 5
15 SETCOLOR 0,2,8:REM ORANGE
16 SETCOLOR 1,12,10:REM GREEN
17 SETCOLOR 2,9,4:REM BLUE
20 DATA 4.8,7.8,4.8,2.8
30 DATA 6.8,7.0,5.8,4.5
40 DATA 9.0,6.0,4.9,3.4
50 DATA 13.3,5.8,0.8,-0.6
60 DATA 18.2,6.2,1.7,-0.4
65 X=8
70 FOR J=1 TO 5
80 GOSUB 200:REM PLOT 4 BARS
90 NEXT J
100 GOSUB 400:REM PRINT HEADING
150 GOTO 150
```

Line 10 enters the low-resolution graphics mode 5. Lines 15–17 define the three color registers given in Table 10.3. 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 start-

TABLE 10.2 Economic data*

	1976	*1977*	*1978*	*1979*	*1980*	
Inflation % change in C.P.I.	4.8	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	− 0.6	− 0.4	Projected 1st Q.

*Adpated from data on p. 67 of the March 10, 1980, issue of *Time* Magazine.

TABLE 10.3 Colors used in ecomony bar graph

	Color No.	*Color*	*Color Register*	*Hue No.*	*Luminance*
Inflation	1	Orange	0	2	8
Unemployment	2	Green	1	12	10
Growth	3	Blue	2	9	4
Personal income	1	Orange	0	2	8

ing with 1976. The value of X is initialized to 8 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 that year. This is done in a subroutine starting at line 200. Line 100 calls a subroutine at line 400 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 10.17. 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 orange inflation bar is plotted in lines 230–240 using the subroutine at line 500 shown in Figure 10.18.

FIGURE 10.17 Subroutine to plot the next four bars of the graph.

```
200 REM PLOT NEXT 4 BARS
210 READ I,U,G,M
230 V=2*I:COLOR 1
240 GOSUB 500:REM PLOT BAR
260 V=2*U:COLOR 2
270 GOSUB 500
290 V=2*G:COLOR 3
300 GOSUB 500
320 V=2*M:COLOR 1
330 GOSUB 500
340 X=X+6
350 RETURN
```

Note that the value of V has been equated to twice the inflation value I. This is done because the maximum data value in Table 10.2 is 18.2. Twice this value is 36.4, which will fit on the screen if we plot 36 squares. Lines 260–270 plot the green unemployment bar. Lines 290–300 plot the blue growth bar. Lines 320–330 plot the orange income bar. Line 340 increases the column number X by 6. This will leave a six-column space between each group of four bars.

The subroutine in Figure 10.18 is a modification of the subroutine in Figure 10.13 that plots a double-width bar. Note that the bars plotted in lines 550 and

```
500 REM PLOT BAR OF LENGTH V
510 L=INT(ABS(V)+0.5)
520 IF L=0 THEN RETURN
530 IF V<0 THEN 570
540 Y1=37
545 FOR K=1 TO 2
550 PLOT X,Y1:DRAWTO X,Y1-L+1
552 X=X+1
555 NEXT K
560 GOTO 590
570 Y1=38
575 FOR K=1 TO 2
580 PLOT X,Y1:DRAWTO X,Y1+L-1
582 X=X+1
585 NEXT K
590 RETURN
```

FIGURE 10.18 Subroutine to plot a double-width bar of length V.

580 are each plotted twice next to each other using FOR . . . NEXT loops. Also note that the value of X is incremented by 1 after each bar is plotted. This will make the double bars plotted with the subroutine at line 500 appear adjacent to each other on the screen.

The subroutine to display the years and heading at the bottom of the graph is shown in Figure 10.19. Line 420 prints the five years under the appropriate bar graphs. Line 430 prints the title of the graph and a statement indicating the graph's scale. Lines 440–450 print a legend to explain the four bars. Note that all text in the low-resolution graphics mode must be confined to the bottom four lines on the screen. The statement POKE 752,1 in line 410 will prevent the cursor from being displayed at the end of the heading. Delete this statement to see the difference. We will discuss POKE statements in detail in Chapter 14.

The entire program to plot the economy bar graph is given by the statements in Figures 10.16–10.19. The result of running this program is shown in Figure 10.20.

Although this entire program is relatively long, you can see that by breaking it up 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.

FIGURE 10.19 Subroutine to display heading and scale.

```
400 REM DISPLAY YEARS AND TITLE
410 POKE 752,1
420 ? "  1976    1977    1978    1979    1980"
430 ? "       THE ECONOMY        1 SQUARE=0.5%"
440 ? " BAR 1-INFLATION          BAR 2-GROWTH"
450 ? " BAR 3-UNEMPLOYMENT       BAR 4-INCOME";
460 RETURN
```

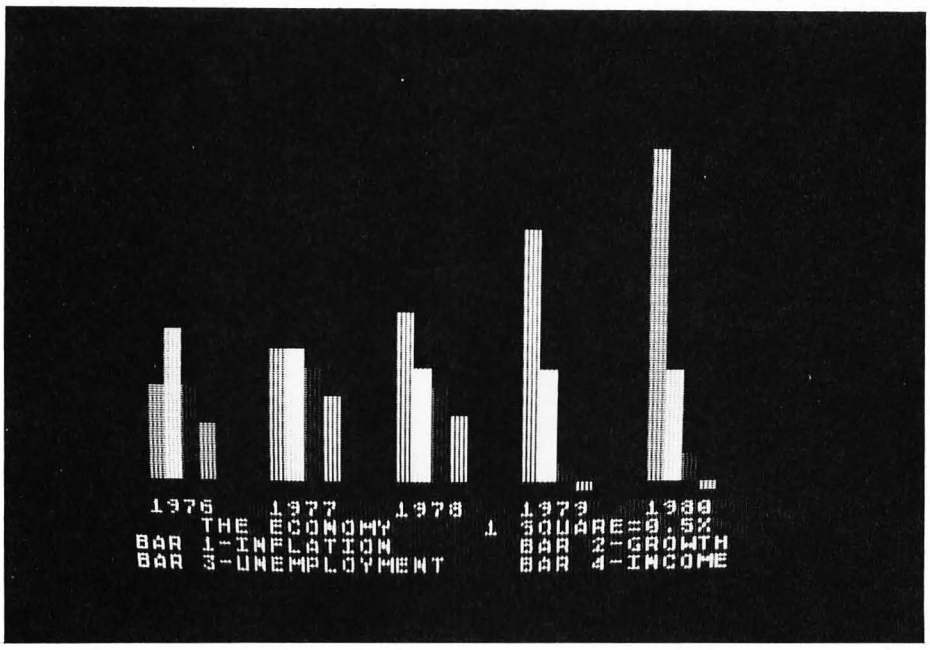


FIGURE 10.20 Bar graph of economic data given in Table 10.2.

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

1. store the data in the table in DATA statements

2. compute the gas mileage in miles per gallon for each fillup and plot the results as a bar graph

3. compute and print out the average miles per gallon for all fillups 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	13,467,400	2,967,909
Canada	22,648,200	3,851,809
China	830,453,000	3,691,502
Great Britain	56,235,500	94,500
India	587,503,700	1,178,995
Japan	108,152,900	143,698
Soviet Union	253,268,300	8,649,489
United States	212,031,000	3,615,122

Write and run a program that will:

1. store the data in the table in DATA statements

2. compute the population density for each nation in persons per square mile

3. plot the results as a bar graph.

11

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$. Remember that strings must be dimensioned with a value equal to or larger than the length of the string. Sometimes it is desirable to identify a collection of memory cells by the same name. Such a collection of memory cells is called an *array;* and the individual memory cells within the array are identified by means of a *subscript.* ATARI BASIC allows numerical arrays but not string arrays. Applications requiring string arrays must be handled in a different manner.

In this chapter you will learn

1. how to represent numerical arrays in BASIC
2. the difference between one-dimensional and multidimensional arrays
3. how to use the DIM statement for numerical arrays
4. how to simulate the use of string arrays in ATARI BASIC
5. how to use arrays when plotting bar graphs
6. how to sort data stored in a one-dimensional array.

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 into the memory cell S$ and each population into the memory cell P. This means that at any one time only one state name and one population were in named memory cells. We printed the state name and plotted a bar with a length proportional to the population. Then we read another state name and population, which replaced the previous ones in S$ and P.

Some programs, however, require that all of the state names and populations be stored in the computer at the same time. We would therefore need 12 different memory cells—six for the state names and six for the populations. This would require 12 different memory cell names. It is convenient to store all of the state populations in an *array* called P. The indi-

vidual memory cells within the array are distinguished by a subscript I. An individual element within the array is sometimes called a subscripted variable, P(I). The array P is shown in Figure 11.1.

P(0)	1124660
P(1)	920610
P(2)	511456
P(3)	5737037
P(4)	3107576
P(5)	947154

FIGURE 11.1 The six subscripted variables P(I) (I = 0,5) contain the state populations.

It would also be convenient to store the six state names in a *string array* S$, such as the one shown in Figure 11.2. Such string arrays are used in many versions of BASIC. However, they are *not* available in ATARI BASIC. Remember that S$(3), for example, is the substring consisting of the characters in S$ starting at location 3. We will see how to combine the six strings in Figure 11.2 into a single long string later in this chapter. First we will consider numerical arrays.

S$(1)	ME
S$(2)	NH
S$(3)	VT
S$(4)	MA
S$(5)	CT
S$(6)	RI

FIGURE 11.2 String arrays are not available in ATARI BASIC.

The DIM Statement

You have already seen in Chapter 1 how to use the DIM statement to assign a length to a string variable. When you are using a numerical array, you must also use a DIM statement to specify the number of elements in the array.

For example, to assign 16 memory cells to the array B you would type

```
DIM B(15)
```

You could then use the 16 memory cells B(0)–B(15). The constant 15 in the DIM statement (this could also be a variable or an expression) represents the upper subscript limit of the array. The lower subscript limit is always assumed to be 0.

You can define more than one array with a single DIM statement. For example, you can write

```
DIM B(15),A(3),C(24)
```

which defines three arrays containing 16, 4, and 25 memory cells, respectively.

Other than using up memory, it does not hurt to reserve more memory locations (by using the DIM statement) than you use in the program. For example, you will reserve 100 memory cells with the statement DIM C(99). In your program you may only refer to the first 20 of these memory cells. This is O.K. However, if you try to refer to C(100), you will obtain the error message ERROR- 9, which means that you have a bad subscript error.

The DIM statement may occur anywhere in the program but it must occur *before* you refer to the corresponding subscripted variable. An array can only be dimensioned once in a program. If you try to dimension it more than once you will also obtain the error messsage ERROR- 9.

The statement CLR will clear, or reinitialize, any dimensioning information. If you want to refer to a string or numerical array after executing CLR, a new DIM statement will be required.

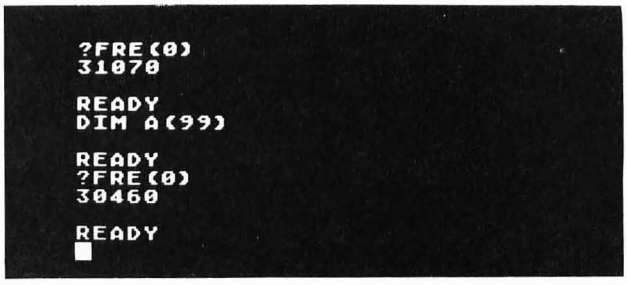
The maximum number of elements in an array will be limited by the amount of memory in your ATARI. If the total amount of memory used by your program, variables, and arrays exceeds the amount of memory in your ATARI you will obtain the error message ERROR- 2, meaning that the computer is out of memory.

Any time you want to know how many bytes of free memory you have left, type

```
?FRE(0)
```

For example, Figure 11.3 shows that an array containing 100 elements uses 610 bytes of memory (6 for each of the 100 elements in the array plus 10 for header information).

FIGURE 11.3 The statement ?FRE(0) will print the number of free bytes of memory left.



Two-Dimensional Arrays

A numerical array that contains a single subscript is called a *one-dimensional array*. An array that contains two subscripts is called a *two-dimensional array* or *matrix*. For example, the DIM statement

```
DIM A(2,3)
```

defines a two-dimensional array containing 12 elements. It can be thought of as two-dimensional matrix containing three rows and four columns, as shown in Figure 11.4.

In the array A(I,J), the first subscript I is the row number in Figure 11.4 and the second subscript J is the column number. Thus for example, in Figure 11.4 the value of A(1,2) is 8 and the value of A(2,1) is 12.

Some versions of BASIC allow arrays to have more than two subscripts. However, ATARI BASIC allows only one- and two-dimensional arrays.

FIGURE 11.4 The array A(I,J), containing 12 elements.

		J			
		0	1	2	3
	0	11	7	0	13
I	1	3	15	8	4
	2	5	12	9	1

SIMULATING STRING ARRAYS

Suppose that you want to store in the computer the six state names shown in Figure 11.2. You could store each one by a different name, such as S1$, S2$, and S3$. Alternatively, you could store them in one long string ST$, such that

```
ST$ = "MENHVTMACTRI"
```

The program shown in Figure 11.5a will do this. The six state names are stored in a DATA statement in line 20. The length of each of these strings is 2. This value is assigned to SLEN in line 30. The FOR . . . NEXT loop in lines 40–80 reads each state name in turn into S$ in line 50 and then adds it to the total state name string ST$ in line 60. The first time through the loop the value of J is 1 and the statement

```
ST$(1,2)=S$
```

will make the first two characters of ST$ ME. The second time through the FOR . . . NEXT loop the value of J will be 3 (incremented by SLEN in line 70) and the string NH will be read into S$ in line 50. Line 60 will then be equivalent to

```
ST$(3,4)=S$
```

so that the third and fourth characters will be NH. This process continues until all six state names have been added (or concatenated) to the string ST$. Note that when the program is run the string ST$ is printed in line 90. The result of running the program is shown in Figure 11.5b.

Once the six state names are stored in the string ST$, the individual names can be extracted by referring to ST$(I,J). For example, ST$(5,6) is VT.

FIGURE 11.5 Program to store six short strings as one long string.

```
10 REM STRING OF STATE NAMES
15 DIM S$(2),ST$(12)
20 DATA ME,NH,VT,MA,CT,RI
30 J=1:SLEN=2
40 FOR I=1 TO 6
50 READ S$
55 ? S$
60 ST$(J,J+SLEN-1)=S$
70 J=J+SLEN
80 NEXT I
90 ? ST$
```

```
RUN
ME
NH
VT
MA
CT
RI
MENHVTMACTRI

READY
```

BAR GRAPHS USING ARRAYS

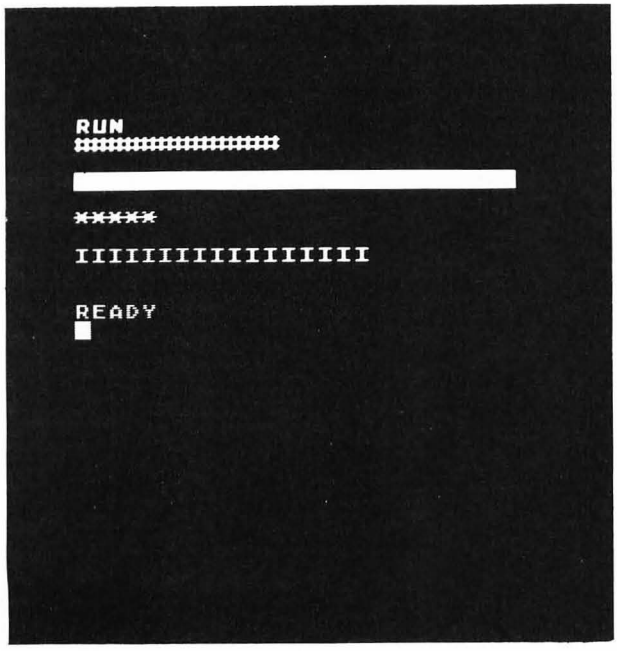
The program shown in Figure 10.5 in Chapter 10 plots four bars of lengths 12, 25, 5, and 17. Review that program and make sure that you understand how it works. Line 400 plots a bar of length L, using the character stored in G$. In this section we will modify this program to plot the same length bars but to use a different character for each bar.

The modified program and its execution are shown in Figure 11.6. Line 15 is a new DATA statement that contains the four characters that will be used for the four bars. (The second character is a reverse video space which is printed as a space in Figure 11.6a.) Line 25 is the DIM statement

```
25 DIM G$(1),GT$(4),L(4)
```

FIGURE 11.6 Bar graph example using arrays.

```
10 REM BAR GRAPH EXAMPLE
15 DATA #, ,*,I
20 DATA 12,25,5,17
25 DIM G$(1),GT$(4),L(4)
30 FOR I=1 TO 4
35 READ G$:GT$(I,I)=G$
40 NEXT I
45 FOR I=1 TO 4:READ L:L(I)=L:NEXT I
50 FOR J=1 TO 4
60 L=L(J):G$=GT$(J,J):GOSUB 400
70 NEXT J
80 END
400 FOR I=1 TO L:? G$;:NEXT I
410 ? :? :RETURN
```



This statement defines a string G$(1) that will contain one symbol and a string GT$(4) that will contain all four symbols. The numerical array L(4) will contain the four lengths. Although this DIM statement defines *five* elements in the array L(I), we will only use the elements L(1)–L(4) and just ignore L(0).

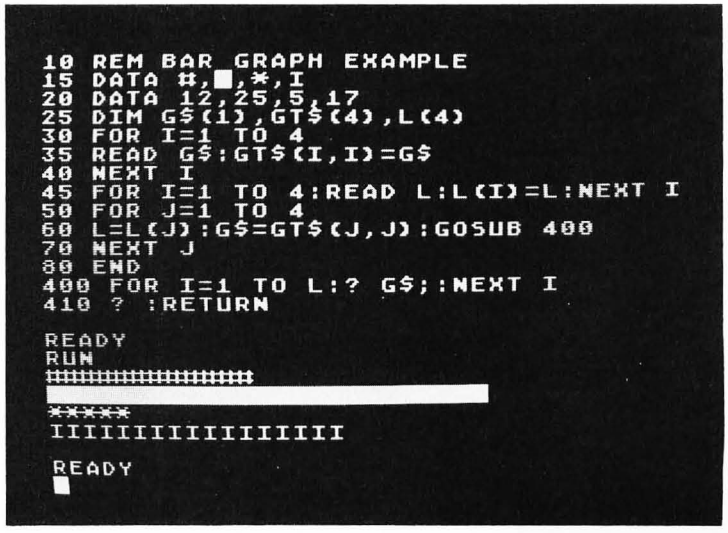
Lines 30–40 read the four characters in the DATA statement on line 15 and concatenate them into the string GT$. Line 45 reads the four values 12, 25, 5, and 17 from line 20 into the four subscripted variables L(1)–L(4), respectively. Note that ATARI BASIC does not allow the READ statement to read data directly into a subscripted variable. Therefore we first READ L and then assign L to L(I).

The loop defined by lines 50–70 plots the four bars. Each time through the loop a new length L(J) and a new character GT$(J,J) are assigned to L and G$ to be plotted in line 400. Note how the subscript J is incremented from 1 to 4 each time it passes through the loop. Also note that the subscripted variable L(J) and the simple variable L are not confused by the ATARI and are treated as separate memory cells.

The four bars in Figure 11.6b can be plotted adjacent to each other by eliminating one of the PRINT statements in line 410, as shown in Figure 11.7.

Suppose that you would like to plot the bars shown in Figure 11.7 in *increasing* order of 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* order. One simple method of sorting an array in increasing order will now be described.

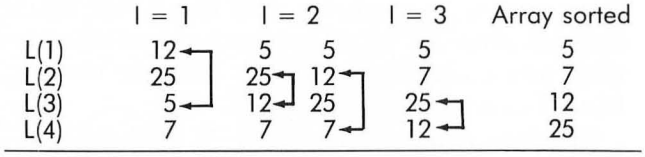
FIGURE 11.7 Plotting the bars adjacent to each other.



Sorting an Array in Increasing Order

Many algorithms have been devised for sorting an array of elements in increasing order. Some are more efficient (that is, they execute faster) than others. Some (not necessarily the same ones) are easier to understand than others. The method of sorting the array L illustrated in Figure 11.8 is fairly easy to understand.

FIGURE 11.8 Sorting an array by moving the smallest succeeding value to location I, I = 1 to N − 1, where N = number of elements in array.



The method begins by comparing the first element in the array (I = 1) with all succeeding elements. Any time a succeeding element is found that is 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 the second element is compared with all succeeding elements and the values interchanged if the succeeding element is smaller than the second one, the next to smallest value will end up 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 increasing order, as shown in Figure 11.8. The algorithm for this procedure is shown in pseudocode in Figure 11.9 Compare Figures 11.8 and 11.9 and make sure you understand how this sorting algorithm works.

FIGURE 11.9 Pseudocode representation of sorting algorithm shown in Figure 11.8.

```
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)
    next J
next I
```

The algorithm shown in Figure 11.9 looks as if it would be 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! Thus, L(I) and L(J) will end up with the same value. It requires *three* statements to interchange the values of L(I) and L(J), as shown in Figure 11.10. 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 which used to be in L(I) can be put in L(J).

FIGURE 11.10 Three statements are required to interchange L(I) and L(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 sorting algorithm shown in Figure 11.9 is written as a BASIC subroutine in Figure 11.11. Note that line 2040 interchanges the values in L(I) and L(J). Add this subroutine to the program shown in Figure 11.7.

FIGURE 11.11 BASIC subroutine to sort array L(I) containing N elements in increasing 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 NEXT J:NEXT I
2060 RETURN
```

FIGURE 11.12 Plotting bar graphs in increasing order using the subroutine in Figure 11.11.

```
10 REM BAR GRAPH EXAMPLE
15 DATA #, ,*,I
20 DATA 12,25,5,17
25 DIM G$(1),GT$(4),L(4)
30 FOR I=1 TO 4
35 READ G$:GT$(I,I)=G$
40 NEXT I
45 FOR I=1 TO 4:READ L:L(I)=L:NEXT I
47 N=4:GOSUB 2000:REM SORT L(I)
50 FOR J=1 TO 4
60 L=L(J):G$=GT$(J,J):GOSUB 400
70 NEXT J
80 END
```

```
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$=GT$(I,I):GT$(I,I)=GT$(J,J):GT$(J,J)=T$
2050 NEXT J:NEXT I
2060 RETURN
```

FIGURE 11.13 Sorting subroutine that interchanges GT$(I,I) and GT$(J,J) each time that L(I) and L(J) are interchanged.

Then add the statement

47 N=4:GOSUB 2000:REM SORT L(I)

to the main program, as shown in Figure 11.12. The result of running this program is also shown in Figure 11.12. Line 47 sets the number of elements in the array L to 4 and then sorts this array by calling the subroutine shown in Figure 11.11.

If you compare Figure 11.12 with Figure 11.7 you will notice that the four characters are plotted in the same relative order. That is, they did not get sorted as the data did. However, it probably makes more sense to associate a particular character with a particular data value (such as inflation or growth); therefore, if the data are rearranged (sorted) the corresponding character should also be rearranged. We can do this by adding the statement

27 DIM T$(1)

to the main program, and the statement

2045 T$=GT$(I,I):GT$(I,I)=
GT$(J,J):GT$(J,J)=T$

to the subroutine given in Figure 11.11, as shown in Figure 11.13. This statement will cause GT$(I,I) and GT$(J,J) to be interchanged each time that L(I) and L(J) are interchanged. This will cause a given data value to "keep" its particular character, as shown in Figure 11.14.

FIGURE 11.14 Plotting bar graphs using the subroutine in Figure 11.13.

```
10 REM BAR GRAPH EXAMPLE
15 DATA #, ,*,I
20 DATA 12,25,5,17
25 DIM G$(1),GT$(4),L(4)
27 DIM T$(1)
30 FOR I=1 TO 4
35 READ G$:GT$(I,I)=G$
40 NEXT I
45 FOR I=1 TO 4:READ L:L(I)=L:NEXT I
47 N=4:GOSUB 2000:REM SORT L(I)
50 FOR J=1 TO 4
60 L=L(J):G$=GT$(J,J):GOSUB 400
70 NEXT J
80 END
400 FOR I=1 TO L:? G$;:NEXT I
410 ? :RETURN
```

Sorting an Array in Decreasing Order

The subroutine in Figure 11.13 can easily be modified to sort the array L(I) in decreasing order rather than increasing order by changing line 2030 to

2030 IF L(I)>=L(J) THEN 2050

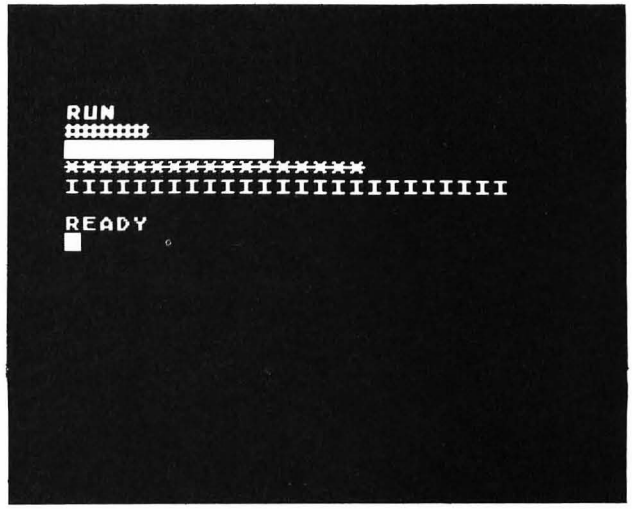
as shown in Figure 11.15. Running the main program with this subroutine will produce the result shown in Figure 11.16.

EXERCISE 11.1

Write a program that

1. stores the six New England states in the string ST$ and their populations in the array P(I), as shown in Figure 11.1

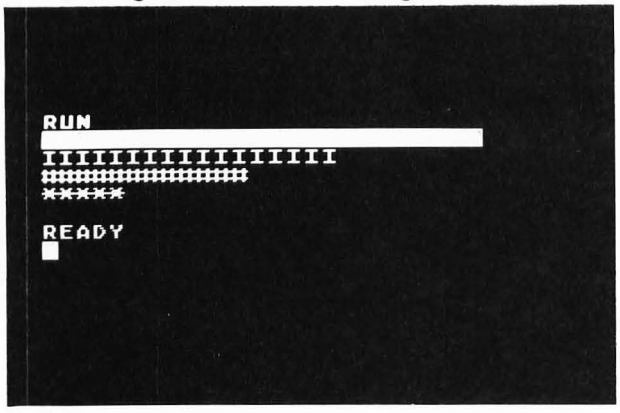
2. plots a vertical bar graph of the populations using low-resolution graphics



```
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$=GT$(I,I):GT$(I,I)=GT$(J,J):GT$(J,J)=T$
2050 NEXT J:NEXT I
2060 RETURN
```

FIGURE 11.15 Subroutine to sort array L(I) containing N elements in *decreasing* order.

FIGURE 11.16 Plotting bar graphs in decreasing order using the subroutine in Figure 11.15.



3. sorts the populations in increasing order

4. plots a second bar graph (after pressing key S) with the populations in increasing order.

EXERCISE 11.2

Write a program that

1. stores N test scores in DATA statements, with the value of N stored as the first entry of the first data statement

2. reads the test scores into an array S(I)

3. computes and prints out the average of the N test scores.

EXERCISE 11.3

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 11.2 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 11.4

The following weights are those of a group of males and females. Write a computer program that will compute and print out the means and standard deviations of the two groups of weights. Modify the program to compute and print the mean and standard deviation of all weights (both male and female). (See Exercise 11.3.)

Weights	
Male	*Female*
200	103
185	105
185	112
125	102
140	160
195	120
190	115
155	130
185	140
140	118
138	
205	
159	
230	
150	
140	
170	
145	
169	
215	

EXERCISE 11.5

A person makes the following monthly deposits in a savings account paying 5 percent 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 third year. Write a computer 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 11.6

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.

12

MORE ABOUT STRINGS

You have learned in previous chapters of this book that memory cells with names like A$ and C3$ contain *strings*. The dollar sign, $, is used in BASIC to identify string-related names. ATARI BASIC requires you to dimension the length of a string variable and then makes it easy for you to manipulate strings by using substrings such as A$(I,J). In this chapter you will learn

1. to use the length function LEN and manipulate substrings
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.

MANIPULATING STRINGS

The string A$(I,J) is used to extract some portion of a string. The function LEN is used to determine the length of a string.

LEN

The function

```
LEN(A$)
```

is equal to the length of the string A$. Note that it is a *numerical* value (not a string). For example, if A$ = "ABCDE" then the value of LEN(A$) is 5. To verify this, type

```
DIM A$(5)
A$="ABCDE"
?LEN(A$)
```

as shown in Figure 12.1.

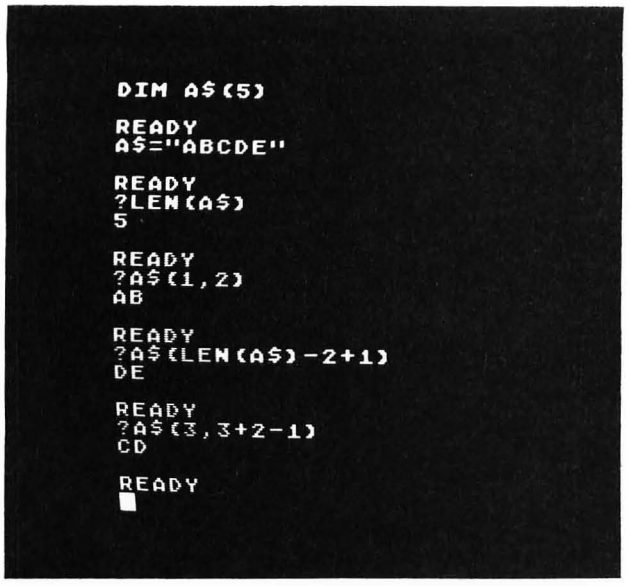


FIGURE 12.1 Using substrings and the string functions.

Substrings

Some popular versions of BASIC (for the Apple II, PET, and TRS-80) have the string functions LEFT$, RIGHT$, and MID$. ATARI BASIC can achieve the same result by using the substring A$(I,J).

For example, the function

LEFT$(A$,I)

is a string that contains the leftmost I characters of the string A$. For example, if A$ = "ABCDE", then LEFT$(A$,2) will be the string "AB". In ATARI BASIC this is the same as A$(1,I). To verify this, assuming that you have already set A$ = "ABCDE", type

?A$(1,2)

as shown in Figure 12.1.

The function

RIGHT$(A$,I)

is a string that contains the rightmost I characters of the string A$. For example, if A$ = "ABCDE", then RIGHT$(A$,2) will be the string "DE". In ATARI BASIC this is the same as

A$(LEN(A$)−I+1)

To verify this, type

?A$(LEN(A$)−2+1)

as shown in Figure 12.1.

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". In ATARI BASIC this is the same as

A$(I,I+J−1)

To verify this, type

?A$(3,3+2−1)

as shown in Figure 12.1.

THE NUMERIC/STRING FUNCTIONS VAL AND STR$

It is important that you 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 containing buildings, roads, people, and so on. "BOSTON" is a six-letter word that is the name of a city. Similarly, 456 is a number that you can add to other numbers. The string "456" is just the three characters 4, 5, and 6 sitting next to each other. 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, VAL(A$) will produce the ERROR- 18 messsage.

As an example of using the VAL function, clear the screen and type

```
CLR
DIM A$(3)
A$="456"
```

FIGURE 12.2 Using the numeric/string function VAL.

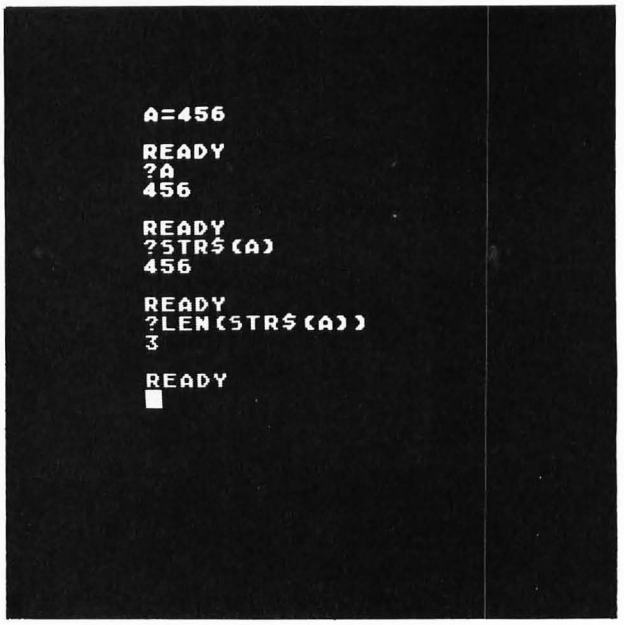


FIGURE 12.3 Using the numeric/string function STR$.

```
?A$
?VAL(A$)
```

as shown in Figure 12.2. It looks as if both PRINT statements print the same value 456. However, in order to see the difference between VAL(A$) and A$, type

```
?VAL(A$)+10
?A$+10
```

as shown in Figure 12.2. Note that the number VAL(A$) can be added to 10, whereas trying to add the string A$ to the number 10 will produce an error.

STR$

The function

```
STR$(A)
```

is the string equivalent of the numerical value A. As an example of how to use the STR$ function, clear the screen and type

```
A=456
?A
?STR$(A)
```

as shown in Figure 12.3. Note that both print statements print the number 456. However, STR$(A) is actually a string containing the three characters 4, 5, and 6. To verify this, type

```
?LEN(STR$(A))
```

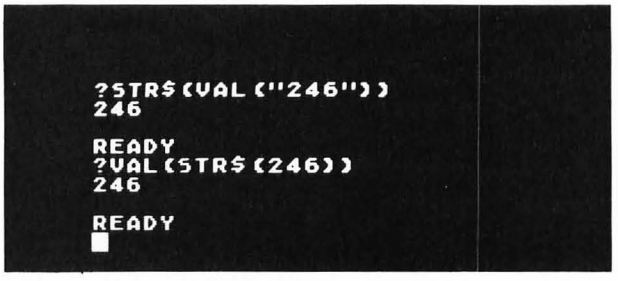
as shown in Figure 12.3.

The functions STR$ and VAL are reciprocal functions, as you can verify by typing

```
?STR$(VAL("246"))
?VAL(STR$(246))
```

as shown in Figure 12.4.

FIGURE 12.4 STR$ and VAL are reciprocal functions.



THE ASCII CODE FUNCTIONS ASC AND CHR$

The name 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 or for sending data from a terminal to a computer. ATARI uses an expanded ASCII code, called an ATARI

ASCII, or ATASCII code. The BASIC function ASC can be used to find the ATASCII number associated with any character, and the function CHR$ can be used to find the character associated with any ATASCII number.

ASC

The function

ASC(A$)

is equal to the ATASCII code of the first character in the string A$. To find some ATASCII codes, clear the screen and type

```
?ASC("A")
?ASC("?")
?ASC("ABC")
?ASC("*")
?ASC("7")
```

as shown in Figure 12.5. Letters, digits, and special character keys all have ATASCII numbers. Note that the ATASCII number for a digit is different from the digit itself (55 is the ATASCII code for 7). Also note that the function ASC("ABC") is the ATASCII code of the first character A.

FIGURE 12.5 Examples of ATASCII codes of ATARI characters.

```
?ASC("A")
65

READY
?ASC("?")
63

READY
?ASC("ABC")
65

READY
?ASC("*")
42

READY
?ASC("7")
55

READY
```

CHR$

If you know the ATASCII code of a character you can generate the string of that character using the function

CHR$(A)

where A is the ATASCII code of the character.

The program shown in Figure 12.6 will display the ATASCII code for any key pressed. Typing the exclamation point (!) will terminate the program. Line 20 is the OPEN statement

OPEN #1,4,0,"K:"

which must be executed before the statement

GET #1,A

in line 30 can be executed. This statement will wait for a key to be pressed and then assign the ATASCII code of the key pressed to the variable A. Line 35 checks to see if this is equal to the ATASCII code for an exclamation point. If it is (the ! key was pressed), the program branches to line 60 where the statement CLOSE #1 closes the file or input/output (I/O) device (keyboard) opened in line 20.

If any key other than the exclamation point is pressed, line 40 will print the character corresponding to the key pressed, CHR$(A), followed by the ATASCII code A. A sample run of this program is shown in Figure 12.7. A list of ATASCII codes is given in Appendix B.

FIGURE 12.6 Program to find the ATASCII codes of each key pressed.

```
10 REM ASCII CODES
20 OPEN #1,4,0,"K:"
30 GET #1,A
35 IF A=ASC("!") THEN 60
40 ? CHR$(A);" ";A,
50 GOTO 30
60 CLOSE #1
70 ? "PROGRAM DONE"
```

FIGURE 12.7 Sample run of program given in Figure 12.6.

```
RUN
A 65        S 83        D 68        F 70
  Q 81        W 87        E 69        R 82
    1 49        2 50        3 51        4 52
      ; 59        + 43        - 45        =
61          Z 90        X 88        C 67
V 86        PROGRAM DONE

READY
```

PRINTING DOLLARS AND CENTS

Many practical programs involve money and require you to display dollars and cents on the screen. This is not as easy as it may seem. First of all, if you compute some monetary value such as interest in 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 scheme, type

```
A=208.4978
A1=A+.005
?A1
?INT(A1*100)/100
```

as shown in Figure 12.8. Note that although this scheme rounded to the nearest cent, the ATARI does not display trailing 0s. Therefore, 50 cents is printed as .5. This would look strange if you printed the amount of a check this way.

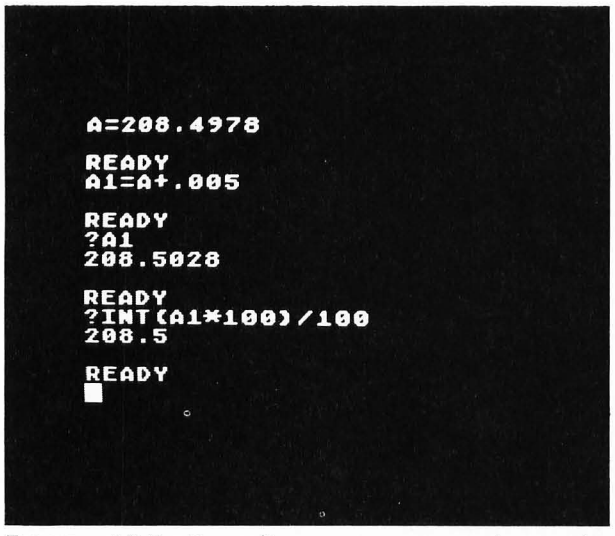


FIGURE 12.8 Rounding a monetary value to the nearest cent.

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

```
DIM A2$(9),A3$(9)
A2=INT(A1)
?A2
A2$=STR$(A2)
?A2$
```

as shown in Figure 12.9. Note that A2 is the dollar value and A2$ is the string representation of this value.

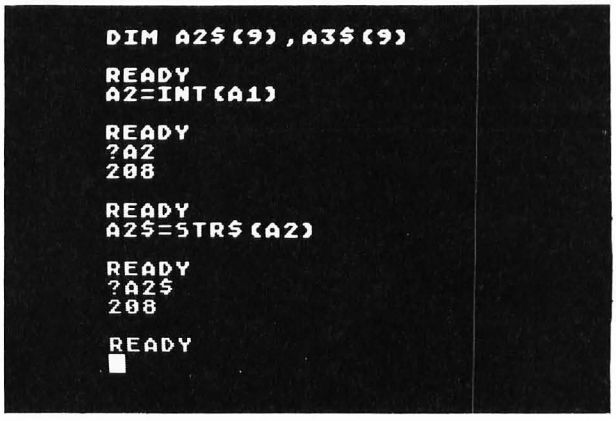


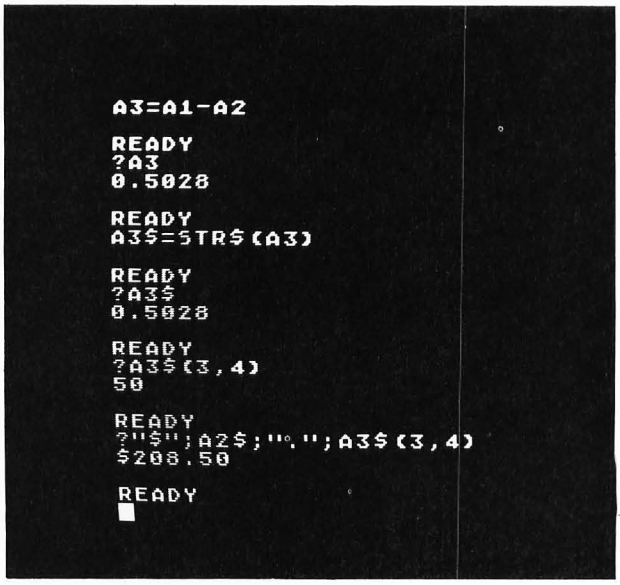
FIGURE 12.9 A2$ is a string representation of the dollar amount.

In order to obtain a string representation of the cents value, type

```
A3=A1-A2
?A3
A3$=STR$(A3)
?A3$
?A3$(3,4)
```

as shown in Figure 12.10. 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 two characters are 0.).

FIGURE 12.10 A3$(3,4) is a string representation of the cents amount.



The total dollars and cents can be displayed by typing

```
?"$";A2$;".";A3$(3,4)
```

which will display

```
$208.50
 |  |
 |  \
A2$ A3$(3,4)
```

as shown at the bottom of Figure 12.10.

The statements shown in Figures 12.8, 12.9, and 12.10 can be combined to form the subroutine shown in Figure 12.11. This subroutine should print the value of A in the form $XX.YY. For the first value of A shown in Figure 12.10, the subroutine works well. However, for a value of A = 159.996 the subroutine prints $160.0E. The problem can be found by looking at the value of A3 as shown in Figure 12.11. If the fractional part of A1 (the cents value A3) is less than 0.01, A3 will be stored in scientific notation. This really messes things up because now the third and fourth characters in STR$(A3) are "0E" rather than "00". The subroutine shown in Figure 12.11 can be fixed by adding the statement

```
925 IF A3<.01 THEN A3$="0000":
GOTO 940
```

as shown in Figure 12.12. Note that this modified subroutine prints the correct dollars and cents values for all of the examples shown.

The last example shown in Figure 12.12 rounds 999999.999 to $1000000.00. When writing a check for this value (or any value over $1000.00) it would look better and make the value easier to read if you included the commas in the dollar amount. A method of doing this will now be explained.

FIGURE 12.11 This subroutine for displaying dollars and cents will not work for cents values less than 0.01.

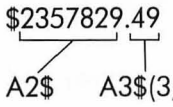
```
900 REM PRINT A AS $XX.YY
910 A1=A+5.0E-03:A2=INT(A1):A3=A1-A2
920 A2$=STR$(A2)
930 A3$=STR$(A3)
940 ? "$";A2$;".";A3$(3,4)
950 RETURN

READY
DIM A2$(9),A3$(9)

READY
A=208.4978:GOSUB 900
$208.50

READY
A=159.996:GOSUB 900
$160.0E

READY
?A3
1.0E-03

READY
```

FIGURE 12.12 Modified subroutine that displays correct dollars and cents value.

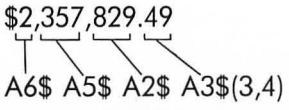
```
LIST

900 REM PRINT A AS $XX.YY
910 A1=A+5.0E-03:A2=INT(A1):A3=A1-A2
920 A2$=STR$(A2)
925 IF A3<0.01 THEN A3$="0000":GOTO 94
0
930 A3$=STR$(A3)
940 ? "$";A2$;".";A3$(3,4)
950 RETURN

READY
DIM A2$(9),A3$(9)

READY
A=159.996:GOSUB 900
$160.00

READY
A=999999.999:GOSUB 900
$1000000.00

READY
```

Adding Commas to the Dollar Amount

Suppose that you want to add commas to the value



First of all, the largest dollar value that our subroutine can handle is 9999999999; after this value the ATARI will store A1 and A2 in scientific notation. Actually, because the ATARI does not keep more than 10 digits of precision when storing numbers, to get the correct cents you should limit the dollar values to 99,999,999.99. Therefore, at most we need to insert two commas. We will therefore divide the string A2$ into the three substrings A4$, A5$, and A6$ as follows:



That is, if L = LEN(A2$), then

$$A4\$ = \begin{cases} A2\$ & (L <= 3) \\ A2\$(L-2,L) & (L > 3) \end{cases}$$

$$A5\$ = \begin{cases} A2\$(1,L-3) & (L <= 6) \\ A2\$(L-5,L-3) & (L > 6) \end{cases}$$

$$A6\$ = A2\$(1,L-6) \quad (L > 6)$$

The algorithm for adding the commas will then be

if L < = 3
then print $A4$.A3$(3,4)
else if L < = 6
    then print $A5$,A4$.A3$(3,4)
    else print $A6$,A5$,A4$.A3$(3,4)

Figure 12.13a shows how this algorithm can be added to the subroutine shown in Figure 12.12. Lines 940–975 implement the algorithm described here. Two examples using this subroutine are shown in Figure 12.13b.

FIGURE 12.13 Subroutine that includes commas when displaying dollars and cents.

```
900 REM PRINT A AS $XX.YY
910 A1=A+5.0E-03:A2=INT(A1):A3=A1-A2
920 A2$=STR$(A2)
925 IF A3<0.01 THEN A3$="0000":GOTO 940
930 A3$=STR$(A3)
940 L=LEN(A2$)
945 ? "$";
950 IF L<=3 THEN A4$=A2$:GOTO 975
955 IF L<=6 THEN A5$=A2$(1,L-3):GOTO 967
960 A6$=A2$(1,L-6):? A6$;",";
965 A5$=A2$(L-5,L-3)
967 ? A5$;",";
970 A4$=A2$(L-2,L)
975 ? A4$;".";A3$(3,4)
980 RETURN
```

```
DIM A2$(9),A3$(9),A4$(3),A5$(3),A6$(3)

READY
A=999999.999:GOSUB 900
$1,000,000.00

READY
A=2357829.49:GOSUB 900
$2,357,829.49

READY
A=32576.248:GOSUB 900
$32,576.25

READY
```

PLAYING CARDS

As another example of how to use 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 12.14. For example, the seven of hearts is number 33 and the jack of diamonds is number 24.

The value of a card (A–K) has a value number V and the four suits have a suit number S, as defined in Figure 12.14.

FIGURE 12.14 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 No. S	1	2	3	4	

It is usually easier to use the card number C as much as possible in a program to distinguish cards and then 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

$$S = INT((C - 1)/13) + 1$$

You should verify this by trying some examples from Figure 12.14. For example, if C = 26 (king of diamonds), then

$$S = INT(25/13) + 1$$
$$= 1 + 1 = 2$$

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). This array can be initialized with the following statements:

```
DIM C(52)
FOR I=1 to 52:C(I)=I:NEXTI
```

Thus, for example, C(47) = 47 and represents the eight of spades.

Suppose that you want to display the 19th 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, from Figure 12.14 the card is the six of diamonds. To display this value, define the two strings V$ and S$ shown in Figure 12.15. The graphic symbols for a club, diamond, heart, and spade are printed using the CTRL key as indicated in Figure 12.15. Note that the position of each value character in V$ corresponds to the appropriate value number V in Figure 12.14. Therefore, the single value character V1$ corresponding to the value number V is given by

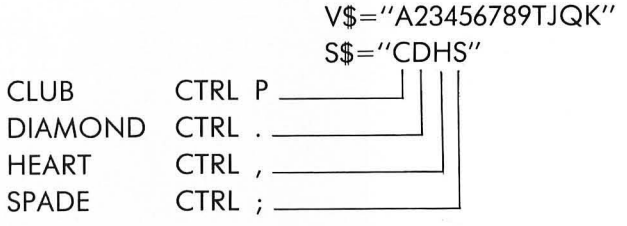
$$V1\$ = V\$(V,V)$$

Similarly, the position of each suit character in S$ corresponds to the appropriate suit number S in Figure 12.14. Therefore, the single suit symbol S1$ corresponding to the suit number S is given by

$$S1\$ = S\$(S,S)$$

These ideas are incorporated in the two subroutines shown in Figure 12.16. 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.

FIGURE 12.15 Definition of the value string V$ and the suit string S$.



```
3000 REM PLAYING CARD SETUP
3010 DIM C(52),V$(13),S$(4),V1$(1),S1$(1)
3020 FOR I=1 TO 52:C(I)=I:NEXT I
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$=V$(V,V)
3140 S1$=S$(S,S)
3150 RETURN
```

FIGURE 12.16 Subroutines to set up deck (line 3000) and pick card at location P (line 3100).

The subroutine given by lines 3100–3150 in Figure 12.16 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 card number C(P). Lines 3110–3120 define the suit number S and value number V using the formulas that have been given here. Lines 3130–3140 find the single character strings V1$ and S1$.

In order to test these subroutines, type

```
GOSUB 3000
P=33:GOSUB 3100:?V1$;S1$
P=52:GOSUB 3100:?V1$;S1$
```

as shown in Figure 12.17. Note that card number 33 is the seven of hearts and card number 52 is the king of spades, as shown in Figure 12.14.

You can display the entire deck by running the program shown in Figure 12.18. Line 20 sets up the deck. Line 25 clears up to 10 tabs and line 27 sets three new tab positions. 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 values and suit symbols in line 50. Note that the PRINT statement in line 50 ends with a tab. The result of running this program is shown in Figure 12.19. Note that the cards are printed in the order shown in Figure 12.14. In order to print them in a random order you must first shuffle the deck.

FIGURE 12.17 Testing the subroutines given in Figure 12.16.

```
GOSUB 3000

READY
P=33:GOSUB 3100:?V1$;S1$
7♥

READY
P=52:GOSUB 3100:?V1$;S1$
K♠

READY
```

FIGURE 12.18 Program to display entire deck of cards.

```
LIST10,100

10 REM DISPLAY DECK
20 GOSUB 3000:REM SETUP DECK
25 ? "                    "
27 ? "        →       →       →"
30 FOR P=1 TO 52
40 GOSUB 3100:REM GET NEXT CARD
50 ? V1$;S1$;"→";:REM DISPLAY CARD
60 NEXT P
100 END

READY
```

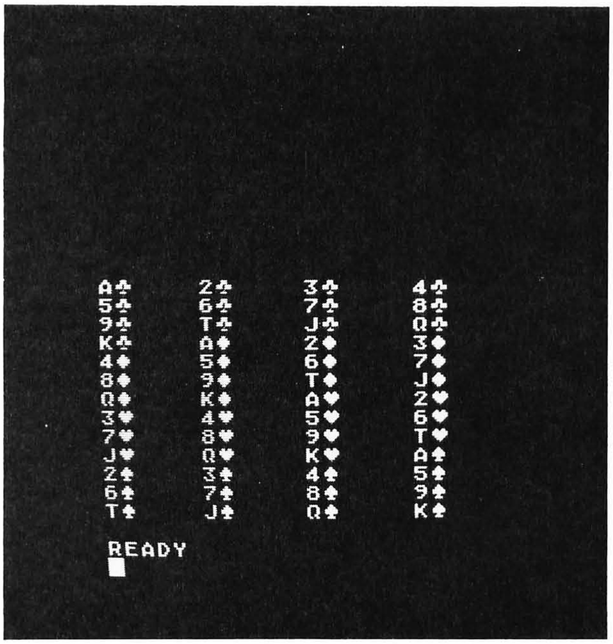


FIGURE 12.19 Result of running program shown in Figure 12.18.

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 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) in turn with another element selected at random.

Recalling that RND(0) is a random number with a value greater than 0 and less than 1, then

$$J = INT(52 * RND(0) + 1)$$

will be a random integer between 1 and 52.

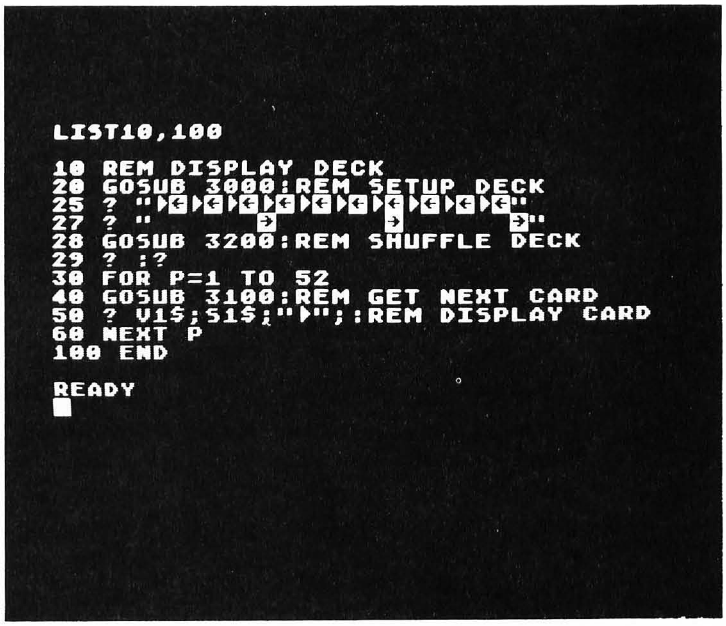
FIGURE 12.20 Subroutine to shuffle a deck of cards.

```
3200 REM SHUFFLE DECK
3210 ? "S H U F F L I N G";
3220 FOR I=1 TO 52
3230 J=INT(52*RND(0)+1)
3240 T=C(I):C(I)=C(J):C(J)=T
3250 NEXT I
3260 RETURN
```

A subroutine that will shuffle the deck while displaying the word "SHUFFLING" is shown in Figure 12.20. The FOR . . . NEXT loop in lines 3220–3250 corresponds to the *for . . . next* loop given here. Line 3230 finds a random number J between 1 and 52. Line 3240 interchanges C(I) and C(J).

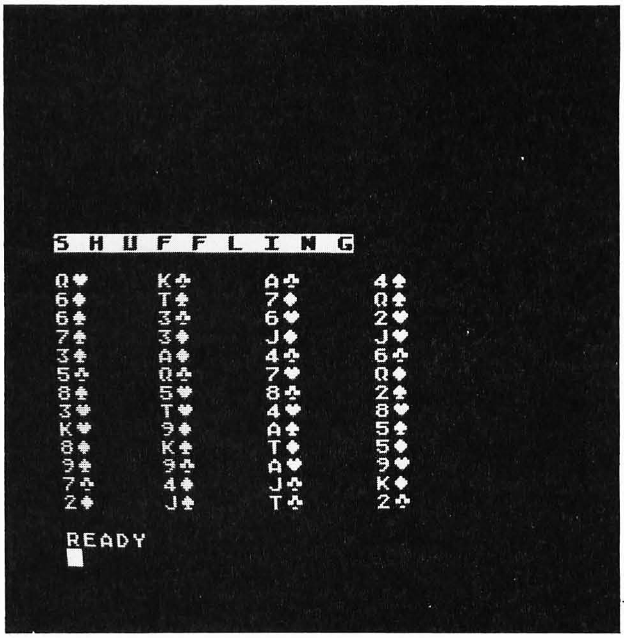
Add lines 28 and 29 shown in Figure 12.21 to the main program given in Figure 12.18. This new program will shuffle the deck and then display all of the cards. A sample run is shown in Figure 12.22.

FIGURE 12.21 Main program to display shuffled deck of cards.



```
LIST10,100

10 REM DISPLAY DECK
20 GOSUB 3000:REM SETUP DECK
25 ? "▶←▶←▶←▶←▶←▶←▶←▶←▶←▶←"
27 ? "      →      →      →"
28 GOSUB 3200:REM SHUFFLE DECK
29 ? :?
30 FOR P=1 TO 52
40 GOSUB 3100:REM GET NEXT CARD
50 ? V1$;S1$;"▶";:REM DISPLAY CARD
60 NEXT P
100 END

READY
```

FIGURE 12.22 Sample run of program shown in Figure 12.21.



```
10 REM DEAL HAND OF CARDS
20 GOSUB 3000:REM SETUP DECK
25 ? ""
27 ? "                                                    "
28 GOSUB 3200:REM SHUFFLE DECK
29 ? :?
30 ? "ENTER NUMBER OF PLAYERS ";
35 INPUT NP
40 ? "ENTER NUMBER OF CARDS PER HAND ";
45 INPUT NC
50 P=1:?
60 FOR I=1 TO NC
70 FOR J=1 TO NP
80 GOSUB 3100:REM DEAL NEXT CARD
90 ? V1$;S1$;"";
100 P=P+1
110 NEXT J
120 ?
130 NEXT I
140 END
```

FIGURE 12.23 Program to deal a hand of cards.

Dealing a Hand of Cards

The program shown in Figure 12.21 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 to each player can be displayed on a single line with the statements

```
 50 P=1:?
 70 FOR J=1 TO NP
 80 GOSUB 3100:REM DEAL NEXT CARD
 90 ? V1$;S1$;"tab";
100 P=P+1
110 NEXT J
```

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 the statements

```
 60 FOR I=1 TO NC
120 ?
130 NEXT I
```

FIGURE 12.24 Sample runs of program shown in Figure 12.23.

```
RUN


S H U F F L I N G
ENTER NUMBER OF PLAYERS ?2
ENTER NUMBER OF CARDS PER HAND ?6

3♣  4♠
J♣  5♣
9♦  A♥
A♣  A♠
6♦  Q♣
8♦  T♠

READY
```

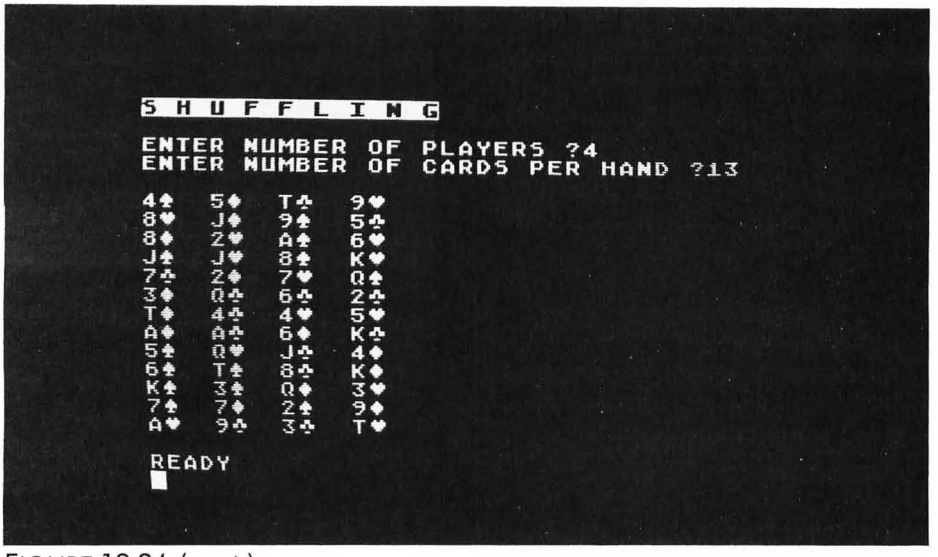


FIGURE 12.24 (cont.)

as shown in Figure 12.23. In this program lines 30–45 allow the user to input the values NP and NC. Line 50 points to the top card of the deck and skips a line. Lines 60–130 make up the outer FOR . . . NEXT loop that prints NC rows of cards. Lines 70–110 make up the inner FOR . . . NEXT loop that deals NP cards and displays them on one line. Line 25 clears the tab position (see Figure 12.21). Eight tab positions have been set up in line 27. 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 12.24. The second example shown in Figure 12.24 could be a bridge hand. It would be nice if you could sort each hand by suit. This is easier to do than you may think.

Sorting Hand by Suit

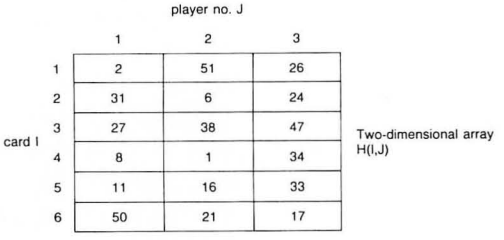
Suppose that a hand contains the cards shown in Figure 12.25a, where the card number for each hand is also given (see Figure 12.14). If the card numbers are sorted in increasing order, the cards will be sorted in increasing order by suit, as shown in Figure 12.25b. This illustrates the advantage of using card numbers to represent playing cards inside the computer.

In order to sort a hand we will therefore 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, or matrix, H(I,J), in which each column will contain the card numbers for a different player, as shown in Figure 12.26. To sort all hands we will need to sort each column in increasing order.

FIGURE 12.25 A hand of cards can be sorted by suit by sorting the card numbers in increasing 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)

FIGURE 12.26 Each column of the two-dimensional array H(I,J) contains the card numbers for one player.



The array H(I,J) needs to contain NC rows (number of cards per hand) and NP columns (number of players). Since we don't know what these values are until lines 35–45 in Figure 12.23 are executed, we will add the following dimension statement at line 47:

```
47 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 12.27. Note that this statement is inside the two nested FOR . . . NEXT loops and will be filled up one row at a time. In Figure 12.27 we have added the one additional statement

```
135 GOSUB 200:REM DISPLAY
SORTED HAND
```

where we will hide everything that we haven't figured out how to do yet!

This subroutine at line 200 will have to sort each column in H(I,J) in increasing order and then display the corresponding cards. This subroutine is shown in Figure 12.28. 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 sort each column in H(I,J). In the interest of putting off, as long as possible, what we haven't figured out how to do, we will just let the subroutine at line 2000 do this. The nested FOR . . . NEXT loops in lines 220–260 are similar to the ones in lines 60–130 in Figure 12.27 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 12.27 where we were incrementing P each time it passed through the loop. However, in Figure 12.28 we don't know P but we *do* know the card number directly—it is just 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 12.28. Note that we must set P = 0 so that the subroutine at line 3100 shown in Figure 12.16 will use C(0), which will be equivalent to using H(I,J).

We're finally to the point where we must figure out how to sort the columns of H(I,J) in increasing order. If you go back and study the sorting algorithm that we developed in the last chapter, given in Figure 11.10, 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 12.29. The BASIC implementation of this algorithm is written as a subroutine in Figure 12.30.

FIGURE 12.28 Subroutine to display the sorted hands of cards.

```
200 REM DISPLAY SORTED HAND
205 ? :? "SORTING":?
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)
245 GOSUB 3100:REM NEXT CARD
250 ? V1$;S1$;"";
260 NEXT J:? :NEXT I:RETURN
```

FIGURE 12.27 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 ? ""
27 ? "                                        "
28 GOSUB 3200:REM SHUFFLE DECK
29 ? :?
30 ? "ENTER NUMBER OF PLAYERS ";
35 INPUT NP
40 ? "ENTER NUMBER OF CARDS PER HAND ";
45 INPUT NC
47 DIM H(NC,NP)
50 P=1:?
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 ? V1$;S1$;"";
100 P=P+1
110 NEXT J
120 ?
130 NEXT I
135 GOSUB 200:REM DISPLAY SORTED HAND
140 END
```

```
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)
        next K
    next I
next J
```

FIGURE 12.29 Algorithm for sorting each column of H(I,J) in increasing order.

FIGURE 12.30 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 NEXT K:NEXT I:NEXT J:RETURN
```

We have now written all of the subroutines needed to run the program shown in Figure 12.27. A sample run is shown in Figure 12.31. Note that each hand is sorted by suit with the suits displayed in the order clubs, diamonds, hearts, and spades.

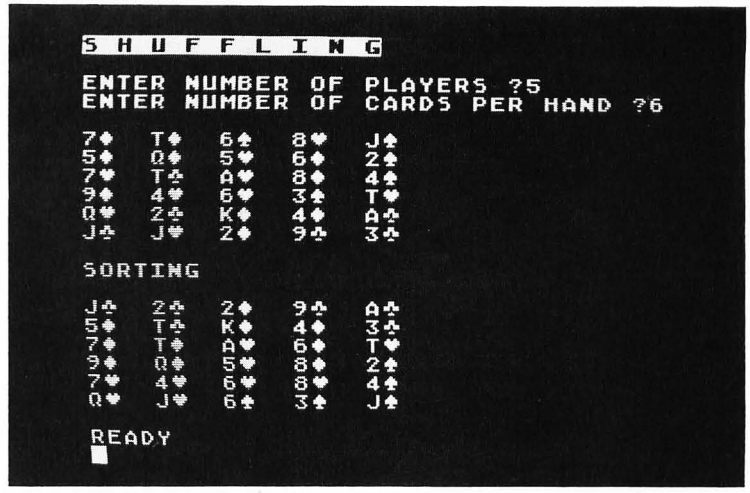


FIGURE 12.31 Sample run of the program shown in Figure 12.27.

EXERCISE 12.1

Write a program that will input a string A$ and a substring B$ 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.

EXERCISE 12.2

Modify the program in Exercise 12.1 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.

13

LEARNING TO USE HIGH-RESOLUTION GRAPHICS

The low-resolution graphics mode 5 was described in Chapter 7 and you have used it in many of your programs in earlier chapters. The ATARI also has other graphics modes that allow you to plot figures on the screen with considerably more detail. In this chapter you will learn

1. the graphics modes available in ATARI BASIC
2. how to use the high-resolution graphics modes on the ATARI
3. how to plot figures in a dot-to-dot fashion by storing the coordinates of the vertices in DATA statements
4. how to draw figures of varying size
5. how to plot figures at different locations on the screen
6. how to plot figures whose coordinates can be calculated.

ATARI GRAPHICS MODES

In Chapter 7 you learned to use graphics mode 5. This was a low-resolution mode in which the screen was divided into an 80×40 grid with four lines of text at the bottom of the screen. This four-line text window can be changed to an additional graphics area by adding 16 to the mode number in the GRAPHICS statement. Thus, the statement

```
GR. 5+16
```

will enter graphics mode 5 with a full-screen 80×48 graphics resolution. If you add 32 to the mode number in the graphics command, the graphics mode will be entered without clearing the screen.

ATARI BASIC will allow you to access nine different graphics modes. Three of these are actually text modes and six are real graphics modes. These nine modes are summarized in Table 13.1. You are already familiar with the text mode GR. 0 and the graphics mode GR. 5.

We will study the text modes GR. 1 and GR. 2 in Chapter 14. In this chapter we will look at the highest-resolution graphics mode, GR. 8.

TABLE 13.1 ATARI BASIC graphics modes

GR. Mode	Type	Resolution With Text Window	Resolution Full-screen	No. of Colors	Required Memory
0	TEXT		40×24	2	993
1	TEXT	20×20	20×24	5	513
2	TEXT	20×10	20×12	5	261
3	GRAPHICS	40×20	40×24	4	273
4	GRAPHICS	80×40	80×48	2	537
5	GRAPHICS	80×40	80×48	4	1017
6	GRAPHICS	160×80	160×96	2	2025
7	GRAPHICS	160×80	160×96	4	3945
8	GRAPHICS	320×160	320×192	1 (2)	7900

GRAPHICS MODE 8

In the high-resolution graphics mode 8 the screen is considered to be divided into a grid of 160 rows and 320 columns with four lines of text at the bottom, as shown in Figure 13.1.* The column positions of the grid are numbered 0 through 319 from left to right. The row positions of the grid are numbered 0 through 150 from top to bottom.

In graphics mode 8 only a single color can be plotted. The background and graphics point must have the same hue but can have different luminances. The background color is specified by color register 2 (COLOR 0). Thus, for example (see Chapter 7),

```
SETCOLOR 2,0,0
```

will make a black background. The points plotted must now be gray (hue = 0) and can only have a different luminance from the background. Color register 1 (COLOR 1) specifies the luminance of the points plotted. For example,

```
SETCOLOR 1,0,14
```

will plot white points with the brightest possible intensity.

Enter graphics mode 8 by typing

```
GR. 8
```

Set the background to black by typing

```
SETCOLOR 2,0,0
```

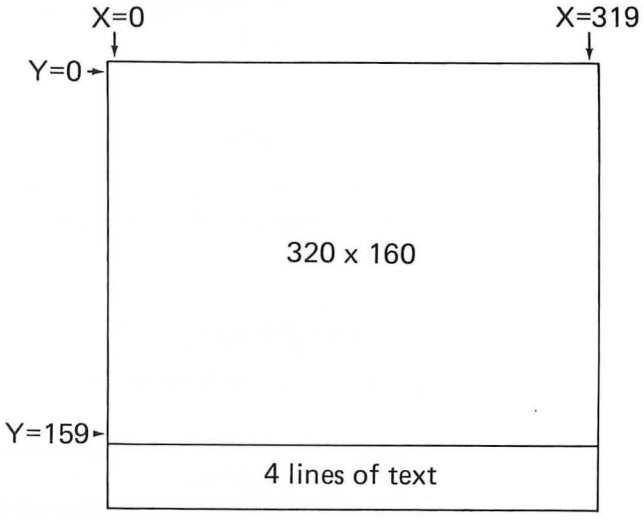
Set the luminance of the graphics points to bright by typing

```
SETCOLOR 1,0,14:COLOR 1
```

You can plot a point at any X,Y location (X between 0 and 319 and Y between 0 and 159) by typing

*A full-screen graphics mode with a 160×192 resolution is entered by typing GR. 8+16.

FIGURE 13.1 The high-resolution graphics mode divides the screen into a 320×160 grid with four lines of text at the bottom.




```
PLOT X,Y
```

For example, if you type

```
PLOT 159,80
```

you should see a small dot near the center of the screen.

The statement

```
DRAWTO X,Y
```

will plot a line from the most recently plotted point to the location X,Y. For example, type

```
PLOT 9,10:DRAWTO 309,10
```

This will plot a horizontal line across the top of the screen as shown in Figure 13.2.

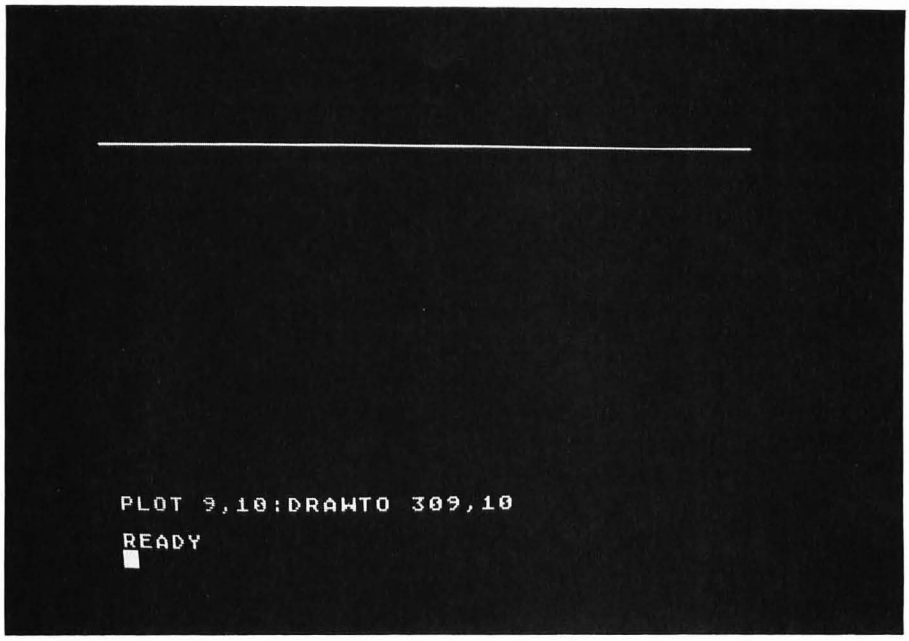


FIGURE 13.2 Plotting a horizontal line in high-resolution graphics.

If you now type

DRAWTO 9,159

the diagonal line shown in Figure 13.3 will be plotted. This is because the point 309,10 was the last point plotted in Figure 13.2.

If you now type

DRAWTO 9,10

the vertical line in Figure 13.4 will be plotted.

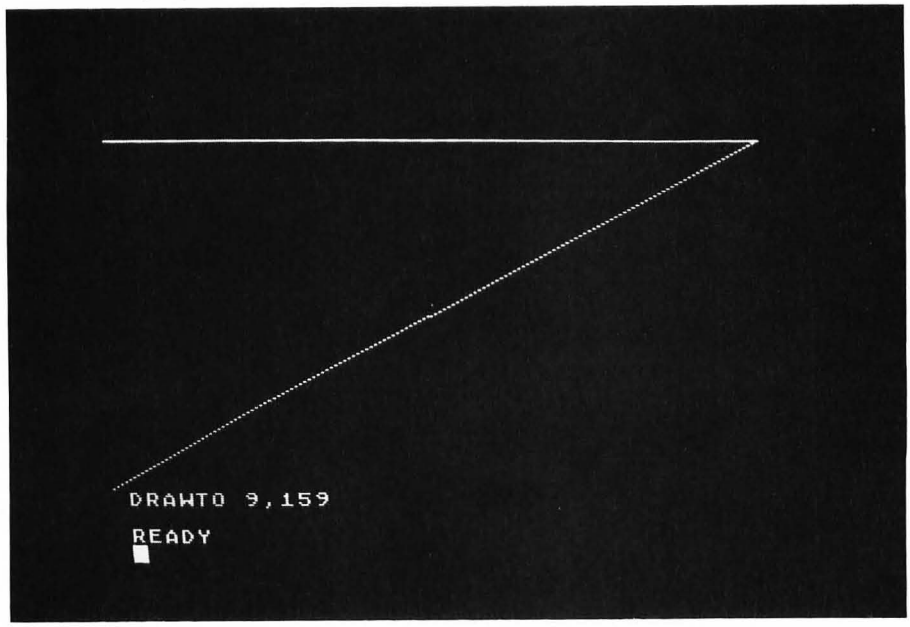
The program shown in Figure 13.5a will plot the pattern shown in Figure 13.5b.

In order to get out of the high-resolution graphics mode 8, type GR. 0 as you did when leaving the low-resolution graphics mode 5.

EXERCISE 13.1

Plot a square in high-resolution graphics that is 100 points on a side and has its upper-left-hand corner at the coordinates X = 90, Y = 30.

FIGURE 13.3 Plotting a diagonal line in high-resolution graphics.



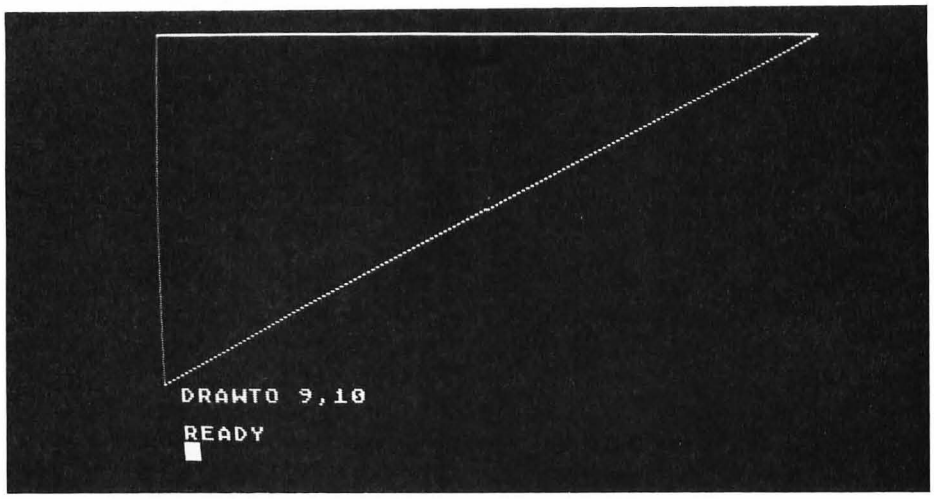
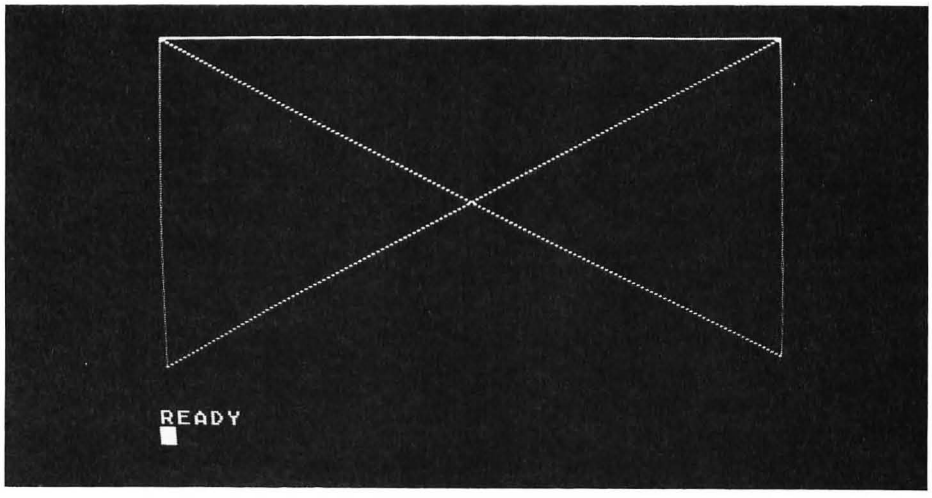


FIGURE 13.4 Plotting a vertical line in high-resolution graphics.

FIGURE 13.5 Plotting multiple lines in graphics mode 8.

```
10 REM HIGH RESOLUTION GRAPHICS
20 GRAPHICS 8
25 SETCOLOR 2,0,0:SETCOLOR 1,0,14
30 COLOR 1
40 PLOT 21,10:DRAWTO 301,10
50 DRAWTO 21,150:DRAWTO 21,10
60 DRAWTO 301,150:DRAWTO 301,10
```



PLOTTING HIGH-RESOLUTION GRAPHIC FIGURES

Let's suppose that you want to draw some arbitrary figure made up of a sequence of straight-line segments. It is convenient to plot the figure on a new X,Y coordinate system that is centered at the screen coordinates XC,YC. For example, in Figure 13.6 a square is shown plotted in such a coordinate system. Note that the Y coordinate is plotted in its "normal" upward direction, which is the opposite of the Y screen coordinate. We will let the computer take care of this difference. Also, note that we have located the origin of our new coordinate system at the center of the square. This means that the coordinates of the

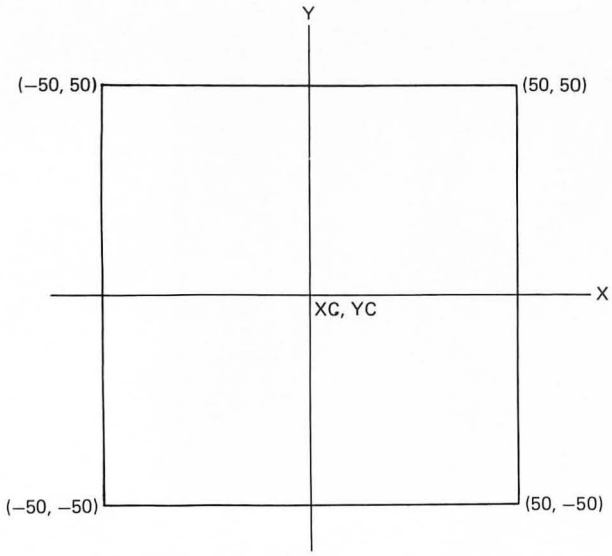


FIGURE 13.6 Defining a square centered on a new X,Y coordinate system.

vertices of the square may contain negative values, as shown in Figure 13.6. Of course, for the square to fit on the screen it is necessary for the value of XC (the center of the square) to be in the range 50–269 and the value of YC to be in the range 50–109.

For any center point XC,YC the square is completely defined by the X,Y coordinates of its vertices:

(– 50,50)
(50,50)
(50,– 50)
(– 50,– 50)

We will store these vertex coordinates in two DATA statements, with all of the X coordinates in the first DATA statement and all of the Y coordinates (in the same order) in the second DATA statement.

The vertex coordinates will be stored in the order you would use to draw the figure in a dot-to-dot fashion. If you return to the starting position, the first X,Y coordinate must also be the last one. Thus, the two DATA statements

250 DATA –50,50,50,–50,–50
260 DATA 50,50,–50,–50,50

will be used to plot the square shown in Figure 13.6. Statement 250 contains all of the X coordinates and statement 260 contains the corresponding Y coordinates.

The program shown in Figure 13.7 will plot this square. After the high-resolution graphics mode is entered in line 20 and color register 1 is set in line 25, the subroutine at line 200 is used to fill the arrays X(I) and Y(I) with the vertex coordinates. These arrays are dimensioned in line 210. The vertex coordinates stored in the DATA statements are then read into the arrays X(I) and Y(I) in lines 220 and 230. Note that in line 230 each value stored in Y(I) is changed in sign. This is because the Y coordinate shown in Figure 13.6 (from which the DATA coordinates were determined) is opposite in direction to the Y screen coordinate (see Figure 13.1).

Line 40 in Figure 13.7 defines the center of the square to be at the screen coordinates 159,80. Line 50 plots the point located at the upper-left-hand corner of the square. Note that the statement

PLOT XC+X(0),YC+Y(0)

will actually be equivalent to

PLOT 159–50,80+(–50)

and will therefore plot the point (– 50,50) shown in Figure 13.6. Remember that all of the signs in Y(I) were inverted in line 230. The statements

FIGURE 13.7 Program to plot a square.

```
10 REM SQUARE
20 GRAPHICS 8:COLOR 1
25 SETCOLOR 2,0,0:SETCOLOR 1,0,14
30 GOSUB 200:REM FILL ARRAYS
40 XC=159:YC=80
50 PLOT XC+X(0),YC+Y(0)
60 FOR I=1 TO M
70 DRAWTO XC+X(I),YC+Y(I)
80 NEXT I
90 END
200 REM FILL X,Y ARRAYS
210 M=4:DIM X(M),Y(M)
220 FOR I=0 TO M:READ X:X(I)=X:NEXT I
230 FOR I=0 TO M:READ Y:Y(I)=-Y:NEXT I
240 RETURN
250 DATA -50,50,50,-50,-50
260 DATA 50,50,-50,-50,50
```

```
60 FOR I=1 TO M
70 DRAWTO XC+X(I),YC+Y(I)
80 NEXT I
```

shown in Figure 13.7 will plot the four sides of the square.

The result of running the program given in Figure 13.7 is shown in Figure 13.8. Note that although we drew a square in Figure 13.6 it did not come out as a square in Figure 13.8. The reason for this is that the distance between adjacent points on a TV screen is different in the vertical and horizontal directions. You can see from Figure 13.8 that since the same number of points were plotted for both the vertical and horizonatal sides of the square, the distance between adjacent vertical points must be larger than the distance between adjacent horizontal points.

There is an easy way to correct for this difference in our program. If we measure the sides of the square on the screen we find that the horizontal length is 18.5 centimeters and the vertical length is 19.5 centimeters. Therefore, if we reduce the value of all Y coordinates by the factor

$$F = 19.5/18.5 = 1.1$$

the square should appear *square*. We can do this by adding the statement

```
215 F=1.1
```

and modifying line 230 to read

```
230 FOR I=0 TO M:READ
Y:Y(I)=-Y/F:NEXT I
```

as shown in Figure 13.9. The result of running this modified program is shown in Figure 13.10. Note that the square now looks like a square on the screen. You should use this vertical scaling factor (measure it for your own screen) in all of your plotting programs in order to produce properly proportioned figures.

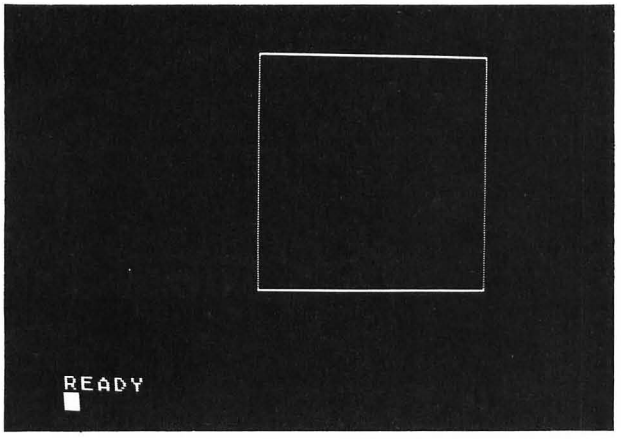


FIGURE 13.8 Result of running the program shown in Figure 13.7.

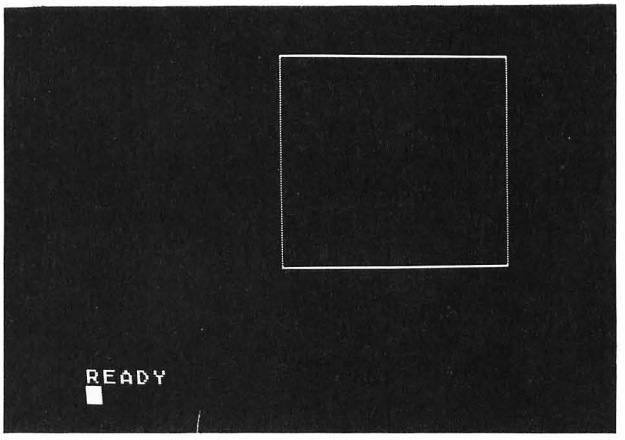


FIGURE 13.10 Result of running the modified "square" program shown in Figure 13.9.

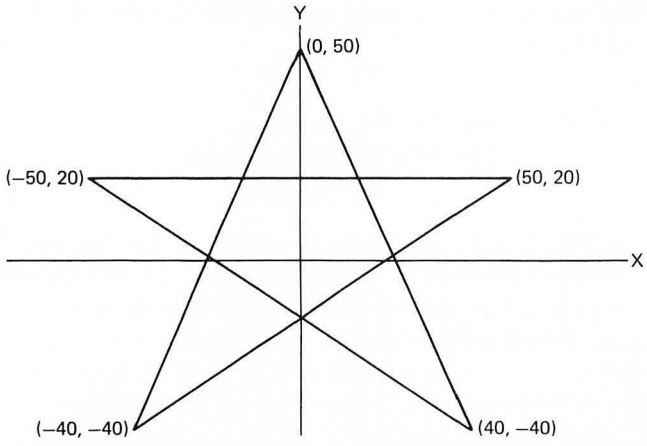
FIGURE 13.9 The changes shown in lines 215 and 230 will properly scale the Y coordinate.

```
10 REM SQUARE
20 GRAPHICS 8:COLOR 1
25 SETCOLOR 2,0,0:SETCOLOR 1,0,14
30 GOSUB 200:REM FILL ARRAYS
40 XC=159:YC=80
50 PLOT XC+X(0),YC+Y(0)
60 FOR I=1 TO M
70 DRAWTO XC+X(I),YC+Y(I)
80 NEXT I
90 END
200 REM FILL X,Y ARRAYS
210 M=4:DIM X(M),Y(M)
215 F=1.1
220 FOR I=0 TO M:READ X:X(I)=X:NEXT I
230 FOR I=0 TO M:READ Y:Y(I)=-Y/F:NEXT I
240 RETURN
250 DATA -50,50,50,-50,-50
260 DATA 50,50,-50,-50,50
```

EXERCISE 13.2

Write a program that will plot the star shown in Figure 13.11 centered at the screen coordinates 159,80.

FIGURE 13.11 Star figure to be plotted in Exercise 13.2.



Scaling Figures

If the coordinates of the points in a figure are stored in the arrays X(I) and Y(I) as we have described, then it is a simple matter to plot the figure in a different size. All you have to do is to always use

X(I)*S

and

Y(I)*S

in your plot statements, where S is the scale factor. For example, a value of S = 2 will cause the figure to be plotted double size and a value of S = 0.5 will cause the figure to be plotted half size.

As an example, the program shown in Figure 13.12 will plot a sequence of concentric squares centered at the screen coordinate 159, 80. Note that the *for . . . next* loop

```
45 FOR S=0.2 TO 1.6 STEP 0.2
    .
    .
    .
85 NEXT S
```

has been added to the program given in Figure 13.9.

FIGURE 13.13 Result of running the program shown in Figure 13.12.

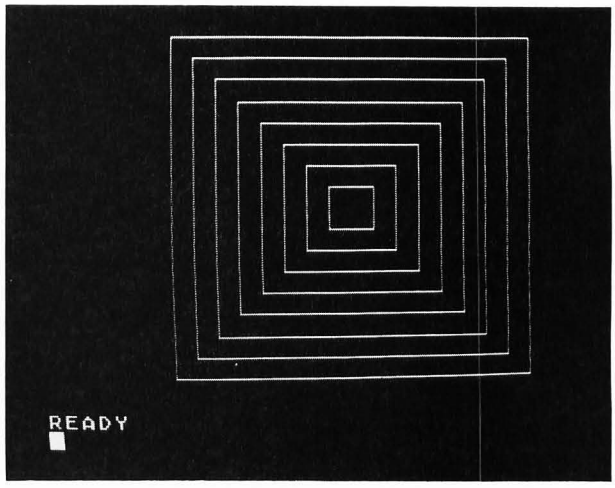


FIGURE 13.12 Program to plot a set of concentric squares.

```
10 REM CONCENTRIC SQUARES
20 GRAPHICS 8:COLOR 1
25 SETCOLOR 2,0,0:SETCOLOR 1,0,14
30 GOSUB 200:REM FILL ARRAYS
40 XC=159:YC=80
45 FOR S=0.2 TO 1.6 STEP 0.2
50 PLOT XC+X(0)*S,YC+Y(0)*S
60 FOR I=1 TO M
70 DRAWTO XC+X(I)*S,YC+Y(I)*S
80 NEXT I
85 NEXT S
90 END
200 REM FILL X,Y ARRAYS
210 M=4:DIM X(M),Y(M)
215 F=1.1
220 FOR I=0 TO M:READ X:X(I)=X:NEXT I
230 FOR I=0 TO M:READ Y:Y(I)=-Y/F:NEXT I
240 RETURN
250 DATA -50,50,50,-50,-50
260 DATA 50,50,-50,-50,50
```

In addition, the PLOT statement in line 50 and the DRAWTO statement in line 70 have been modified to include the scale factor S. The result of running this program is shown in Figure 13.13.

In addition to plotting a figure in different sizes by using the scale factor S, it is a simple matter to plot the figure at different locations on the screen by changing the values of the center coordinates XC and YC.

EXERCISE 13.3

Write a program that will plot nine squares in a 3×3 arrangement on the screen. The size of each square should be 40×40 and the squares should not overlap.

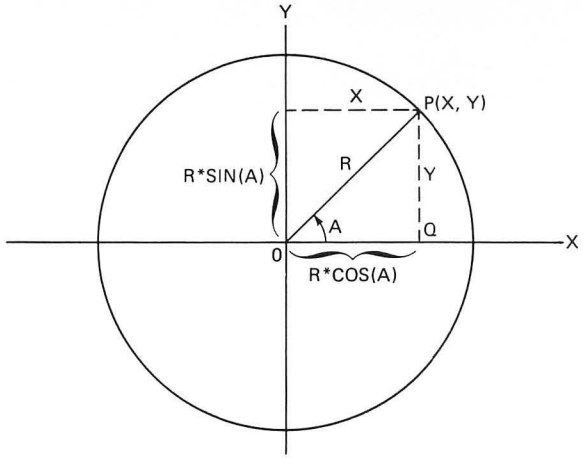
EXERCISE 13.4

Write a program that will plot five stars on the screen at random locations. Take care to ensure that no part of any star can ever extend beyond the edge of the screen.

EXERCISE 13.5

Modify the program in Exercise 13.4 so that the size of each star as well as its position is random.

FIGURE 13.14 Relationship between a point *X,Y* on a circle and the radius *R* and angle *A*.



Plotting Circles

How can you plot a circle in high-resolution graphics? Figure 13.14 shows a circle of radius *R* with its center at the origin of a local *X,Y* coordinate system. Recall from trigonometry that for the triangle *OPQ* the sine of the angle *A* is defined as

$$SIN(A) = Y/R$$

and the cosine of the angle *A* is defined as

$$COS(A) = X/R$$

Therefore, we see that any point *P* on the circle has the *X,Y* coordinates

$$X = R * COS(A)$$
$$Y = R * SIN(A)$$

We can therefore plot the circle by letting the angle *A* increase in steps, calculate new values for *X* and *Y* from the preceding equations, and DRAWTO to the new points. If we let *A* increase from 0 to 360 degrees we will plot the entire circle.

A program for plotting circles is shown in Figure 13.15. Lines 20–22 enter the high-resolution graphics mode and set color register 1. The user can specify the radius R and angular step size S in lines 25–45.

FIGURE 13.15 A program for plotting a circle of radius *R* using high-resolution graphics.

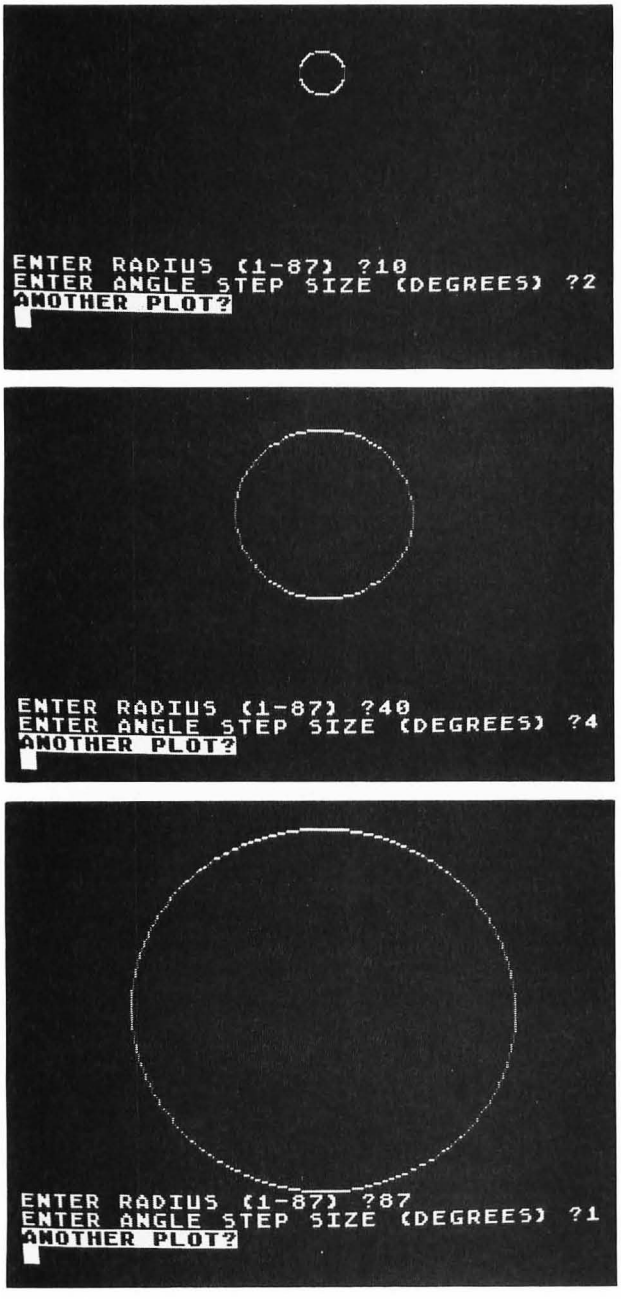
```
10 REM PLOTTING CIRCLES
15 OPEN #1,4,0,"K:"
20 GRAPHICS 8:COLOR 1
22 SETCOLOR 2,0,0:SETCOLOR 1,0,14
25 ? "ENTER RADIUS (1-87) ";
30 INPUT R
35 IF R<1 OR R>87 THEN 30
40 ? "ENTER ANGLE STEP SIZE (DEGREES) ";
45 INPUT S
50 DEG
60 F=1.1
70 XC=160:YC=80
80 PLOT XC+R,YC
90 FOR AD=0 TO 360 STEP S
100 X=R*COS(AD)
110 Y=R*SIN(AD)
120 Y=-Y/F
130 DRAWTO XC+X,YC+Y
140 NEXT AD
150 ? "ANOTHER PLOT?"
160 GET #1,A
170 IF A=ASC("Y") THEN 20
180 CLOSE #1:RAD
190 GRAPHICS 0:END
```

Line 50 sets DEG so that the arguments of the SIN and COS functions will be in degrees. Line 60 sets the vertical scale factor F. The center of the circle will be located at the screen coordinates XC,YC defined in line 70. The first point on the circle located at the screen coordinates XC + R, YC is plotted in line 80. The FOR . . . NEXT loop in lines 90–140 plots the rest of the circle. Notice that the angle AD increases from 0 degrees to 360 degrees in steps of S degrees. Lines 100–110 calculate the next values of X and Y on the circle. Line 120 inverts and scales the Y value as we have described so that the circle will look like a circle. The next segment of the circle is plotted in line 130.

Line 160 waits for a key to be pressed and then stores the ATASCII code for that key in A. The number 1 in this GET statement refers to the device number (for the keyboard) defined in the OPEN statement in line 15. This device is closed with the CLOSE statement in line 180.

You should type in this program and run it. Some sample runs are shown in Figure 13.16. Note that if the angle step size becomes too large, a polygon will be plotted rather than a circle. This suggests an easy way to plot some interesting multiple polygon figures. The next section will describe such a program.

FIGURE 13.16 Running the program shown in Figure 13.15.



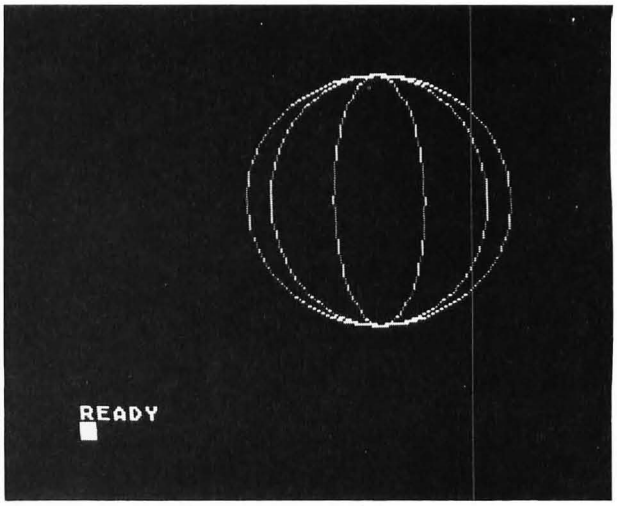
EXERCISE 13.6

Write a program to plot the "ball" shown in Figure 13.17a. Think of looking down on the top of the ball and then plotting a series of circles at different angles B, as shown in Figure 13.17. Both halves of this circle will appear on the screen, so you only need to let B increase from 0 to 90 degrees. The X coordinate of each circle will now be

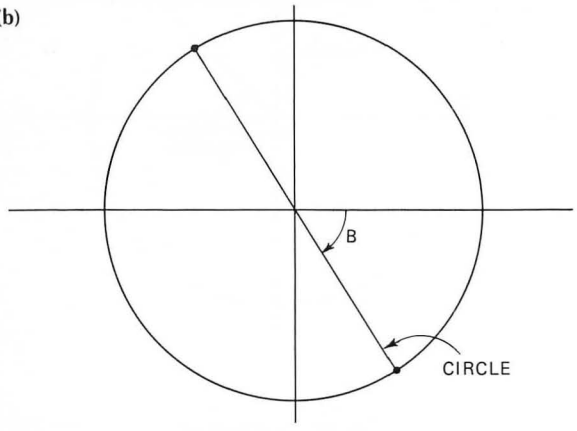
$$X = R * COS(A) * COS(B)$$

FIGURE 13.17 Plotting a 3-D ball (Exercise 13.6): (a) ball to be plotted; (b) looking down on top of the ball.

(a)



(b)



Plotting Polygons

Suppose that you would like to plot the picture shown in Figure 13.18. How would go about it? You could start at the vertex X(1),Y(1) and draw the four lines to X(K),Y(K) (K = 2 to 5), as shown in Figure 13.19a. Next you could add the three lines from X(2),Y(2) to X(K),Y(K) (K = 3 to 5) as shown in Figure 13.19b. Next you could add the two lines from X(3),Y(3) to X(K),Y(K) (K = 4 to 5) as shown in Figure 13.19c.

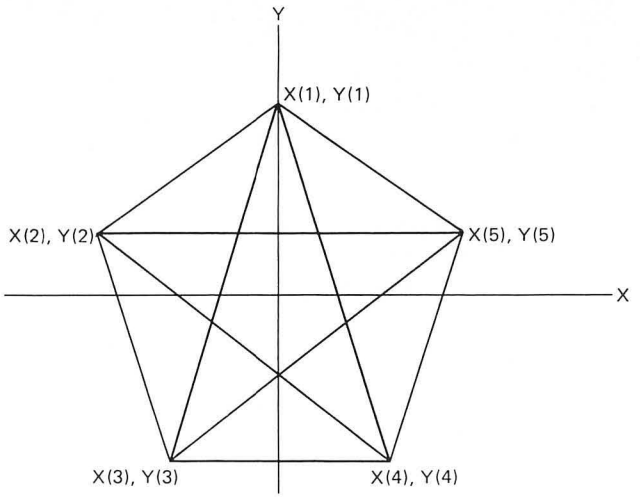
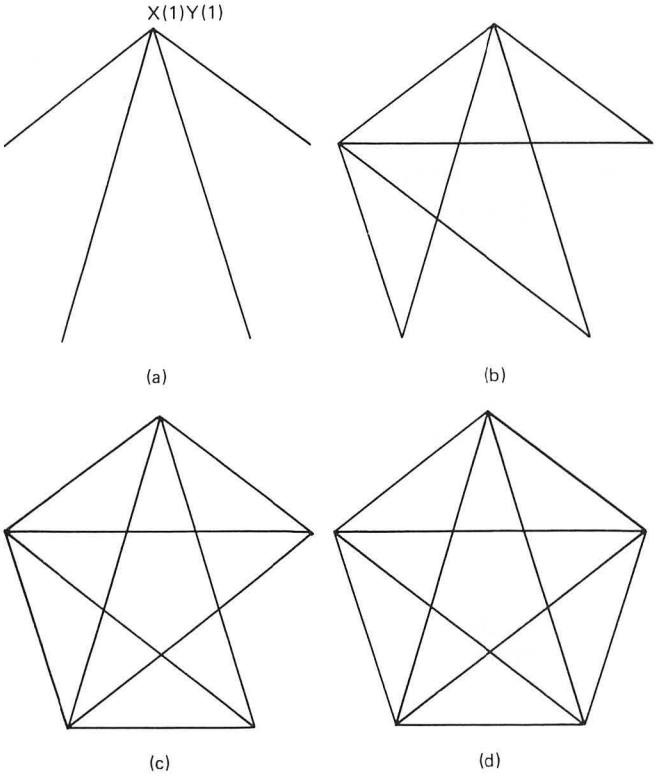


FIGURE 13.18 A polygon with a line down between all vertices.

FIGURE 13.19 Steps in generating the picture shown in Figure 13.18.



Finally you would add the line from X(4),Y(4) to X(5),Y(5). Note that the four steps shown in Figure 13.19 can be carried out by the algorithm

```
for J = 1 to N − 1
    for K = J + 1 to N
        plot line from X(J),Y(J) to X(K),Y(K)
    next K
next J
```

where N is the number of vertices in the polygon (five in Figure 13.19).

Think of a circle that passes through all of the vertices of the polygon. If R is the radius of this circle, then the N coordinate pairs X(I),Y(I) can be calculated from the following algorithm:

```
for  I = 1 to N
        A = I * 360/N
        X(I) = R          *
        COS(A)
        Y(I) = R * SIN(A)
next I
```

Note that this algorithm divides the circle into N pie-shaped wedges, where the angle of each "pie piece" is 360/N degrees. The coordinates X(I),Y(I) are then calculated using the equations of the circle.

A program that will plot this polygon figure for polygons with from 3 to 15 sides is shown in Figure 13.20. The center of the polygon will be at XC,YC, which is specified to be 160,80 in line 50. The radius of the circle that would pass through the polygon vertices is set to 85 in line 50. After entering the number of sides N in line 65 and checking to make sure that N is between 3 and 15 in line 70, the subroutine at line 200 is called. This subroutine calculates the N coordinates X(I),Y(I) as described previously. Note that line 250 inverts and scales the Y coordinates by our usual vertical scale factor F = 1.1 (defined in line 40).

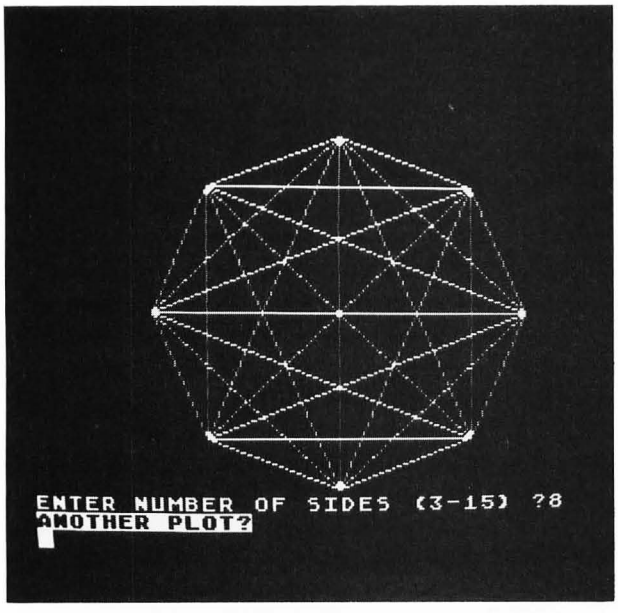
FIGURE 13.20 Program to plot a polygon figure with from 3 to 15 sides.

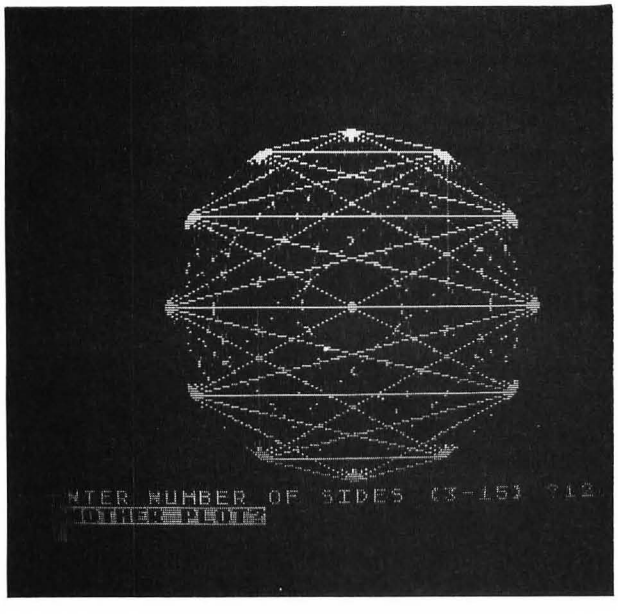
```
10 REM POLYGON FIGURE
15 OPEN #1,4,0,"K:"
20 DIM X(15),Y(15)
30 GRAPHICS 8:COLOR 1
35 SETCOLOR 2,0,0:SETCOLOR 1,0,14
40 DEG :F=1.1
50 XC=160:YC=80:R=85
60 ? "ENTER NUMBER OF SIDES (3-15) ";
65 INPUT N
70 IF N<3 OR N>15 THEN 60
80 GOSUB 200:REM CALCULATE POINTS
90 FOR J=1 TO N-1
100 FOR K=J+1 TO N
110 PLOT XC+X(J),YC+Y(J):DRAWTO XC+X(K),YC+Y(K)
120 NEXT K:NEXT J
130 ? "ANOTHER PLOT?"
140 GET #1,A
150 IF A=ASC("Y") THEN 30
160 CLOSE #1:RAD
170 GRAPHICS 0:END
200 REM CACULATE POINTS
210 FOR I=1 TO N
220 AD=I*360/N
230 X(I)=R*COS(AD)
240 Y(I)=R*SIN(AD)
250 Y(I)=-Y(I)/F
260 NEXT I
270 RETURN
```

Lines 90–120 actually plot the polygon figure using the nested *for . . . next* loops. Notice that the PLOT statement in line 110 will cause the polygon to be centered at XC,YC. After plotting the figure, line 130 will ask the user if another plot is desired. The GET statement in line 140 will wait for a response and then in line 150 will branch back to line 30 if key Y is pressed. Any other key will cause line 160 to be executed, which will close the keyboard device number, clear the screen, and stop the program.

Type in this program and run it. Two figures that can be generated by this program are shown in Figure 13.21. Try some different values of N.

FIGURE 13.21 Examples of polygon figures that can be generated by the program shown in Figure 13.20.





Plotting Functions

The high-resolution graphics capability of the ATARI makes it a useful tool for studying the behavior of mathematical functions. For example, the function

$$y(x) = A \sin(2\pi x/T + \phi)$$

defines a *sine wave* with amplitude A, period T, and phase angle ϕ. You can calculate this function by using the BASIC statement

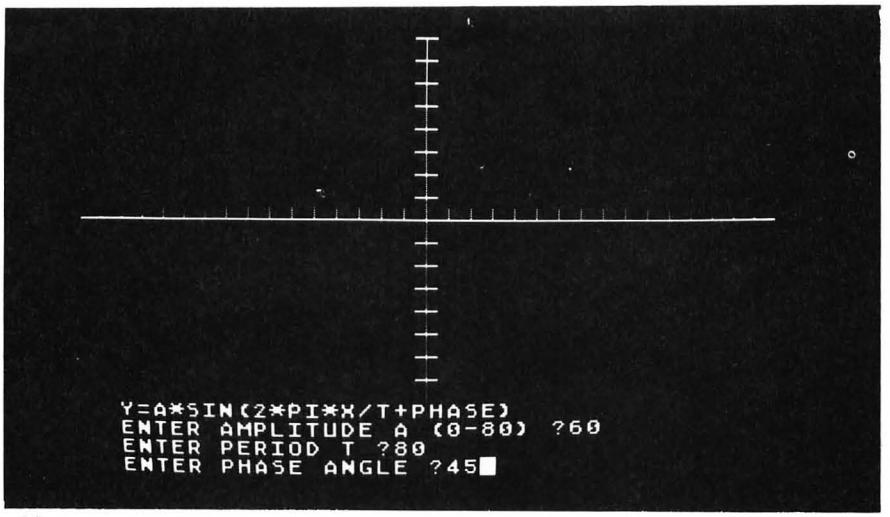
```
Y=-A*SIN(2*PI*X/T+PH)
```

where PI = 3.1415926, PH is the phase angle in radians, and the minus sign is our usual inversion because the positive Y screen coordinate points downward.

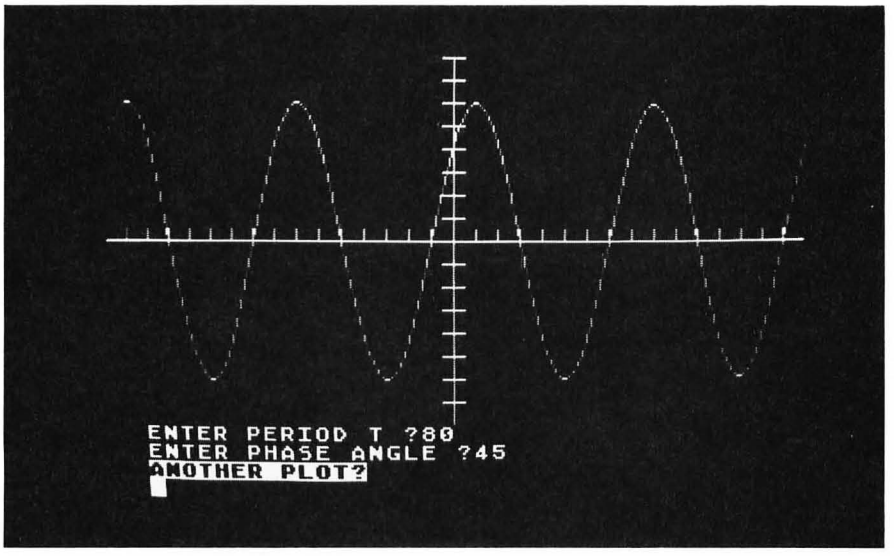
A program that will plot this function is shown in Figure 13.22. The subroutine at line 200 that is called in line 30 will plot the axes shown in Figure 13.23a.

FIGURE 13.22 Program to plot a sine wave.

```
10 REM SINE WAVE
15 OPEN #1,4,0,"K:"
20 GRAPHICS 8:COLOR 1
22 SETCOLOR 2,0,0:SETCOLOR 1,0,14
25 PI=3.1415926
30 GOSUB 200:REM PLOT AXES
40 ? "Y=A*SIN(2*PI*X/T+PHASE)"
50 ? "ENTER AMPLITUDE A (0-80) ";
55 INPUT A
60 ? "ENTER PERIOD T ";
65 INPUT T
70 ? "ENTER PHASE ANGLE ";
75 INPUT PH
80 PH=PH*PI/180
90 X=-159:Y=-A*SIN(2*PI*X/T+PH)
95 PLOT XC+X,YC+Y
100 FOR X=-158 TO 160
110 Y=-A*SIN(2*PI*X/T+PH)
120 DRAWTO XC+X,YC+Y
130 NEXT X
140 ? "ANOTHER PLOT?"
150 GET #1,KEY
160 IF KEY=ASC("Y") THEN 20
170 CLOSE #1:GRAPHICS 0:END
200 REM PLOT AXES
210 XC=159:YC=80
220 PLOT 0,YC:DRAWTO 319,YC
230 PLOT XC,0:DRAWTO XC,159
240 FOR X=9 TO 309 STEP 10
250 PLOT X,YC:DRAWTO X,YC-5
260 NEXT X
270 FOR Y=0 TO 150 STEP 10
280 PLOT XC-5,Y:DRAWTO XC+5,Y
290 NEXT Y
300 RETURN
```



(a)



(b)

FIGURE 13.23 (a) Axes plotted by the subroutine at line 200 in Figure 13.22; (b) example of sine wave that is plotted by the program shown in Figure 13.22.

Each grid mark on the axes represents an increment of 10 screen units. Lines 50–75 allow the user to enter values for the amplitude A, the period T, and the phase angle PH (in degrees). Line 80 converts the phase angle from degrees to radians.

The coordinate system plotted in the subroutine at line 200 originates at the screen coordinates XC,YC (defined to be 159,80 in line 210). The first point of the function that is plotted (in line 95) will be the leftmost value of X (− 159). This value is assigned in line 90 together with the corresponding value of Y.

The FOR . . . NEXT loop in lines 100–130 plots the rest of the curve. Notice that the value of X is increased from − 158 to + 160 and for each value of X the value of the function Y is calculated in line 110.

Type in this program and run it. A sample run is shown in Figure 13.23b. By entering different values of the amplitude A, period T, and phase angle PH you will be able to get a good idea of how this function behaves.

EXERCISE 13.7

Write programs that the user can use to plot the following functions for different values of the parameters A, C, and N:

1. Y = A * LOG(X/C) X > 0
2. Y = A * EXP(− X/C)
3. Y = X ∧ N/C
4. Y = A * SQR(X/C) X > 0

14

LEARNING TO PEEK AND POKE

As you have learned, the ATARI 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 to the outside world. Examples of communication with the outside world include getting data from the keyboard and game paddles, and writing and reading data to and from a diskette.

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 ATARI contains the data in C3. The ATARI's BASIC interpreter automatically takes care of assigning these locations. However, in order to use the full power of the ATARI, you must sometimes read and write data to *specific* memory locations within the ATARI. 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 even in BASIC. You do this by using the PEEK and POKE statements. In this chapter you will learn

1. how data are stored in memory locations in the ATARI
2. how to use the PEEK and POKE statements
3. how to use the console switches on the keyboard
4. how to tell if a key on the keyboard has been pressed
5. how to use the graphics 1 and 2 text modes
6. how to define your own character set.

THE STATEMENTS PEEK AND POKE

The 6502 microprocessor (see Figure 3.1) that is the "brain" of the ATARI can address a total of 65536 memory locations (with addresses between 0 and 65535). The reason for this is that the 6502 has 16 address lines and each line can be either high or low (1 or 0). Thus a typical address might be represented by the 16 bits

0011010111000001

This *binary* number is equivalent to the *decimal* number 13761. Thus, this memory location would have an address of 13761. Since each of the 16 bits in the address can be either a 1 or a 0, the total number of possible addresses is

$$2^{16} = 65536$$

Your ATARI will actually contain less than this maximum amount of memory.

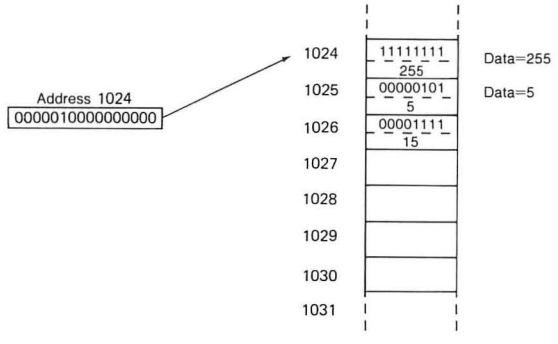
When working with binary numbers such as this address, it is convenient to represent these binary numbers as *hexadecimal* numbers. This is not necessary when using BASIC, because the PEEK and POKE statements use only *decimal* numbers. However, if you want to program in assembly language, the use of hexadecimal numbers is essential. Although you do not need to know anything about hexadecimal numbers to use this book, if you are curious, a brief discussion of hexadecimal numbers is given in Appendix D.

In addition to the 16 address lines, the 6502 microprocessor has 8 data lines. These lines connect the microprocessor to all of the memory chips in the ATARI. Thus, data are moved between memory locations in groups of 8 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 ATARI can have a value between 0 and 255. This relationship between addresses and data is shown in Figure 14.1.

FIGURE 14.1 Each address in the range 0–65535 points to a memory location containing data in the range 0–255.



In this figure memory location 1024 contains a data value of 255 (eight 1s), and memory location 1025 contains a data value of 5.

You can find the data value stored in a particular location by using the PEEK statement. You can store a particular data value in a given memory location by using the POKE statement.

PEEK

The function

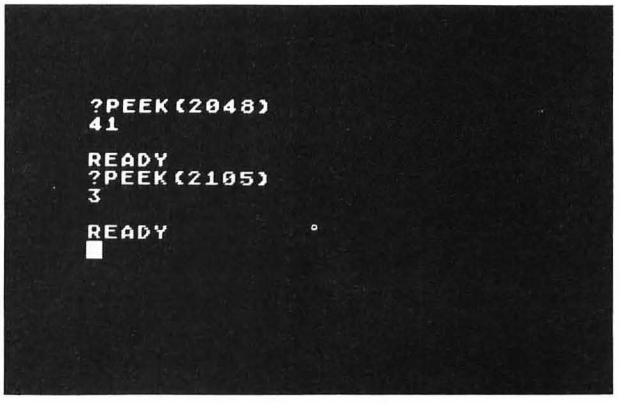
PEEK(*Addr*)

returns the data value stored in the memory location with an address *Addr*. The value of *Addr* must be in the range 0 through 65535. Try printing some value of PEEK to see what you get. For example, try

```
?PEEK(2048)
?PEEK(2105)
```

as shown in Figure 14.2. Your ATARI will probably contain different data values in these locations from those shown in Figure 14.2.

FIGURE 14.2 Memory location 2048 contains the value 41 and memory location 2105 contains the value 3.



Certain memory locations have particular meanings to the ATARI. For example, memory location 53279 will tell you which of the console keys (OPTION, SELECT, and START) are being pressed.

In order to see how this works, type and run the following one-line program:

```
10 ?PEEK(53279):GOTO 10
```

The data value in location 53279 will keep being displayed and will scroll off the screen. Press the console keys and watch what happens. Note that the value displayed is between 0 and 7, according to Table 14.1.

TABLE 14.1 Value of PEEK(53279) when console keys are pressed

Value	*OPTION*	*SELECT*	*START*
0	X	X	X
1	X	X	
2	X		X
3	X		
4		X	X
5		X	
6			X
7			

This fact can be used in programs if you want to know when one of the console keys is being 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 the START key is pressed. When you release the START key the ball will stop.

FIGURE 14.3 The ball will move back and forth across the screen as long as the START key is pressed.

```
10 REM BOUNCING BALL
20 GRAPHICS 5:SETCOLOR 0,0,14
30 FOR X=5 TO 75
35 COLOR 1:PLOT X,20
40 GOSUB 200:REM TEST START KEY
50 COLOR 0:PLOT X,20
60 NEXT X
70 FOR X=74 TO 6 STEP -1
80 COLOR 1:PLOT X,20
90 GOSUB 200:REM TEST START KEY
100 COLOR 0:PLOT X,20
110 NEXT X
120 GOTO 30
200 REM TEST START KEY
210 IF PEEK(53279)<>6 THEN 210
220 RETURN
```

The FOR . . . NEXT loop in lines 30–60 causes the ball to move from left to right across the screen if the START key is pressed. The subroutine in lines 200–220 tests if the START key is being pressed. If it is *not,* then line 210 just loops on itself. This means that the last spot plotted in line 35 will remain stationary on the screen. If the START key is being pressed, the subroutine will exit at line 220 and line 50 will then erase the most recently plotted spot. A new spot will then be plotted in line 35, just to the right of the previously plotted (and erased) spot, as the FOR . . . NEXT loop increments the value of X by 1.

After a spot has been plotted and erased at the screen position X = 75, the FOR . . . NEXT loop is exited and another FOR . . . NEXT loop in lines 70–110 is executed. This loop moves the spot in a similar manner from right to left. When this loop is exited after the spot has been plotted and erased in position X = 6, then line 120 branches back to line 30 so that the entire process will be repeated as long as the START key is being pressed. Since the START key testing subroutine is always called (in lines 40 and 90) after a spot has been plotted and before it has been erased, when you release the START key a single spot will always remain on the screen. Type in this program and run it. Modify the program so that the ball moves back and forth when the OPTION key is pressed.

POKE

Whereas PEEK allows you to read the data value in a particular memory location, the statement

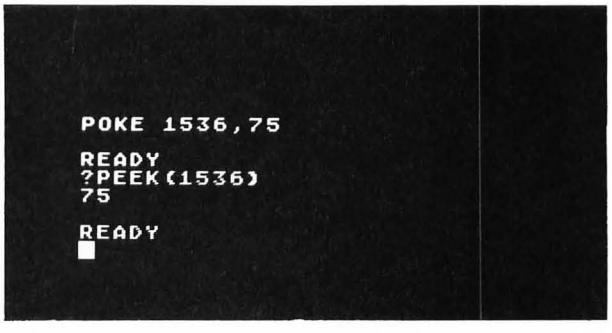
POKE *Addr, Data*

allows you to store the value *Data* in the memory location *Addr.* For example, type

```
POKE 1536,75
?PEEK(1536)
```

as shown in Figure 14.4. Note that PEEK(1536) verifies that you actually stored the value 75 in memory location 1536.

FIGURE 14.4 POKE 1536,75 stores the value 75 in memory location 1536.



Memory location 752 controls the visibility of the cursor. A value of 0 in location 752 causes the screen cursor to be visible. If you POKE any other value into location 752 the cursor will disappear. This can be useful when you don't want the cursor to remain on the screen. For example, add the statement

55 POKE 752,1

to the program shown in Figure 5.4 in Chapter 5 and note that the cursor is not displayed after the horizontal line is drawn.

READING THE KEYBOARD

The memory location 764 is used by the ATARI to store the value of the last key pressed. As long as no key has been pressed, the value stored in memory location 764 will be 255. When a key is pressed a value associated with that key is stored in location 764. These key values are *not* the ASCII codes but are special key codes given in Table 14.2.

In order to see how this works type in and run the following one-line program:

10 ?PEEK(764):GOTO 10

TABLE 14.2 Key codes stored in location 764 when a key is pressed

A	63	N	35	0	50	;	2	'	115	space	33
B	21	O	8	1	31	+	6	@	117	return	12
C	18	P	10	2	30	*	7	(	112	ESC	28
D	58	Q	47	3	26	=	15	)	114	TAB	44
E	42	R	40	4	24	-	14	_	78	clear	118
F	56	S	62	5	29	<	54			insert	119
G	61	T	45	6	27	>	55	|	79	delete	116
H	57	U	11	7	51	!	95	\	70	back s	52
I	13	V	16	8	53	"	94	^	71	lowr	60
J	1	W	46	9	48	#	90	?	102	caps	124
K	5	X	22	,	32	$	88	[	96	ATARI	39
L	0	Y	43	.	34	%	93	]	98		
M	37	Z	23	/	38	&	91	:	66		

The data value in location 764 will keep being displayed and will scroll off the screen. Press any key and watch what happens. Note that the value displayed corresponds to the most recently pressed key according to Table 14.2.

PEEKing the special keyboard memory location 764 is useful when you want to see if a key has been pressed. For example, suppose that you want to move the bouncing ball in Figure 14.3 by pressing *any* key on the keyboard. You can do this by modifying the subroutine at line 200 in Figure 14.3 to read

```
200 REM TEST ANY KEY
210 IF PEEK(764)=255 THEN 210
220 POKE 764,255
230 RETURN
```

The resulting program is shown in Figure 14.5. Note that after each new ball is plotted (and before it is erased), the subroutine at line 200 is called. Line 210 checks to see if any key has been pressed. If not, line 210 will then loop on itself until any key is pressed. During this time the ball will remain stationary on the screen. When a key has been pressed (any key), line 220 will restore the value 255 in location 764 before returning to the main program. The main program will then erase the ball, plot a new one at the adjacent location, and call the subroutine at line 200 again.

FIGURE 14.5 Any key can be used to start and stop the bouncing ball.

```
10 REM BOUNCING BALL
20 GRAPHICS 5:SETCOLOR 0,0,14
30 FOR X=5 TO 75
35 COLOR 1:PLOT X,20
40 GOSUB 200:REM TEST ANY KEY
50 COLOR 0:PLOT X,20
60 NEXT X
70 FOR X=74 TO 6 STEP -1
80 COLOR 1:PLOT X,20
90 GOSUB 200:REM TEST ANY KEY
100 COLOR 0:PLOT X,20
110 NEXT X
120 GOTO 30
200 REM TEST ANY KEY
210 IF PEEK(764)=255 THEN 210
220 POKE 764,255
230 RETURN
```

Type in this program and run it. Note that you can move the bouncing ball by pressing any key. Also note that if you hold down any key, the repeat feature will cause the ball to continually start and stop and therefore move back and forth across the screen.

TEXT MODES 1 AND 2

Graphics modes 1 and 2 are expanded text modes. (See Table 13.1 in Chapter 13.) These modes may or may not include a four-line (GR. 0) text window at the bottom of the screen. The GR. 1 mode displays characters double width in the graphics screen area. The GR. 2 mode displays characters both double width and double height. This means that with the four-line text window at the bottom of the screen, text mode 1 can display 20 rows of 20 characters per row, while the text mode 2 can display 10 rows of 20 characters per row. The full-screen mode GR. 1 + 16 will add four more lines of text 1 characters, and the full-screen mode GR. 2 + 16 will add two more rows of text 2 characters.

To see how these expanded text modes work, type the following statements in the immediate mode as shown in Figure 14.6:

```
GR. 1
SETCOLOR 0,0,14:SETCOLOR 1,0,14
POSITION 2,2
?#6;"MODE 1 CHARACTERS"
?"MODE 0 CHARACTERS"
```

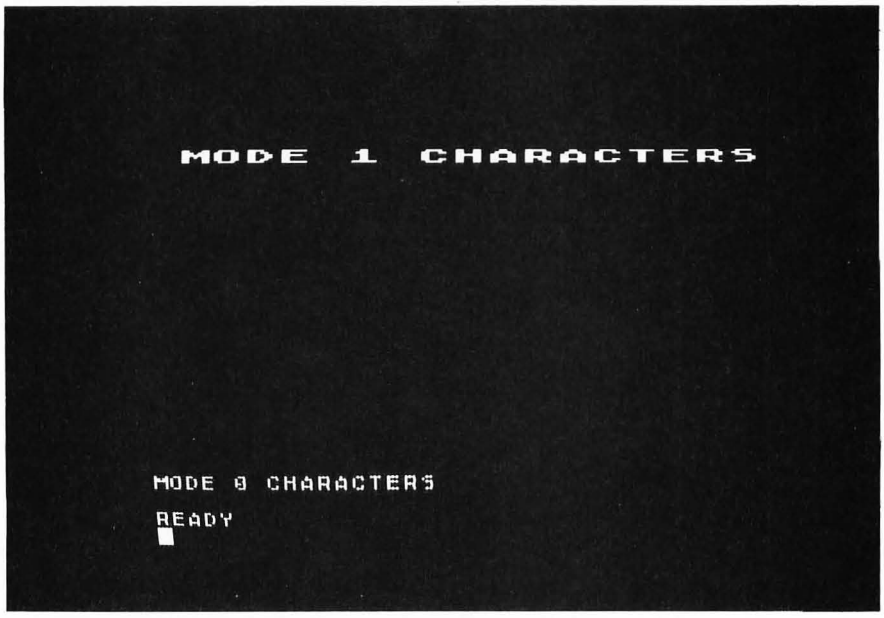


FIGURE 14.6 Mode 1 expanded text characters.

Note that the print statement PRINT #6; or ?#6; is used to print characters in the text 1 (or text 2) screen area. The standard print statement will print the characters in the mode 0 text window at the bottom of the screen.

To print mode 2 expanded text characters type the following statements as shown in Figure 14.7:

```
GR. 2
SETCOLOR 1,0,14:SETCOLOR 0,0,14
POSITION 2,2
?#6;"MODE 2 CHARACTERS"
```

The mode 1 and 2 text screens do not behave like the mode 0 text screen in several regards. First of all, mode 0 can display the entire set of 128 characters in both normal and reverse video. Modes 1 and 2, on the other hand, can display only a 64-character set (with no reverse video) at any one time.

The character set used by the ATARI is stored in ROM and consists of the 128 characters shown in Table 14.3. The first 64 characters, consisting of upper-case, digits, and punctuation, comprise the standard set normally used by the GR. 1 and GR. 2 expanded text modes. If you execute the statement

```
POKE 756,226
```

you will change to the alternate character set consisting of the 64 characters in columns 3 and 4 in Table 14.3. These include the graphic characters and lower-case letters.

FIGURE 14.7 Mode 2 expanded text characters.

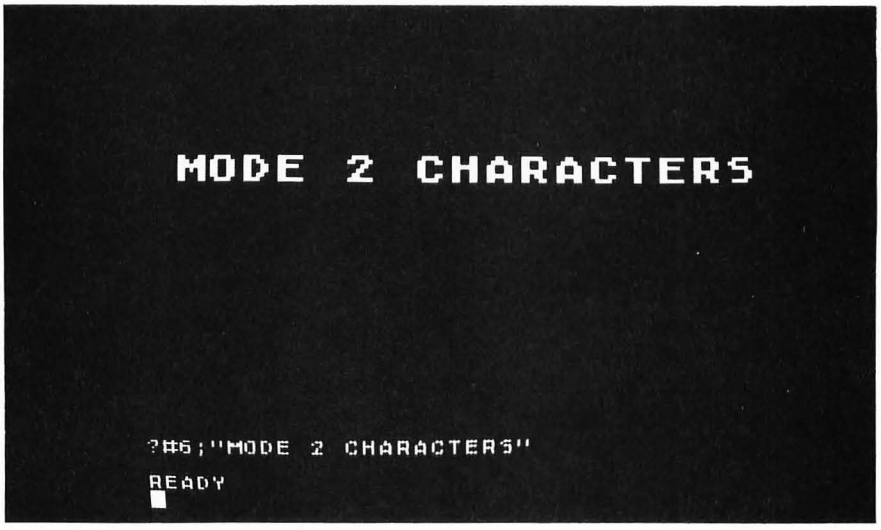


TABLE 14.3 The internal character set

Standard Character Set						Alternate Character Set					
Column 1			Column 2			Column 3			Column 4		
		ATASCII			ATASCII			ATASCII			ATASCII
No.	Char.	Code	No.	Char.	Code	No.	Char.	Code	No.	Char.	Code
0	Space	32	32	@	64	64		0	96		96
1	!	33	33	A	65	65		1	97	a	97
2	"	34	34	B	66	66		2	98	b	98
3	#	35	35	C	67	67		3	99	c	99
4	$	36	36	D	68	68		4	100	d	100
5	%	37	37	E	69	69		5	101	e	101
6	&	38	38	F	70	70		6	102	f	102
7	'	39	39	G	71	71		7	103	g	103
8	(	40	40	H	72	72		8	104	h	104
9	)	41	41	I	73	73		9	105	i	105
10	*	42	42	J	74	74		10	106	j	106
11	+	43	43	K	75	75		11	107	k	107
12	,	44	44	L	76	76		12	108	l	108
13	–	45	45	m	77	77		13	109	m	109
14	.	46	46	N	78	78		14	110	n	110
15	/	47	47	O	79	79		15	111	o	111
16	0	48	48	P	80	80		16	112	p	112
17	1	49	49	Q	81	81		17	113	q	113
18	2	50	50	R	82	82		18	114	r	114
19	3	51	51	S	83	83		19	115	s	115
20	4	52	52	T	84	84		20	116	t	116
21	5	53	53	U	85	85		21	117	u	117
22	6	54	54	V	86	86		22	118	v	118
23	7	55	55	W	87	87		13	119	w	119
24	8	56	56	X	88	88		24	120	x	120
25	9	57	57	Y	89	89		25	121	y	121
26	:	58	58	Z	90	90		26	122	z	122
27	;	59	59	[	91	91		27	123		123
28	<	60	60	\	92	92		28	124	|	124
29	=	61	61	]	93	93		29	125		125
30	>	62	62	^	94	94		30	126		126
31	?	63	63	_	95	95		31	127		127

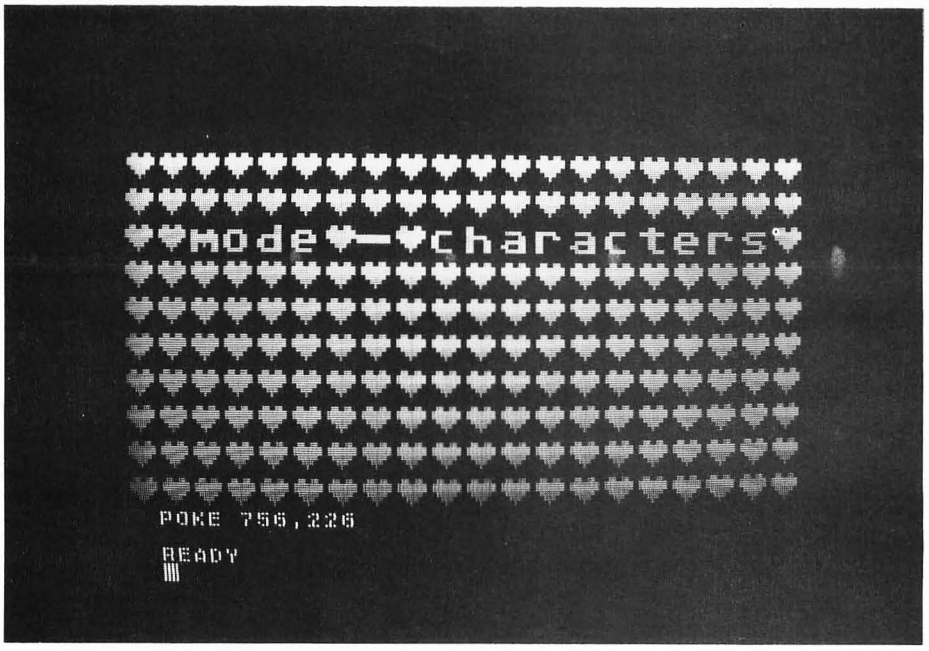


FIGURE 14.8 POKE 756,226 will switch mode 1 or mode 2 to the alternate character set.

After displaying the mode 2 characters shown in Figure 14.7, type

POKE 756,226

which will switch to the alternate character set, as shown in Figure 14.8. Note that all blank spaces (character 0 in Table 14.3) have changed to heart graphic characters (character 64 in Table 14.3). All other characters in columns 1 and 2 in Table 14.3 are replaced by the corresponding characters in columns 3 and 4. This means that in the expanded text modes 1 and 2 you can use either the characters in columns 1 and 2 in Table 14.3 *or* the characters in columns 3 and 4. However, you cannot use both because the text modes 1 and 2 can display only 64 characters at a time. Later in this chapter, you will learn how to mix some characters from columns 3 and 4 with some from columns 1 and 2 by defining your own character set.

You can return to the standard character set by typing

POKE 756,224

Try it.

The text modes 1 and 2 do not scroll and you will get an error if you try to print characters outside the screen area.

Mode 1 and 2 Colors

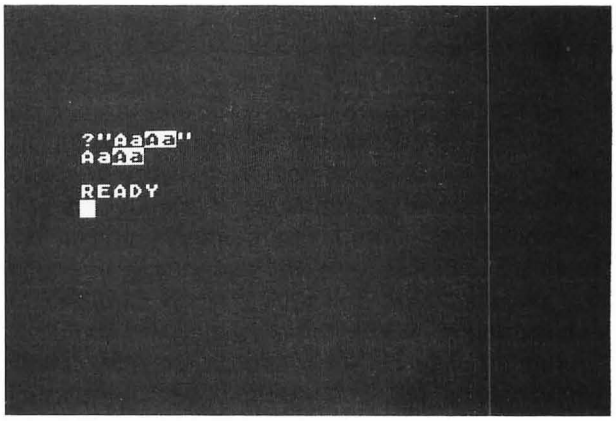
Color register number 4 defines the background color in modes 1 and 2. The default background color is black (SETCOLOR 4,0,0). Color registers 0–3 are used to define up to four colors for text 1 and 2 characters. However, these colors are defined in a somewhat strange way.

Consider the following four ATASCII codes (see Appendix B):

65 Upper-case A
97 Lower-case a
193 Reverse video upper-case A
225 Reverse video lower-case a

You can type each of these characters in the GR. 0 text mode by using the CAPS/LOWR key and the ATARI key (for reverse video), as shown in Figure 14.9. Now type these same four characters in the text 2 mode by typing

FIGURE 14.9 ATASCII codes 65, 97, 193, and 225 in the GR. 0 text mode.



```
GR. 2
POSITION 2,2
?#6;"AaAa"
     ↑↑___ reverse video
```

as shown in Figure 14.10. Note that each of the four characters is displayed as an upper-case A with a different color.

In text modes 1 and 2 the ATASCII code determines not only the character to be plotted but also the color. The four ATASCII codes 65, 97, 193, and 225 will all plot an upper-case A, but the color of the A will be determined by the value in color registers 0, 1, 2, and 3, respectively. The default SETCOLOR values are shown in Table 14.4. You can change the colors of the letters displayed in Figure 14.10 by changing the values in color registers 0–3. For example, type SETCOLOR 1,0,12 and the second A should change to white. Change the values in the other color registers.

TABLE 14.4 Default SETCOLOR values (SETCOLOR R, H, L)

Color Register R	*Hue H*	*Luminance L*	*Color*
0	2	8	Orange
1	12	10	Green
2	9	4	Blue
3	4	6	Red
4	0	0	Black

There is another way to plot characters in text modes 1 and 2. You can use the COLOR statement to define both the character and the color, and then use the PLOT statement.

For example, type

```
GR. 2
COLOR 65:PLOT 4,4
```

Note that an A with a color determined by color register 0 is plotted at location 4,4 on the screen. If you type

```
COLOR 193:PLOT 8,5
```

an A will be plotted at location 8,5 with its color determined by color register 2.

Table 14.5 shows what numbers to use in the COLOR statement to plot any standard character in any of four colors. Note that the characters are listed in the order shown in columns 1 and 2 of Table 14.3. This is the order in which the characters are stored inside the computer. If you type POKE 756,226, then the COLOR numbers shown in Table 14.5 will plot the corresponding alternate characters shown in columns 3 and 4 in Table 14.3

TABLE 14.5 COLOR values for text modes 1 and 2

		Color Register			
Char #	*Char*	*0*	*1*	*2*	*3*
0	Space	32	0	160	128
1	!	33	1	161	129
2	"	34	2	162	130
3	#	35	3	163	131
4	$	36	4	164	132
5	%	37	5	165	133
6	&	38	6	166	134
7	'	39	7	167	135
8	(	40	8	168	136
9	)	41	9	169	137
10	*	42	10	170	138
11	+	43	11	171	139
12	,	44	12	172	140
13	-	45	13	173	141
14	.	46	14	174	142
15	/	47	15	175	143
16	0	48	16	176	144
17	1	49	17	177	145
18	2	50	18	178	146
19	3	51	19	179	147
20	4	52	20	180	148
21	5	53	21	181	149
22	6	54	22	182	150
23	7	55	23	183	151
24	8	56	24	184	152
25	9	57	25	185	153
26	:	58	26	186	154
27	;	59	27	187	None
28	<	60	28	188	156
29	=	61	29	189	157
30	>	62	30	190	158
31	?	63	31	191	159
32	@	64	96	192	224
33	A	65	97	193	225
34	B	66	98	194	226
35	C	67	99	195	227
36	D	68	100	196	228
37	E	69	101	197	229
38	F	70	102	198	230
39	G	71	103	199	231
40	H	72	104	200	232
41	I	73	105	201	233
42	J	74	106	202	234
43	K	75	107	203	235
44	L	76	108	204	236
45	M	77	109	205	237
46	N	78	110	206	238
47	O	79	111	207	239
48	P	80	112	208	240
49	Q	81	113	209	241
50	R	82	114	210	242
51	S	83	115	211	243
52	T	84	116	212	244
53	U	85	117	213	245
54	V	86	118	214	246
55	W	87	119	215	247
56	X	88	120	216	248
57	Y	89	121	217	249
58	Z	90	122	218	250
59	[	91	123	219	251
60	\	92	124	220	252
61	]	93	None	221	253
62	^	94	126	222	254
63	_	95	127	223	255

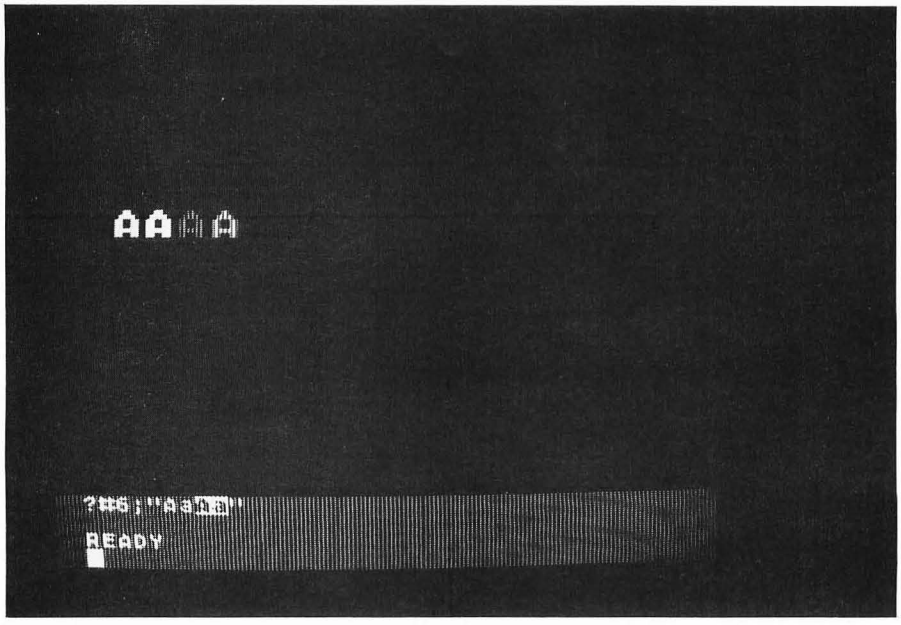


FIGURE 14.10 ATASCII codes 65, 97, 193, and 225 in the GR. 2 text mode.

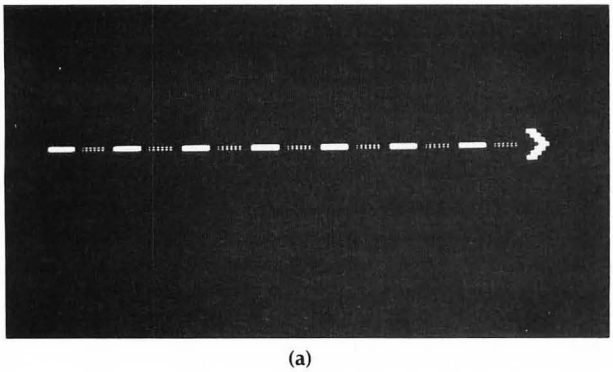
As an example using the expanded text 2 mode, type in and run the program shown in Figure 14.11. Within the FOR . . . NEXT loop (lines 30–60), line 40 plots a series of dashes, spaced two apart, whose color is determined by color register 0 (see Table 14.5). Line 50 plots a similar sequence of dashes in between the other set with a color determined by color register 1. Line 70 plots a right arrow (>) at the end of the line of dashes. The color of the arrow is determined by color register 0.

Lines 80–90 set color register 0 to white and color register 1 to blue. Every other dash will therefore be white and the alternate ones will be blue. The arrow will appear as shown in Figure 14.12a. After a short delay in line 100, lines 110–120 reverse the colors of the dashes (and the arrow, >). The arrow will now appear as shown in Figure 14.12b. After another short delay in line 130 the process is repeated by branching to line 80. Modify this program by using all four color registers 0–3 so that four different colored dashes occur in a row. Make it appear as if the four colors move from left to right.

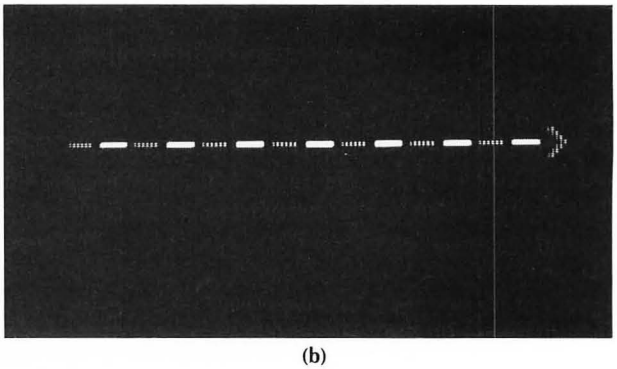
FIGURE 14.11 Moving arrow program using text 2 mode.

```
10 REM COLOR ARROW
20 GRAPHICS 2
30 FOR X=2 TO 14 STEP 2
40 COLOR 45:PLOT X,4
50 COLOR 13:PLOT X+1,4
60 NEXT X
70 COLOR 62:PLOT 16,4
80 SETCOLOR 0,0,14:REM WHITE
90 SETCOLOR 1,7,6:REM BLUE
100 FOR I=1 TO 80:NEXT I
110 SETCOLOR 0,7,6:REM BLUE
120 SETCOLOR 1,0,14:REM WHITE
130 FOR I=1 TO 80:NEXT I
140 GOTO 80
```

FIGURE 14.12 Screen display while running program shown in Figure 14.11.



(a)



(b)

Table 14.3 shows the internal character set of the ATARI. This character set is stored in ROM starting at memory location 57344. Each character uses eight consecutive memory locations for its definition. The character is defined on the 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.

FIGURE 14.13 PEEKing the eight values used to define the letter A.

```
LIST

10 FOR J=0 TO 7
20 ? PEEK(57344+8*33+J)
30 NEXT J

READY
RUN
0
24
60
102
102
126
102
0

READY
```

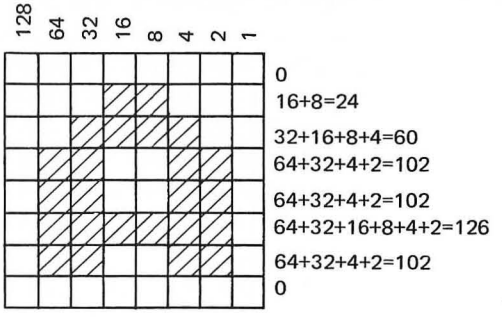
The characters given in Table 14.3 are listed in the order in which they are stored in ROM. Character A is number 33; therefore its definition starts at location 57344 + 8 * 33 (remember, each definition takes eight memory locations). In order to list the 8 bytes used to define the letter A, type in and run the following program, as shown in Figure 14.13:

```
10 FOR J=0 TO 7
20 ?PEEK(57344+8*33+J)
30 NEXT J
```

The relationship between these eight values and the letter A is shown in Figure 14.14. 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 ATARI defines the letter A (and all other characters), you may wonder if you can define your own characters. You can.

FIGURE 14.14 Defining the letter A.



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 ATARI where you moved it.

The beginning of the character set table is normally at location 57344. This starting value divided by 256 is stored in location 756. If you PEEK location 756 you should find the value 224 stored there (224 * 256 = 57344). When you POKE 756,226 in text modes 1 and 2 you are moving the beginning of the character set table to location 226 * 256 = 57856, which is the beginning of the character set starting in column 3 in Table 14.3.

You can move the beginning of the character set table to a memory location in RAM by POKEing the starting address divided by 256 in memory location 756. Suppose that you want to move the 512 bytes corresponding to the character set table of the 64 characters in columns 1 and 2 in Table 14.3 to some locations in RAM. The address of the top of RAM divided by 256 is stored in memory location 106. If you decrease this value by 2 (2 * 256 = 512), you can set aside 512 bytes of RAM into which you can move the character set table. The following four statements will make these changes:

```
855 NTOP=PEEK(106)-2
860 POKE 106,NTOP    :REM New top of RAM
865 NSET=NTOP*256    :REM Address of Start of
870 POKE 756,NTOP    :REM new character set
```

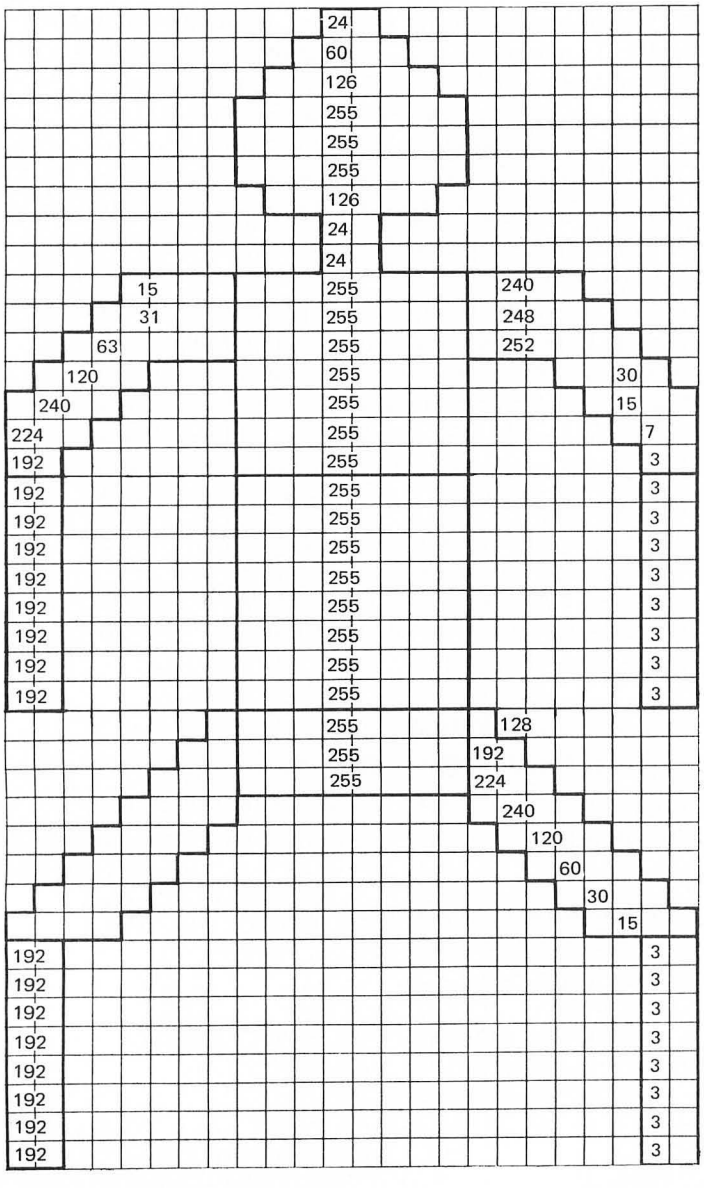
The following loop will move the 512 bytes of the character set in ROM to the new location in RAM:

```
875 FOR J=0 to 511
880 POKE NSET+J,PEEK(57344+J):NEXT J
```

In Chapter 15 we will write a HANGMAN game using the graphics mode 2. We want to use both upper-case and some graphic characters. But as we have seen, in graphics modes 1 and 2 you can use either the characters in columns 1 and 2 in Table 14.3 or the characters in columns 3 and 4. You cannot use both. Instead we will replace character numbers 1–10 (punctuation characters ! " # $ % & , ( ) *) with the 10 graphic characters of our own design shown in Figure 14.15. Each graphic character is defined on an 8×8 grid. The value of each row in this grid is given in Figure 14.15, using the technique illustrated in Figure 14.14.

The eight values for each of the 10 characters are stored in 10 DATA statements in the subroutine shown in Figure 14.16. This subroutine moves the 512 bytes of the original character set as described previously and then replaces character numbers 1–10 with the new graphic characters in lines 1585–1590.

FIGURE 14.15 Specially defined graphic characters used in the HANGMAN program in Chapter 15.




```
1500 REM DEFINE NEW CHAR SET
1505 DATA 24,60,126,255,255,255,126,24
1510 DATA 24,255,255,255,255,255,255,255
1515 DATA 255,255,255,255,255,255,255,255
1520 DATA 255,255,255,0,0,0,0,0
1525 DATA 0,15,31,63,120,240,224,192
1530 DATA 0,240,248,252,30,15,7,3
1535 DATA 1,3,7,15,30,60,120,240
1540 DATA 128,192,224,240,120,60,30,15
1545 DATA 192,192,192,192,192,192,192,192
1550 DATA 3,3,3,3,3,3,3,3
1552 POSITION 4,2:? #6;"PLEASE WAIT"
1555 NTOP=PEEK(106)-4
1560 POKE 106,NTOP
1565 NSET=NTOP*256
1570 FOR J=0 TO 511
1575 POKE NSET+J,PEEK(57344+J):NEXT J
1580 POKE 756,NTOP
1585 FOR K=0 TO 79
1590 READ B:POKE NSET+8+K,B:NEXT K
1595 RETURN
```

FIGURE 14.16 Subroutine used to define new character graphics.

In graphics mode 2 we can plot these graphics characters by plotting the corresponding character numbers (1–10) given in Table 14.5. For example, the program shown in Figure 14.17 will plot the hangman figure shown in Figure 14.18.

FIGURE 14.17 Program to plot new character graphic symbols.

(a)

```
10 REM PLOT NEW CHARACTER SYMBOLS
20 GRAPHICS 2:SETCOLOR 0,0,14
25 SETCOLOR 1,12,14:SETCOLOR 2,9,14
26 SETCOLOR 3,4,14
27 GOSUB 1500:REM DEFINE NEW CHAR SET
30 GOSUB 900:REM PLOT FIGURE
40 END
```

(b)

```
900 REM PLOT HANGMAN FIGURE
940 COLOR 33:PLOT 11,3
950 COLOR 2:PLOT 11,4
955 COLOR 3:PLOT 11,5
957 COLOR 4:PLOT 11,6
960 COLOR 165:PLOT 10,4
965 COLOR 169:PLOT 10,5
970 COLOR 166:PLOT 12,4
975 COLOR 170:PLOT 12,5
980 COLOR 135:PLOT 10,6
985 COLOR 137:PLOT 10,7
990 COLOR 136:PLOT 12,6
995 COLOR 138:PLOT 12,7
997 RETURN
```

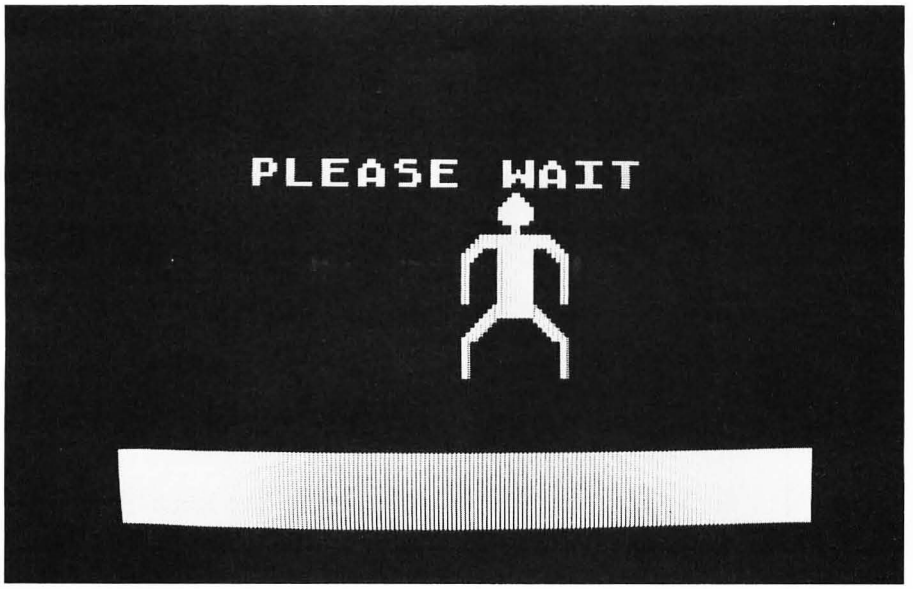


FIGURE 14.18 Result of running the program shown in Figure 14.17.

Writing Text on the High-Resolution Graphics Screen

In the graphics mode 8 used in Chapter 13 you saw that you could write text only in the four-line window at the bottom of the screen. The top part of the screen is divided into a 320×160 grid. Each of the 160 rows uses 40 8-bit bytes to store the graphics data (40 * 8 = 320).

Each byte of graphics data displays a bit pattern in the same way that each row of an 8×8 character does, as defined in Figure 14.14. This means that if the value of each row of a character is POKEd into the proper memory location corresponding to a position on the high-resolution graphics screen, then any character can be plotted on the high-resolution graphics screen.

The starting address of the screen display is given in locations 88 and 89. It is equal to

SA = PEEK(88) + 256 * PEEK(89)

Suppose that you want to plot the letter A at location X = 20 (X between 0 and 39) and Y = 100 (Y between 0 and 152). The address of the first byte of the character A (the top row) to be plotted will be

LA = SA + 40 * Y + X

The character A is stored at character number 33 in the character set table. The 8 bytes that define the letter A will begin at the address

CHAD = 57344 + 33 * 8

Therefore, the following loop will plot the entire letter A at location X,Y on the high-resolution screen:

```
FOR I=0 TO 7
POKE LA+I*40,PEEK(CHAD+I)
NEXT I
```

Remember that each line uses 40 bytes of memory so that the screen addresses increase by 40 from one row to the next. The program shown in Figure 14.19 will plot the letter A at location X = 20, Y = 100 on the high-resolution screen with a box around it. The result of running the program is shown in Figure 14.20.

Modify this program to display different letters. This technique will be used in the ATARI organ program written in Chapter 15 to display the letters on the keyboard.

FIGURE 14.19 Program to plot the letter A on the high-resolution graphics screen.

```
10 REM PROGRAM TO DISPLAY AN A
15 REM ON HI-RES GRAPHICS SCREEN
20 GRAPHICS 8:SETCOLOR 2,0,0
30 SETCOLOR 1,0,14:COLOR 1
40 PLOT 140,80:DRAWTO 185,80
50 DRAWTO 185,130:DRAWTO 140,130
60 DRAWTO 140,80
70 SA=PEEK(88)+256*PEEK(89)
80 X=20:Y=100
90 LA=SA+40*Y+X
100 CHAD=57344+33*8
110 FOR I=0 TO 7
120 POKE LA+I*40,PEEK(CHAD+I)
130 NEXT I
```

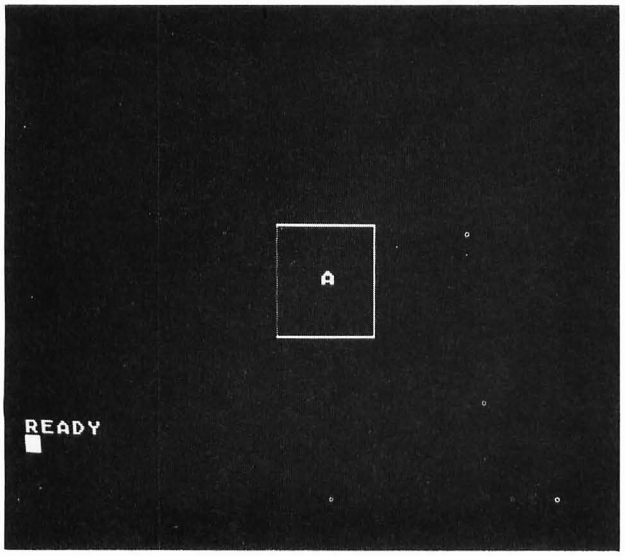


FIGURE 14.20 Result of running the program shown in Figure 14.19.

EXERCISE 14.1
Write a program that bounces a "ball" off the four sides of the screen at 45 degrees. Have the ball stop each time you hold the OPTION key down. The ball should start moving again when you release the key.

EXERCISE 14.2
Modify the program in Figure 5.7 so that the cursor does not appear on the screen after the border is drawn.

EXERCISE 14.3
Modify Exercise 14.1 so that the ball starts and stops each time any key is pressed.

Exercise 14.4
Plot a rectangular marquee in four colors in graphics mode 2 and make it appear to rotate, using the technique shown in Figure 14.11.

EXERCISE 14.5
Define your own graphic characters and use them to plot a house in graphics mode 2.

EXERCISE 14.6
Define a new character set in graphics mode 2 that replaces the punctuation marks !, ", #, and $ with the four playing card graphic symbols.

EXERCISE 14.7
Write a program that will replace the letter A in Figure 14.20 with the letter T.

15

LEARNING TO PUT IT ALL TOGETHER

In the previous 14 chapters you have learned how to use the various features of ATARI BASIC. You have learned how to write short programs that draw pictures and make sound. 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 ATARI 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. Although both programs are useful and fun to run, it is the process of developing the programs that we are trying to illustrate in this chapter.

The first program is a popular word-guessing game called HANGMAN. The second program converts your ATARI into an organ that will allow you to play songs on the keyboard.

In this chapter you will learn

1. how to define what you want to do and give a word-description of the program
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 diskette
6. how to play music on your ATARI.

HANGMAN

We will now develop a program to play the word-guessing game HANGMAN. The following six steps will help you develop a program with the minimum amount of difficulty. We will follow these six steps in developing HANGMAN:

1. Define *what* you want the program to do.
2. Give a word-description of the program.
3. Define program variable names.
4. Write and test the main program and essential subroutines.
5. Write and test the remaining subroutines.
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 don't have a clear idea of what you want the program to do, you will have little chance of writing a program to do it. Now you may not know all of the features that your program will eventually have. Indeed, programming is an iterative process in which 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 ATARI thinks of a word and displays a blank for each letter in the word. You guess a letter. If the letter occurs in the word, it is inserted at all locations where it occurs in the word. 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 guess six wrong letters, at which time your body is complete and you are hanged.

The first thing to decide is 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. Figure 15.1 shows the screen layout for the HANGMAN game where we have decided to use the text mode GR. 2 and plot the gallows and body parts using character graphics.

When the program is first executed the name HANGMAN is written at the top of the screen. The program then displays the gallows, the blanks for the word, and the words GUESS A LETTER. As each correct letter is guessed, it is filled in at the appropriate blank position or positions. When an incorrect letter is guessed, the phase NO X will be printed on the bottom line 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. If you fail to guess the word, the correct word will be displayed above the blanks and the words YOU ARE HANGED! will be printed. The words PLAY AGAIN???? will then be displayed where the words GUESS A LETTER were printed.

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. You should study this word-description carefully.

FIGURE 15.1 Screen layout for HANGMAN program.

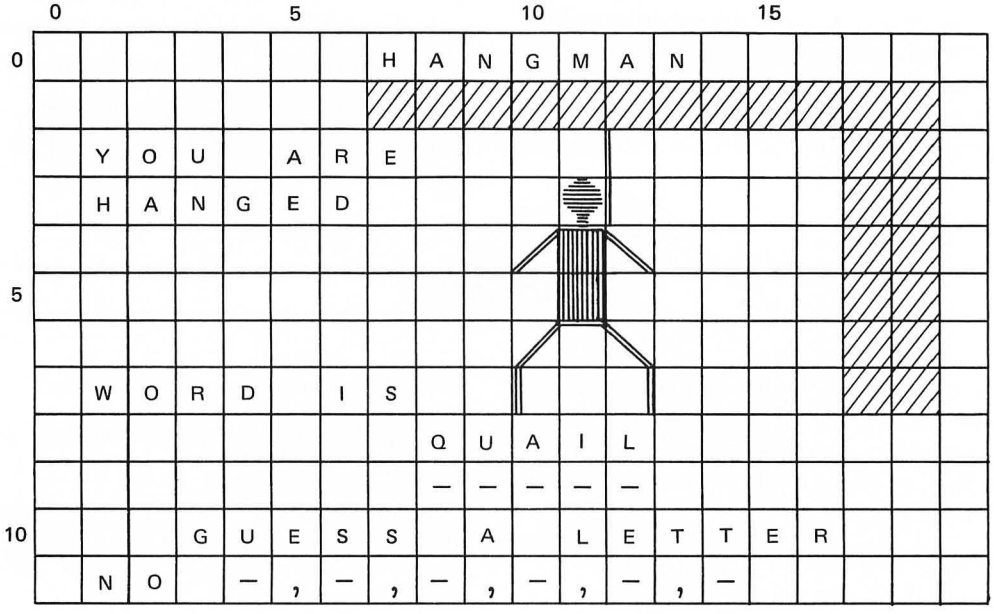


FIGURE 15.2 Pseudocode word-description of HANGMAN program.

```
loop: Clear screen
      Display HANGMAN
      Display gallows
      Display the words GUESS A LETTER
      Find a random word
      Display blanks 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 posi-
               tion
               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"
```

Note that it follows closely 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.

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 won't know all of the variables you will end up using, but don't worry about that. Define the important ones you do know. This will help you to focus in 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$. The length of this string (the number of letters in the word) will be L. How can you tell when the word is guessed correctly or when it's time to be hanged? You will need to keep track of the number of blanks that have been correctly filled in. We will call this value NL. You will also need to keep track of the number of incorrect guesses. We will call this value NH. Each letter guessed will be stored in the string G$. You can determine if a guessed letter G$ is in the word W$ by comparing G$ with each letter in W$ and noting where any matches occur. You will also need to know if any match occurs. For this purpose define a flag R and set R to 1 if any match occurs; set R to 0 if G$ is not in W$. At this point, therefore, we have 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.

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$
```

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 letter that is found in its proper position, so nothing more needs to be done in the *then* part of the following *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, at this point 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. Your goal should be to write this *entire* program so that it fits on a single page and you can read it all at once. To do this use subroutine calls for anything that takes a lot of coding or that you haven't figured out how to do yet.

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$
   R = 0
   for I = 1 to L
        if G$ = W$(I,I)
        then print G$
             NL = NL + 1
             R = 1
        else move cursor 1 space
   next I
   if R = 1
   then do nothing
   else NH = NH + 1
             display NO "letter"
             add part to body
enddo
```

The main program for HANGMAN is shown in Figure 15.5. Lines 20–26 set the graphics mode 2 to full screen and set color registers 0–3. Line 27 calls a subroutine at line 1500 that defines a new character set. This character set includes the graphic characters for forming the hangman figure defined in Chapter 14. Line 30 displays the word HANGMAN and the gallows in subroutine 600. Line 40 finds a random word W$ in subroutine 1000. Line 50 finds the length, L, of the word W$ and moves the cursor to the first "blank" position. In text mode 2 there are 20 column positions. Therefore, the statement POSITION 10-L/2,9 will center each word on the screen. Line 60 prints the L blanks 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 70 was added when the subroutine to guess a letter in line 400 was written.) Line 90 calls subroutine 400 to guess a letter. The words GUESS A LETTER that appear on the screen will be written in this subroutine. Line 100 moves the cursor to the first letter position in the word and line 105 sets the flag R to 0. Lines 110–130 implement the *for . . . next* loop given in Figure 15.4. Note how the statement POSI-

FIGURE 15.5 Main program for HANGMAN.

```
10 REM HANGMAN
12 OPEN #1,4,0,"K:"
15 DIM W$(15),G$(1),S$(20)
20 GRAPHICS 2+16:SETCOLOR 0,0,14
25 SETCOLOR 1,12,14:SETCOLOR 2,9,14
26 SETCOLOR 3,4,14
27 GOSUB 1500:REM DEFINE NEW CHAR SET
30 GOSUB 600:REM DISPLAY HANGMAN & GALLOWS
40 GOSUB 1000:REM FIND WORD W$
50 L=LEN(W$):POSITION 10-L/2,9
60 FOR I=1 TO L:? #6;"-";:NEXT I
70 NL=0:NH=0:S$=""
80 IF NL=L OR NH=6 THEN 160
90 GOSUB 400:REM GUESS A LETTER
100 X=10-L/2:POSITION X,9
105 R=0
110 FOR I=1 TO L
112 X=X+1
115 IF G$=W$(I,I) THEN ? #6;G$;:NL=NL+1:R=1:GOTO 130
120 POSITION X,9
130 NEXT I
140 IF R=1 THEN 80
150 NH=NH+1:GOSUB 900:GOTO 80
160 POSITION 1,2:? #6;"YOU ARE"
170 IF NH=6 THEN ? #6;" HANGED";:GOSUB 300:GOTO 190
180 ? #6;" SAVED  ";:GOSUB 700
190 POSITION 3,10
195 ? #6;"PLAY AGAIN???? ";
200 GET #1,G:IF G=ASC("Y") THEN 30
210 CLOSE #1:GRAPHICS 0:END
```

TION X,9 is used in line 120 to move the cursor one space in the *else* clause. The value of X is incremented by 1 each time it passes through the *for . . . next* loop (in line 112).

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. Lines 190–195 will ask to PLAY AGAIN???? Line 200 will then get G and cause a new game to be played if the answer to PLAY AGAIN???? is "Y".

We have therefore written a complete main program that implements the HANGMAN algorithm given in Figure 15.2. We have also identified all the subroutines that must still be written. These are summarized in Figure 15.6.

FIGURE 15.6 List of subroutines called from main program.

Line No.	*Subroutine*
1500	Define new character set
600	Initial display (HANGMAN, gallows)
1000	Find a word, W$
400	Guess a letter, G$
900	Wrong guess-NO "letter," add to body
300	Print correct word
700	Blink word

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* could be just 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 write and test each subroutine separately. They can then be tested, of course, by running the main program which calls the subroutine. This technique of *top-down programming* allows you to plan the entire program and begin to test it before you have to get 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.

Subroutine 1500 will be the one used in Chapter 14 for defining a new character set. However, for now just type

```
1500 REM DEFINE NEW CHAR SET
1510 RETURN
```

In subroutine 600, it will take some thought to figure out how to draw the gallows. Therefore, for now, just type

```
600 REM INITIAL DISPLAY
670 RETURN
```

for subroutine 600 and worry about the details later.

In order to test the main program, you should store a known word in W$. Therefore, for subroutine 1000 type

```
1000 REM FIND A WORD
1010 W$="HANGMAN"
1020 RETURN
```

which will assign the word HANGMAN to W$. 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 400 will display the words GUESS A LETTER and will then have the player guess a letter, G$. At first it looks as if this is just the statement GET G:G$=CHR$(G). However, you do not want to allow letters that have already been guessed. (Otherwise, you could hang yourself by typing the same wrong letter six times.) Therefore, subroutine 400 must keep track of all letters that have been typed and only return new values for G$. We'll figure out how to do this later. For now just type

```
400 REM GUESS A LETTER
410 POSITION 3,10
420 PRINT #6;"GUESS A LETTER";
430 GET #1,G
435 G$=CHR$(G)
440 RETURN
```

For subroutine 900 type

```
900 REM WRONG GUESS
910 POSITION 1,11
920 PRINT #6;"NO";G$
930 RETURN
```

You know that this will print all wrong guesses at the same location on the screen, but it will help test the main program. You can fix it up later and figure out how to add a new part to the body each time.

For subroutines 300 and 700 just type the stubs

```
300 REM PRINT CORRECT WORD
310 RETURN
```

and

```
700 REM BLINK WORD
710 RETURN
```

and worry about these subroutines later.

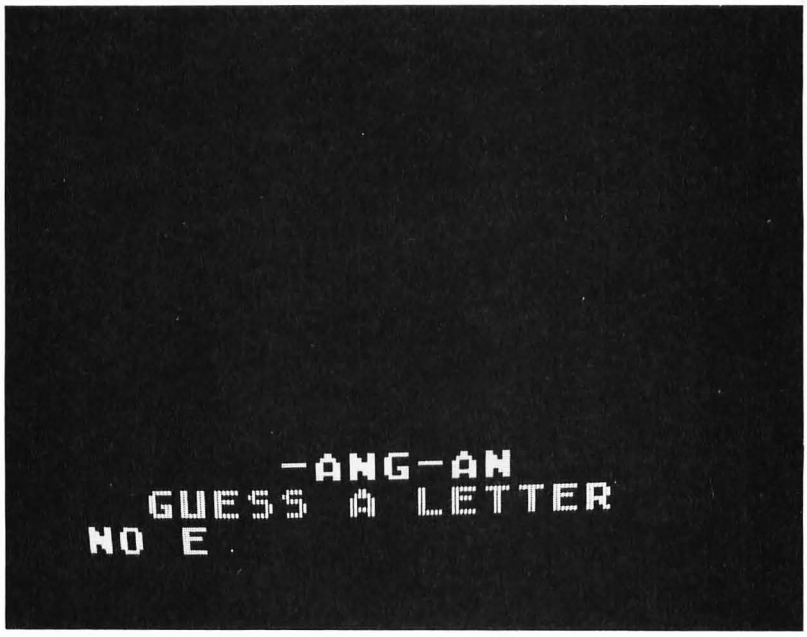


FIGURE 15.7 Testing the main program of HANGMAN.

With this much of the program written, you can run the main program and test that it is working properly. Figure 15.7 shows what the screen might look like during such a test.

After you have debugged the main program (it won't work the first time—Figure 15.5 was *not* my first version) you are ready to tackle the remaining subroutines one by one.

Writing the Remaining Subroutines

You can now go through and finish the subroutines listed in Figure 15.6. Subroutine 1500 will be the one given in Figure 14.16 in Chapter 14 that we used to define the new character set. This subroutine is shown in Figure 15.8 where one new statement, line 1502, has been added. The reason for this statement

FIGURE 15.8 Subroutine to define new graphic characters for the HANGMAN program.

```
1500 REM DEFINE NEW CHAR SET
1502 NW=10:FOR I=1 TO NW:READ W$:NEXT I
1505 DATA 24,60,126,255,255,255,126,24
1510 DATA 24,255,255,255,255,255,255,255
1515 DATA 255,255,255,255,255,255,255,255
1520 DATA 255,255,255,0,0,0,0,0
1525 DATA 0,15,31,63,120,240,224,192
1530 DATA 0,240,248,252,30,15,7,3
1535 DATA 1,3,7,15,30,60,120,240
1540 DATA 128,192,224,240,120,60,30,15
1545 DATA 192,192,192,192,192,192,192,192
1550 DATA 3,3,3,3,3,3,3,3
1552 POSITION 4,5:? #6;"PLEASE WAIT"
1555 NTOP=PEEK(106)-4
1560 POKE 106,NTOP
1565 NSET=NTOP*256
1570 FOR J=0 TO 511
1575 POKE NSET+J,PEEK(57344+J):NEXT J
1580 POKE 756,NTOP
1585 FOR K=0 TO 79
1590 READ B:POKE NSET+8+K,B:NEXT K
1595 RETURN
```

```
600 REM INITIAL DISPLAY
605 COLOR 0:FOR Y=0 TO 11:FOR X=1 TO 19:PLOT X,Y:NEXT X:NEXT Y
PLOT X,Y:NEXT X:NEXT Y
610 COLOR 163
620 FOR X=7 TO 18
630 PLOT X,1:NEXT X
640 FOR Y=2 TO 7
650 PLOT 17,Y:PLOT 18,Y
660 NEXT Y
665 POSITION 7,0:? #6;"HANGMAN"
670 RETURN
```

FIGURE 15.9 Initial display subroutine that erases the screen, draws the gallows, and prints the word HANGMAN.

will be described later. Omit it until the subroutine in Figure 15.14 has been written. Recall that the subroutine in Figure 15.8 defines the new graphic characters shown in Figure 14.15. These characters can be plotted by using the COLOR numbers for character numbers 1–10 in Table 14.5.

Figure 15.9 is a listing of subroutine 600. Line 605 erases the screen from a previous game. Note that line 200 in the main program (Figure 15.5) branches to line 30. Branching to line 20 would automatically clear the screen. However, it would also reset the address of the start of the character set table to 57344. Line 27 would then have to move the character set table again. This takes quite a long time. Branching to line 30 from line 200 avoids this. Lines 610–660 draw the blue gallows. Note that 163 is the color number for character number 3 (solid square) of our new character set. Line 665 prints the word HANGMAN at the top of the screen.

The "guess a letter" subroutine 400 is shown in Figure 15.10, where lines 435–470 have been added to ensure that no letter is guessed more than once. Line 470 keeps track of all letters that have been guessed by adding each new letter to the string S$.

FIGURE 15.10 The subroutine to guess a letter.

```
400 REM GUESS A LETTER
410 POSITION 3,10
420 ? #6;"GUESS A LETTER ";
430 GET #1,G
432 G$=CHR$(G)
435 LS=LEN(S$)
437 IF LS=0 THEN 470
440 FOR J=1 TO LS
450 IF G$=S$(J,J) THEN 430
460 NEXT J
470 S$(LS+1)=G$
480 RETURN
```

FIGURE 15.11 The "wrong guess" subroutine prints NO "letter" and adds a part to the body.

```
900 REM WRONG GUESS
910 POSITION 1,11
915 IF NH=1 THEN ? #6;"NO ";G$;:GOTO 935
920 POSITION 2*NH+1,11
930 ? #6;",";G$;
935 ON NH GOTO 940,950,960,970,980,990
940 COLOR 33:PLOT 11,3:RETURN
950 COLOR 2:PLOT 11,4
955 COLOR 3:PLOT 11,5
957 COLOR 4:PLOT 11,6:RETURN
960 COLOR 165:PLOT 10,4
965 COLOR 169:PLOT 10,5:RETURN
970 COLOR 166:PLOT 12,4
975 COLOR 170:PLOT 12,5:RETURN
980 COLOR 135:PLOT 10,6
985 COLOR 137:PLOT 10,7:RETURN
990 COLOR 136:PLOT 12,6
995 COLOR 138:PLOT 12,7
997 COLOR 41:PLOT 12,2:PLOT 12,3:RETURN
```

(This is why we initialized S$ to the null string "" in line 70 of the main program.) Each time that lines 430–432 get a new letter G$, it is compared with all previous letters (stored in S$) in the loop in lines 440–460. If a match is found in line 450, the program gets a new letter in line 430.

The "wrong guess" subroutine 900 is shown in Figure 15.11. Lines 910–930 print NO "letter" at the bottom of the screen for the first wrong guess. Subsequent wrong guesses are added to the list following a comma. Lines 935–995 add the appropriate part of the body to the hanging person. Note the use of the ON . . . GOTO statement in line 935 to add the appropriate part of the body shown in Figure 15.1 and defined in detail in Chapter 14. This statement branches to one of the line numbers following GOTO, depending on the value of NH. If NH = 1 it branches to line 940 (the first number), if NH = 2 it branches to line 950 (the second number), and so forth. Line 940 plots the head (first wrong guess); lines 950–957 plot the body (second wrong guess); lines 960–965 plot the right arm (on your left—third wrong guess); lines 970–975 plot the left arm (fourth wrong guess); lines 980–985 plot the right leg (fifth wrong guess); lines 990–995 plot the left leg (sixth and last wrong guess); line 997 plots the rope that does the hanging.

Subroutine 300, shown in Figure 15.12, prints the correct word above the blanks when the person is hanged. Subroutine 700, shown in Figure 15.13, blinks the word that was guessed.

Lines 705–730 change the color number of each letter in the word by adding 128 to the current color number. This will make the color of each letter controlled by color register 2. The statement LOCATE X,Y,AC in line 715 returns a value AC equal to the color number of the character at location X,Y. Line 720 adds 128 to the color number of each letter in the word and line 725 replots the letter. Lines 735–760 blink the word by changing the color in color register 2. When this subroutine is called, the gallows and parts of the body will also blink because they are also controlled by color register 2 (see Table 14.5).

FIGURE 15.12 This subroutine prints the correct word above the blanks.

```
300 REM PRINT CORRECT WORD
310 POSITION 1,7
320 ? #6;"WORD IS"
325 POSITION 10-L/2,8
330 ? #6;W$
340 RETURN
```

FIGURE 15.13 This subroutine blinks the word.

```
700 REM BLINK WORD
705 X1=10-L/2
710 FOR X=X1 TO X1+L
715 LOCATE X,9,AC
720 AC=AC+128
725 COLOR AC:PLOT X,9
730 NEXT X
735 FOR I=1 TO 40
740 SETCOLOR 2,3,14
745 FOR J=1 TO 10:NEXT J
750 SETCOLOR 2,9,14
755 FOR J=1 TO 10:NEXT J
760 NEXT I
770 RETURN
```

If all of these 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 15.14. Line 1020 defines the number of words NW stored in the DATA statement 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. Note that the DATA statements in line 1100 must be the first DATA statements in the program. There are others at lines 1505–1550. This is why you must now add line 1502 in Figure 15.8, so that the subroutine at line 1500 will skip over the first 10 DATA statement values in lines 1100–1110. Line 1050 reads the first X words. Therefore, word number X will end up in W$. Note that with this subroutine

FIGURE 15.14 Subroutine that finds one of 10 words at random.

```
1000 REM FIND A WORD
1020 NW=10
1030 X=INT(RND(0)*NW+1)
1040 RESTORE
1050 FOR I=1 TO X:READ W$:NEXT I
1060 RETURN
1100 DATA HIPPOPOTAMUS,NURSE,FAMOUS,EMPIRE,ELK,DIGNITY
1110 DATA CONDITIONAL,BRIBE,PAPER,QUAIL
```

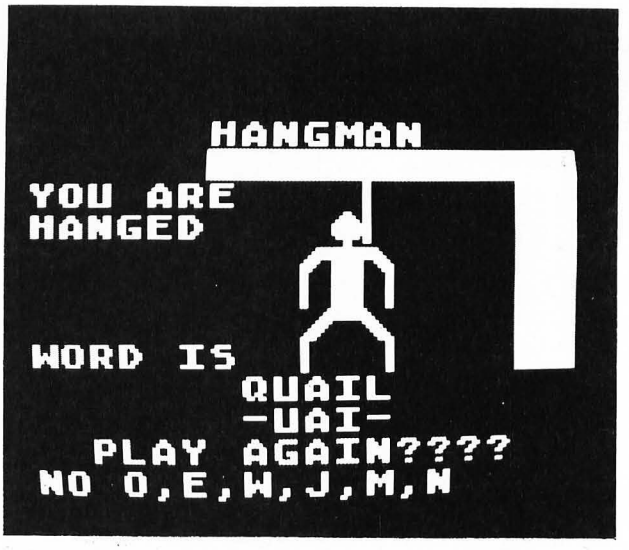


FIGURE 15.15 Sample run of HANGMAN program.

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 be able to play only 10 times before having to rerun the program. A sample run of this program is shown in Figure 15.15.

It is clear that to make this game really interesting you need a large dictionary of possible words so that you will use different words each time you play. One way to do this is to store a large number of words on a diskette and then read in a random word each time the game is played. The next section will show you how to store a list of words on a diskette.

STORING DATA ON A DISKETTE

You may have been using diskettes to save and load your BASIC programs. It is also possible for you to include statements in your programs that will allow you to store *data* on a diskette and later read back these data. You do this using the PRINT # and INPUT # statements. However, in order to use these statements you must use first the OPEN statement and then the CLOSE statement. In the following sections you will learn how to use the statements OPEN, CLOSE, PRINT #, INPUT #, and TRAP.

Storing Words in a Sequential File

The program shown in Figure 15.16 gives you the option to (1) write words to a new file, (2) add words to an existing file, or (3) read words from an existing file. Type in this program and add the three stubs

```
1000 RETURN
2000 RETURN
3000 RETURN
```

Executing this program will produce the menu shown in Figure 15.17.

FIGURE 15.16 BASIC listing of main program illustrating the use of sequential files.

```
10 REM STORING WORDS IN A
12 REM SEQUENTIAL FILE
15 DIM A$(1),N$(20),F$(23),W$(25)
20 OPEN #1,4,0,"K:"
25 ? "}":POSITION 5,3
30 ? "1. WRITE WORDS TO A NEW FILE"
35 POSITION 5,5
40 ? "2. ADD WORDS TO AN EXISTING FILE"
45 POSITION 5,7
50 ? "3. READ WORDS FROM EXISTING FILE"
55 POSITION 5,9
60 ? "4. EXIT PROGRAM":? :?
70 GOSUB 200:REM SELECT NUMBER
80 IF A$="4" THEN ? "}":CLOSE #1:END
85 I=VAL(A$)
90 ON I GOSUB 1000,2000,3000
95 GOTO 25
200 REM PICK A NUMBER
210 ? "SELECT A NUMBER";:? " ";
220 GET #1,A
230 A$=CHR$(A)
240 IF A$<"1" OR A$>"4" THEN 220
250 RETURN
```

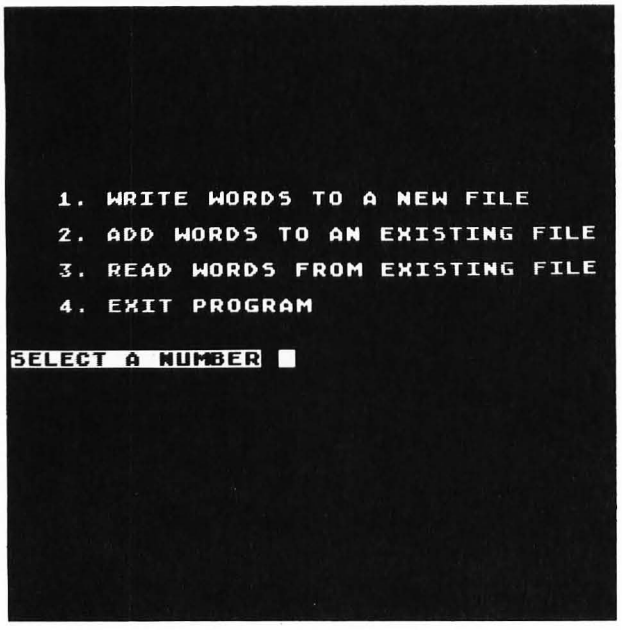


FIGURE 15.17 Menu produced when the program in Figure 15.16 is executed.

The subroutine at line 200 (which is called in line 70) waits for the user to press a key. If the key pressed (A$) is a number between 1 and 4, the subroutine is exited. If A$ = "4" (key 4 was pressed), line 80 clears the screen and terminates the program.

The statement

ON I GOSUB 1000,2000,3000

in line 90 will branch to the subroutine at line 1000, 2000, or 3000, depending upon whether I is 1, 2, or 3. Thus, if key 1 was pressed, the value of I will be 1 and the program will branch to the subroutine at line 1000. We only have the stub RETURN there now, so if you press key 1 the program will immediately return to line 90 and then branch back to line 25.

A subroutine at line 1000 that will allow you to write words into a new file is shown in Figure 15.18. Lines 1010–1030 cause the messages shown in Figure 15.19 to be displayed on the screen. Lines 1040–1050 allow the user to return to the main program (and the original menu) at this point by pressing any key other than D. This is a good option to give a user who may not be prepared to actually write data on a diskette at this time.

If the user presses key D, line 1060 will ask the user to type in a file name. This file name will be stored in N$ in line 1070 and will be the file in which the words will be written. Before any words can be written to this file, however, the file must be opened. This is done using the following statement given in line 1080:

OPEN #2,8,0,F$

```
1000 REM CREATE NEW FILE
1010 GOSUB 1200:REM WRITE SETUP
1020 ? "PRESS KEY 'D' TO STORE WORDS"
1025 ? "ON DISKETTE":?
1030 ? "PRESS ANY OTHER KEY TO EXIT"
1040 GET #1,A
1045 A$=CHR$(A)
1050 IF A$<>"D" THEN RETURN
1060 ? :? "WHAT FILE NAME? ";
1070 INPUT N$
1075 F$="D1:":F$(4)=N$
1080 OPEN #2,8,0,F$
1100 GOSUB 1300:REM ENTER WORDS
1110 CLOSE #2
1120 RETURN
1200 REM WRITE SETUP
1210 ? "}":POSITION 2,5
1220 ? "INSERT DISKETTE ON WHICH WORDS"
1230 ? "ARE TO BE SAVED":?
1240 RETURN
1300 REM ENTER WORDS
1310 ? "ENTER WORDS; TYPE ! TO STOP"
1320 INPUT W$
1330 IF W$="!" THEN RETURN
1340 ? #2;W$
1350 GOTO 1320
```

FIGURE 15.18 Subroutine to write words into a new file.

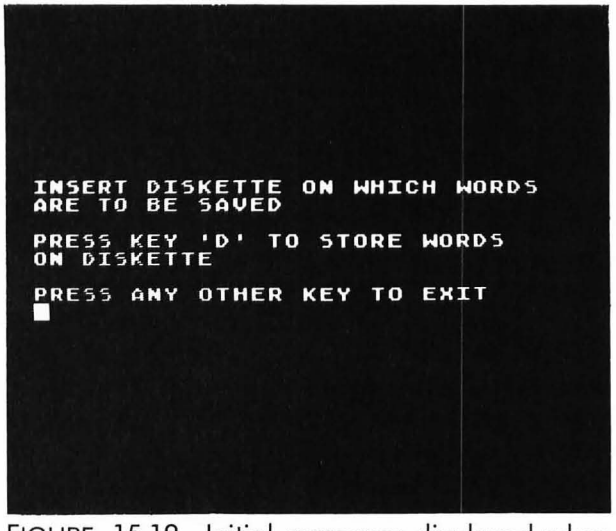


FIGURE 15.19 Initial messages displayed when the subroutine in Figure 15.18 is executed.

In this statement F$ is the file name in the form "D1:FILENAME", where FILENAME is the name entered as N$ in line 1070. Note how the statements

F$="D1:":F$(4)=N$

are used in line 1075 to form the equivalent of

F$="D1:FILENAME"

The OPEN statement is of the general form

OPEN #*fileno, code, aux, dev*

where *fileno* is a file number between 1 and 7, *code* is a code number given in Table 15.1, *aux* is an auxiliary code that is normally 0, and *dev* is a device designation given in Table 15.2.

TABLE 15.1 Code values in the OPEN statement

Code	*Operation*
4	Input (read)
8	Output (write)
12	Input and output (read and write)
6	Read disk directory
9	Append to end of file

TABLE 15.2 Device designations in OPEN statement

Device	*dev Designation*
Disk file	"D[n]:*filename*[.*ext*]"
Keyboard	"K:"
Display screen	"S:"
Printer	"P:"
Screen editor	"E:"
Recorder	"C:"
RS-232 serial port	"R[n]"

Therefore, the statement

OPEN #2,8,0,F$

directs you to open the file "D1:N$" for writing (output) and give it the file number 2. After writing to this file, you must close it using the statement

CLOSE #2

Once the file N$ is opened, the subroutine at line 1300 is used to enter a list of words. This subroutine allows you to enter as many words as you wish. You type an exclamation point (!) to indicate the end of the list. Each word is stored in W$ using the INPUT statement in line 1320. The statement

PRINT #2;W$

will not print the word W$ on the TV screen, but rather will write this word into the diskette file N$. Line 1350 branches back to the INPUT statement in line 1320.

Type in the subroutine shown in Figure 15.18 and execute it by pressing key 1 after running the main program. An example of the screen output while this subroutine is being executed is shown in Figure 15.20.

If you enter an exclamation point in line 1320, line 1330 will cause the subroutine to return to line 1110. This line closes the file N$, using the CLOSE statement.

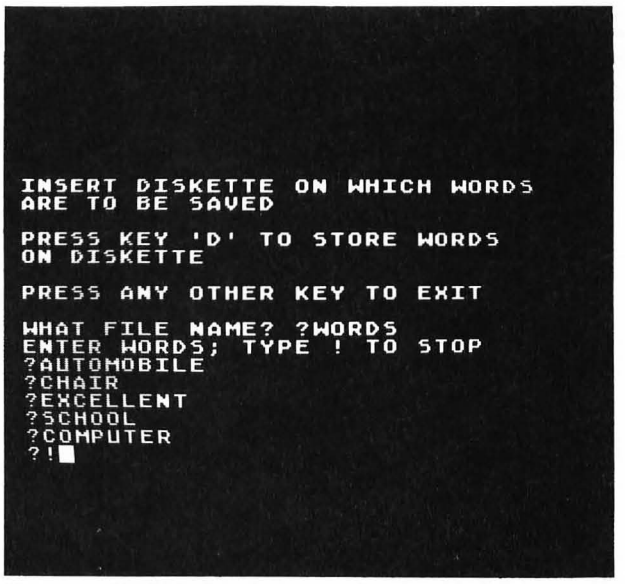


FIGURE 15.20 Storing words on a diskette using the subroutine given in Figure 15.18.

Suppose that after writing some words into the file you want to *add* some more words at a later time. You cannot call the subroutine at line 1000 again because this will delete your old file and start a new one. You will lose all of your old words. Instead, you must append words to the existing file by using the code 9 in the OPEN statement. This is illustrated in line 2075 of the subroutine shown in Figure 15.21.

FIGURE 15.21 Subroutine to add words to an existing file.

```
2000 REM ADD WORDS TO FILE
2010 GOSUB 1200:REM WRITE SETUP
2020 ? "PRESS KEY 'A' TO ADD WORDS"
2030 ? "PRESS ANY OTHER KEY TO EXIT"
2040 GET #1,A
2045 A$=CHR$(A)
2050 IF A$<>"A" THEN RETURN
2060 ? :? "WHAT FILE NAME ";
2065 INPUT N$
2070 F$="D1:":F$(4)=N$
2075 OPEN #2,9,0,F$
2080 GOSUB 1300:REM ENTER WORDS
2090 CLOSE #2
2095 RETURN
```

This subroutine starts at line 2000 and is called in line 90 of the main program when key 2 is pressed.

Type in this subroutine and then add some more words to your existing word file. Using this program, you can build up a large file of words to be used in the HANGMAN program. However, before we can use these stored words in the HANGMAN program we must learn how to *read* the words from the diskette.

Reading Words from a Sequential File

When key 3 is pressed in response to the menu in the main program in Figure 15.16, line 90 branches to the subroutine in line 3000. If at line 3000 we write the subroutine shown in Figure 15.22, this subroutine will produce a second menu, shown in Figure 15.23.

FIGURE 15.22 Subroutine to read words.

```
3000 REM READ WORDS
3010 ? "}":POSITION 2,5
3020 ? "1. READ ENTIRE FILE":?
3030 ? "2. READ N WORDS STARTING"
3035 ? "   AT LOCATION L":?
3040 ? "3. READ 1 WORD AT RANDOM":?
3050 ? "4. QUIT":? :?
3060 GOSUB 200:REM PICK A NUMBER
3070 IF A$="4" THEN RETURN
3075 I=VAL(A$)
3080 ON I GOSUB 3100,3200,3300
3090 GOTO 3010
```

FIGURE 15.23 Menu produced by the subroutine given in Figure 15.22.

Note that this subroutine gives you three choices other than returning to the main menu. This technique of using menus is a good way to steer a user through a large program. It is also a good way to keep the organization of your program under control.

To read the entire file, the user presses key 1. This will cause line 3080 to branch to the subroutine at line 3100. This subroutine is shown in Figure 15.24. Line 3110 calls the subroutine at line 4000 that is shown in Figure 15.25. This subroutine displays the messages shown in Figure 15.26.

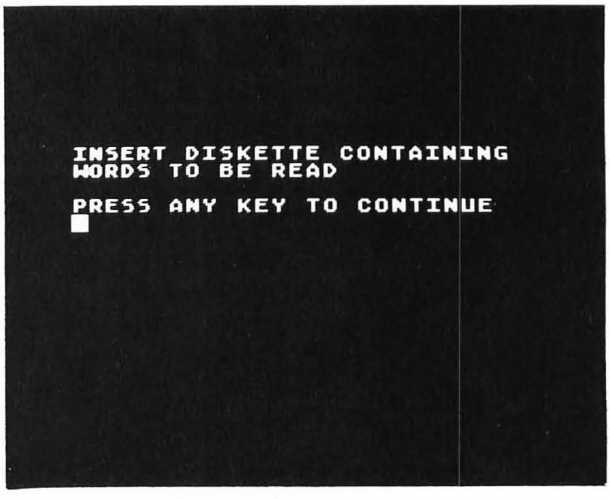
```
3100 REM READ ENTIRE FILE
3105 ? "}"
3110 GOSUB 4000:REM READ SETUP
3120 OPEN #2,4,0,F$
3130 INPUT #2,W$
3140 ? W$
3150 GOTO 3130
```

FIGURE 15.24 Subroutine to read entire file.

FIGURE 15.25 Subroutine to display initial messages for reading data.

```
4000 REM READ SETUP
4010 ? "INSERT DISKETTE CONTAINING"
4020 ? "WORDS TO BE READ":?
4030 ? "PRESS ANY KEY TO CONTINUE"
4040 GET #1,A
4050 ? :? "WHAT FILE NAME ";
4060 INPUT N$
4070 TRAP 6000
4080 F$="D1:":F$(4)=N$
4090 RETURN
```

FIGURE 15.26 Messages displayed by the subroutine shown in Figure 15.25.



Line 3120 in Figure 15.24 opens file N$. The statement

INPUT #2,W$

will accept its characters from the specified sequential disk file rather than from the keyboard. Thus, in line 3130 in Figure 15.24, the string that is stored in W$ will be the next word in the file N$. Line 3140 will print this word on the screen. Line 3150 branches back to line 3130, which will input the next word from the disk file. Note there is *no exit* from this loop. When the program tries to read beyond the end of the file an error condition will result.

It is possible for you to have your program branch to a specified line number when an error is detected.

```
6000 REM ERROR HANDLING ROUTINE
6005 EC=PEEK(195)
6010 IF EC=136 THEN ? "END OF DATA":GOTO 6050
6020 ? "ERROR NO. ";EC
6050 CLOSE #2
6060 ? :? "PRESS ANY KEY TO CONTINUE ";
6070 GET #1,A
6080 RETURN
```

FIGURE 15.27 Error-handling routine executed when an error occurs after the statement TRAP 6000.

You do this by executing the TRAP statement before an error occurs. In line 4070 in Figure 15.25, the statement

TRAP 6000

will cause the program to jump to line 6000 whenever an error is detected. In particular, it will jump to line 6000 when it has read all of the words from the file N$ and tries to read beyond the end of the file. The error-handling routine at line 6000 is shown in Figure 15.27.

After an error occurs, an error code is stored in memory location 195. Line 6005 assigns this error code to the variable name EC. A list of all error codes is given in Appendix C. In particular, the error code 136 occurs when a program tries to read beyond the end of data in a disk file. This is what would happen eventually in line 3130 in Figure 15.24. When this occurs, line 6010 in Figure 15.27 will print the messsage

END OF DATA

and then close the file N$ in line 6050. Pressing any key will then return the program to line 3090, which will branch to line 3010 and redisplay the read menu. An example showing the entire file being read is given in Figure 15.28.

FIGURE 15.28 Reading the entire file.

```
INSERT DISKETTE CONTAINING
WORDS TO BE READ

PRESS ANY KEY TO CONTINUE

WHAT FILE NAME ?WORDS
AUTOMOBILE
CHAIR
EXCELLENT
SCHOOL
COMPUTER
END OF DATA

PRESS ANY KEY TO CONTINUE
```

Reading a Partial List of Words

Suppose that you have a data file containing a large number of words and you want to read N of these words, starting at location L. The subroutine shown in Figure 15.29 will do this. It is called from line 3080 in Figure 15.22 when key 2 is pressed.

In lines 3210–3225 the user enters the number of words N and the starting location L from the keyboard. Line 3232 calls the setup subroutine at line 4000, shown in Figure 15.25. After the file N$ has been opened in line 3235, the FOR . . . NEXT loop in lines 3240–3245 will read the first L − 1 words and discard them. The FOR . . . NEXT loop in lines 3250–3265 will read the next N words from the disk file and print them on the screen. Line 3270 closes the file and line 3285 waits for any key to be pressed before returning to line 3090 in Figure 15.22.

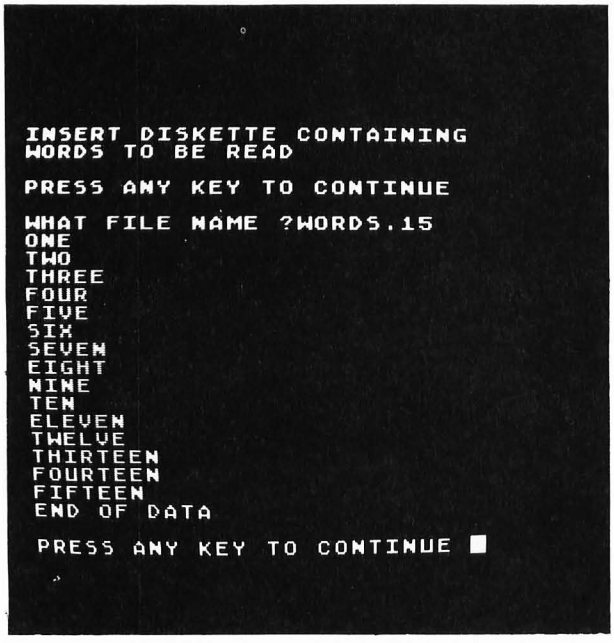
Note that if the subroutine in Figure 15.29 tries to read past the end of the file, line 4070 in Figure 15.25

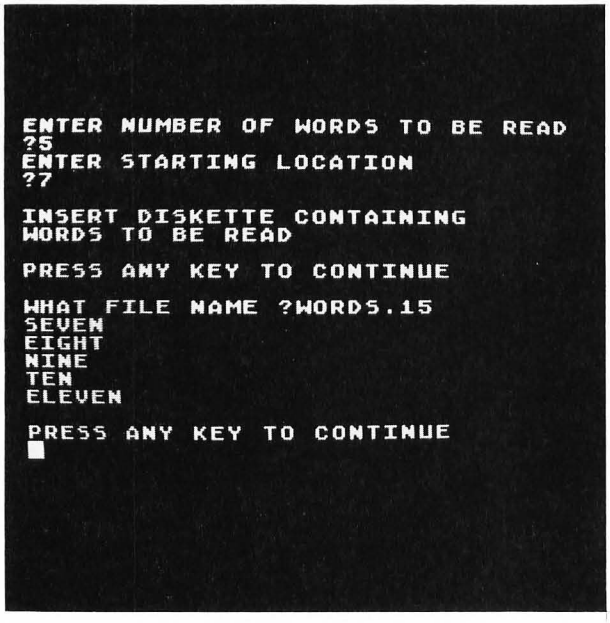
FIGURE 15.29 Subroutine to read N words from a data file starting at location L.

```
3200 REM READ N WORDS STARTING
3202 REM AT LOCATION L
3205 ? "¶"
3210 ? "ENTER NUMBER OF WORDS TO BE READ"
3215 INPUT N
3220 ? "ENTER STARTING LOCATION"
3225 INPUT L
3230 L=L-1
3232 ? :GOSUB 4000:REM READ SETUP
3235 OPEN #2,4,0,F$
3237 IF L=0 THEN 3250
3240 FOR I=1 TO L
3245 INPUT #2,W$:NEXT I
3250 FOR I=L+1 TO L+1+N-1
3255 INPUT #2,W$
3260 ? W$
3265 NEXT I
3270 CLOSE #2
3280 ? :? "PRESS ANY KEY TO CONTINUE"
3285 GET #1,A
3290 RETURN
```

will cause a branch to the error-handling routine in Figure 15.27. Execution of the subroutine in Figure 15.29 is shown in Figure 15.30.

FIGURE 15.30 Reading N words starting at location L.





In order to read a word selected at random from the file N$, we might read X − 1 dummy words and then read word number X, where X is some random integer. The subroutine shown in Figure 15.31 will do this. It is called in line 3080 in Figure 15.22 when key 3 is pressed. The user enters the maximum number of words in the file in line 3315. The usual read setup subroutine in Figure 15.25 is called in line 3320.

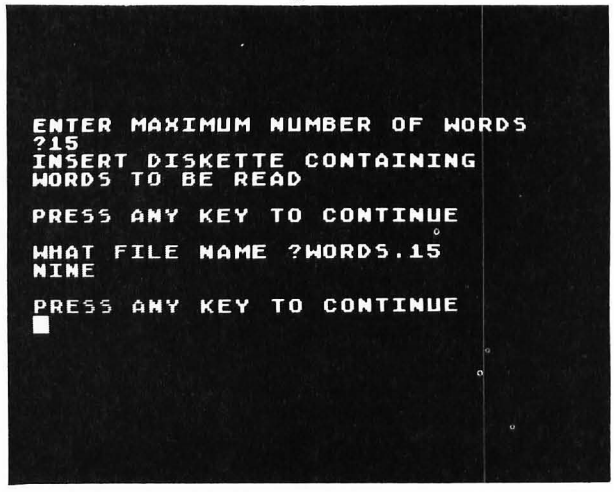
```
3300 REM READ 1 WORD AT RANDOM
3305 ? "¶"
3310 ? "ENTER MAXIMUM NUMBER OF WORDS"
3315 INPUT N
3320 GOSUB 4000:REM READ SETUP
3325 X=INT(RND(0)*N+1)
3330 OPEN #2,4,0,F$
3335 IF X=1 THEN 3360
3340 FOR I=1 TO X-1
3350 INPUT #2,W$:NEXT I
3360 INPUT #2,W$
3370 ? W$
3375 CLOSE #2
3380 ? :? "PRESS ANY KEY TO CONTINUE"
3385 GET #1,A
3390 RETURN
```

FIGURE 15.31 Subroutine to read one word at random.

Line 3325 finds a random number X between 1 and N. Line 3330 opens the file N$ for reading (code = 4). Lines 3340–3350 form a loop that reads X − 1 words. Line 3360 then reads word number X.

Figure 15.32 illustrates the use of the subroutine in Figure 15.31 to read a word at random.

FIGURE 15.32 Reading a word at random.



Modified HANGMAN Program

Suppose that you have created a data file called WORDS that contains a large number of words (say 100) using the program described earlier (see Figure 15.18). We saw in Figure 15.31 how to read a word at random from such a file.

To incorporate these ideas into the HANGMAN program, change the "find a word" subroutine given in Figure 15.14 to the subroutine shown in Figure 15.33. Each time this subroutine is called, a random word will be read from the disk file.

```
1000 REM FIND A WORD
1030 X=INT(RND(0)*NW+1)
1040 OPEN #2,4,0,F$
1050 FOR I=1 TO X:INPUT #2,W$:NEXT I
1055 CLOSE #2
1060 RETURN
```

FIGURE 15.33 Modified HANGMAN subroutine that will find a word from a collection of words stored on a cassette data tape.

Storing Numbers in a Sequential File

The subroutine shown in Figure 15.18 stored words in a sequential file on a diskette. It is also possible to store numerical data on a diskette. To investigate this, substitute the line

1100 GOSUB 1400

in the subroutine in Figure 15.18 and then add the subroutine shown in Figure 15.34. This subroutine will store the numbers 1–10 on the disk. The important thing to remember when storing numerical data is that each numerical value must be followed by a carriage return character. This is why line 1430 is written as

PRINT #2;I:PRINT #2;I+1

The form

PRINT #2;I,I+1

will not work because the comma is not recognized when writing to a disk file.

To read back the numerical data, change the subroutine in Figure 15.24 to that shown in Figure 15.35. The result of executing this subroutine is shown in Figure 15.36. Note that the input statement

INPUT #2,X,Y

will read two data fields and store the values in X and Y.

```
1400 REM SAVE NUMERICAL DATA
1410 FOR I=1 TO 10 STEP 2
1420 ? #2;I;? #2;I+1
1430 NEXT I
1440 RETURN
```

FIGURE 15.34 Subroutine to store numerical data on a diskette.

FIGURE 15.35 Subroutine to read numerical data from a disk file.

```
3100 REM READ NUMERICAL DATA
3105 ? "▯"
3110 GOSUB 4000:REM READ SETUP
3120 OPEN #2,4,0,F$
3130 FOR I=1 TO 5
3140 INPUT #2,X,Y
3150 ? X,Y
3160 NEXT I
3170 CLOSE #2
3175 ? "PRESS ANY KEY TO CONTINUE"
3180 GET #1,A
3190 RETURN
```

FIGURE 15.36 Result of executing the subroutine in Figure 15.35.

```
INSERT DISKETTE CONTAINING
WORDS TO BE READ

PRESS ANY KEY TO CONTINUE

WHAT FILE NAME ?NUMBERS
1           2
3           4
5           6
7           8
9           10
PRESS ANY KEY TO CONTINUE
```

ATARI ORGAN

As another example of developing a BASIC program we will turn the ATARI into a musical instrument. First we will learn how to play the notes of the scale by pressing keys on the keyboard; we will then develop a complete program that will display the musical keys on the screen.

Playing a Tone When a Key Is Pressed

You should review the sections in Chapters 4 and 5 on making sounds with the ATARI. Recall that the statement

SOUND V,P,D,L

will produce a single tone for voice V on the TV speaker of pitch P, distortion D, and loudness L.

Type in the following short program and run it:

```
10 P=121
20 GET #1,A
30 A$=CHR$(A)
40 IF A$= " " THEN SOUND 0,0,0,0:GOTO 20
50 SOUND 0,P,10,8
60 GOTO 20
```

This program should play a note each time any key is pressed. The note can be turned off by pressing the space bar. We must now make different keys play different notes.

Screen Layout

The program we will write will display 10 white keys and 7 black keys in high-resolution graphics according to the layout shown in Figure 15.37. The keys A through ; on the computer keyboard will be the "white" keys of the organ; the keys in the row above will be used for sharps and flats (black keys). The ATARI keys corresponding to each key on the screen will be printed on each key by POKEing the bit codes of the internal character set. The words ATARI ORGAN will be plotted using PLOT and DRAWTO statements.

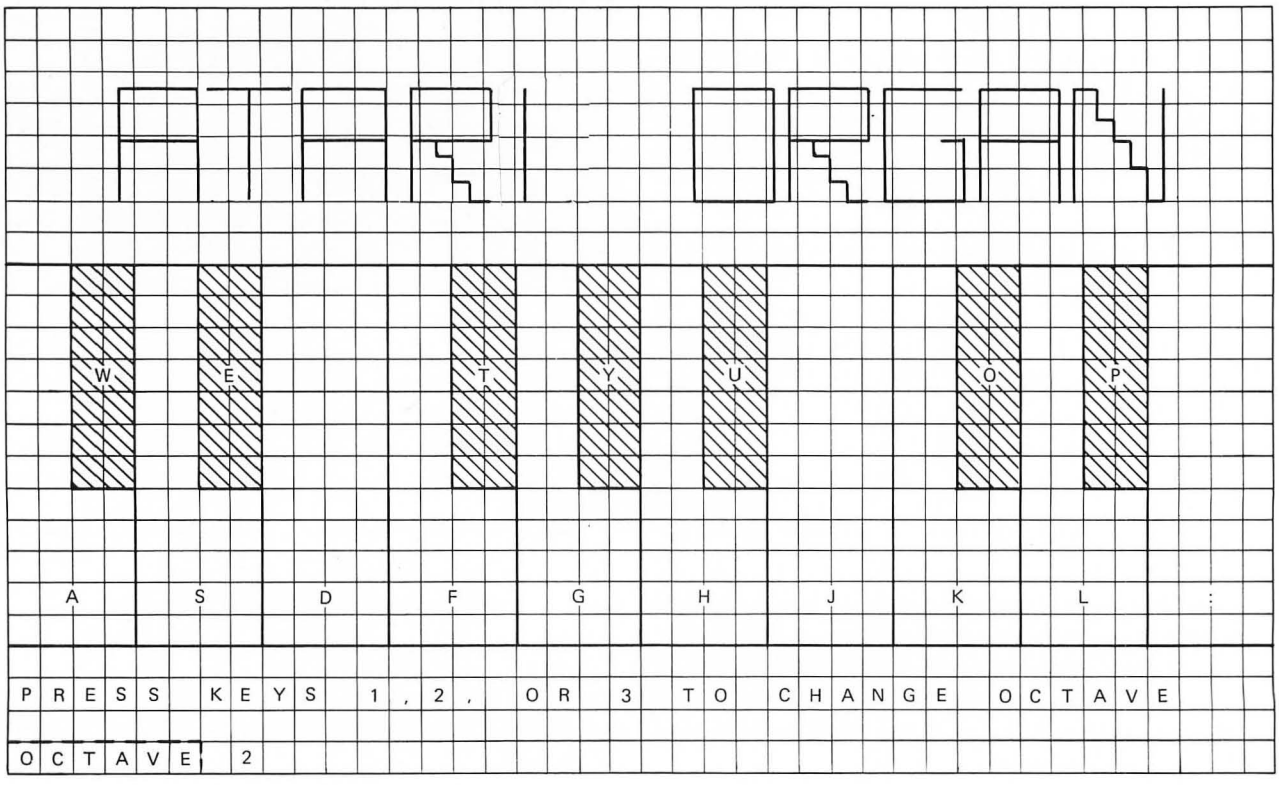
When a note is played, an eighth note will be displayed on the key being pressed for as long as the tone continues. Thus, as a song is played this note will move from key to key. This note will be plotted using PLOT and DRAWTO statements.

The ATARI organ will start out in octave 2. Pressing key 1 will change it to octave 1 (one octave lower), and pressing key 3 will change it to octave 3 (one octave higher). Thus, the total range of the ATARI organ will be over three octaves. For example, pressing key A on the computer keyboard in the octave 2 mode will produce the same note as pressing key K in the octave 1 mode. The octave number that is active at any time is displayed near the bottom of the screen.

Pitch Values for the Musical Scale

The pitch values for each note in the three octaves corresponding to our screen layout are given in Figure 15.38. The pitch values shown in Figure 15.38 will be stored in a two-dimensional array P(I,J), where I (1–17) corresponds to the white and black keys being displayed on the screen and J (0–2) corresponds to the octave number minus 1.

FIGURE 15.37 Screen layout for the ATARI organ.



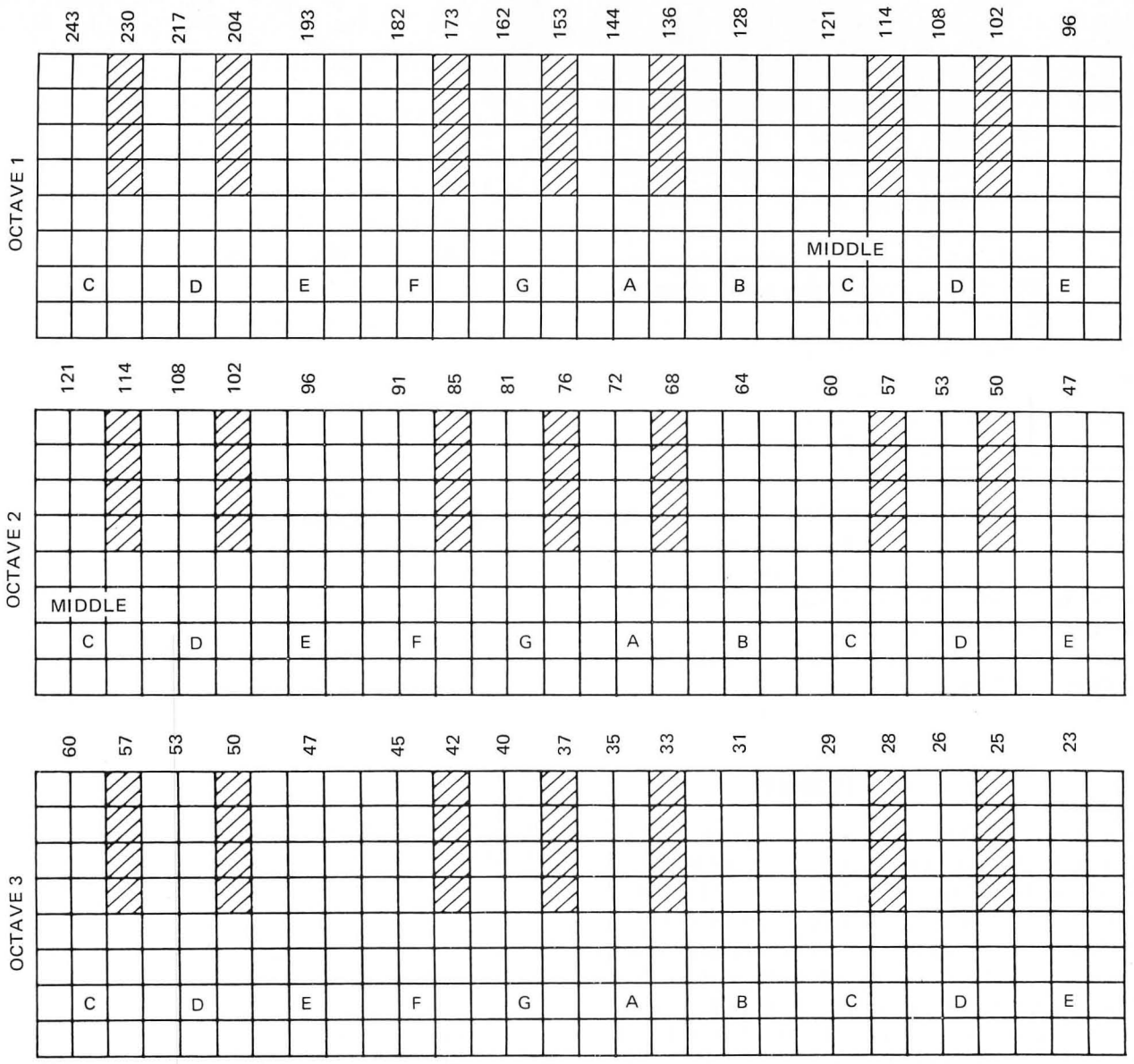


FIGURE 15.38 Pitch values corresponding to all notes that can be played on the ATARI organ.

Word Description of Program

The program for the ATARI organ can be understood from the pseudocode word-description given in Figure 15.39.

Variable Definitions

The major variables used in the ATARI organ progam are defined in Figure 15.40. Using these variables the pseudocode program shown in Figure 15.39 can be refined to the program shown in Figure 15.41. 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 value stored in P(I,J). If key 1,2, or 3 is pressed, no match will occur

FIGURE 15.39 Pseudocode word-description for ATARI organ program.

```
Initialize variables
Display keyboard
loop: Wait for key to be pressed
      if key is space bar
      then turn off sound
      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
```

FIGURE 15.40 Major variables used in ATARI organ program.

K(18)	An array containing the ATARI computer key ATASCII codes: 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(17,2)	A two-dimensional array containing the pitch values for the corresponding keys in K(I); the columns of the array correspond to the three octaves.
T	Octave number minus 1
P	Pitch value being played
A$	Character value corresponding to a computer key pressed

in the *do until* loop and I will equal 18. This is why K(I) must be dimensioned to K(18). In this case I is changed to VAL(A$), which will be 1, 2, or 3 (for valid keys), and T is changed to I − 1 to select the proper pitch values in P(I,T).

We are now at the point where the BASIC program will practically write itself.

The Main Program

The main program for the ATARI organ is shown in Figure 15.42. It follows closely the pseudocode description shown in Figure 15.41. Lines 20–50 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(I,J) is the same as that of the corresponding key values in K(I), according to Figure 15.38. Lines 35 and 50 store tab data that will be used later (in subroutine 1400) to plot the eighth note on the black keys.

Lines 70–170 are a direct implementation of the *loop . . . repeat forever* loop in Figure 15.41. Subroutine 1400, which is called in line 130, will play the note. Evenutally this subroutine will display the eighth note on the screen keyboard while playing the note. However, for now you can test out the playing of the ATARI organ by typing

```
1400 REM PLAY NOTE
1450 SOUND 0,P,10,8
1470 RETURN
```

which just plays the note. You will also need to write the following stub for the keyboard display subroutine:

```
600 REM DISPLAY KEYBOARD
610 RETURN
```

FIGURE 15.41 Pseudocode description of ATARI organ program.

```
Fill arrays K(I) and P(I,J) with appropriate values
Display keyboard
T = 1
loop: Wait for key A$ (code A) to be pressed
      if A$ = " " then turn off sound: go to loop
      I = 1
      do until A = K(I) or I = 18
          I = I + 1
      enddo
      if I < 18
      then P = P(I,T)
           play note
      else I = VAL(A$)
           if I = 0 or I > 3
             then do nothing (invalid key)
             else
                  Display I
                  T = I − 1
repeat forever
```

Try running the program now. The notes should change when you press different keys. You can stop any note by pressing the space bar.

Remaining Subroutines

Once you have the musical part of the ATARI organ working you can finish the "display keyboard" subroutine, as shown in Figure 15.43. This subroutine will produce the keyboard shown in Figure 15.44; the title and lettering on the keyboard are printed using subroutine 800, shown in Figure 15.45.

The subroutine at line 800 POKEs the bit codes stored in the internal character set directly on the screen. The DATA statement in line 805 contains the offsets into the character table for the letters on the keys.

The subroutine at line 900 that is called in line 895 plots the title at the top of the screen. The individual letters are plotted using the subroutines at lines 1010–1070.

The eighth-note shape is displayed when a note is played in subroutine 1400, shown in Figure 15.46. The value of I in line 1410 is the value found in the *do until* loop in lines 90–100 of the main program. If this value is greater than 10, a "black" key was pressed and the coordinate X at which the note will be plotted is determined by the statement in line 1420. The X

FIGURE 15.42 Main program for the ATARI organ.

```
10 REM ATARI ORGAN
12 OPEN #1,4,0,"K:"
15 DIM K(18),P(17,2),TB(7),A$(1)
20 ? "}":? "STORING DATA -- BE PATIENT!!"
25 DATA 65,83,68,70,71,72,74,75,76,59,87,69,84,89,85,79,80
26 DATA 243,217,193,182,162,144,128,121,108,96,230,204,173,153,136,114,102
28 DATA 121,108,96,91,81,72,64,60,53,47,114,102,85,76,68,57,50
30 DATA 60,53,47,45,40,35,31,29,26,23,57,50,42,37,33,28,25
35 DATA 28,60,124,156,188,252,284
40 FOR I=1 TO 17:READ K:K(I)=K:NEXT I
45 FOR J=0 TO 2:FOR I=1 TO 17:READ P:P(I,J)=P:NEXT I:NEXT J
50 FOR I=1 TO 7:READ TB:TB(I)=TB:NEXT I
55 GOSUB 600:REM DISPLAY KEYBOARD
60 T=1
70 GET #1,A
75 A$=CHR$(A)
80 IF A$="X" THEN GRAPHICS 0:CLOSE #1:END
82 IF A$=" " THEN SOUND 0,0,0,0:GOSUB 1500:GOTO 70
85 I=1
90 IF A=K(I) OR I=18 THEN 110
100 I=I+1:GOTO 90
110 IF I=18 THEN 140
120 P=P(I,T)
130 GOSUB 1400:GOTO 70
140 IF A$<"1" OR A$>"3" THEN 70
150 I=VAL(A$)
160 POSITION 8,22:? "";A$;
170 T=I-1:GOTO 70
```

FIGURE 15.43 Subroutine to display the keyboard for the color organ.

```
600 REM DISPLAY KEYBOARD
610 ? "}":GRAPHICS 8:SETCOLOR 2,0,0:SETCOLOR 1,0,12:COLOR 1
615 FOR Y=63 TO 159
620 PLOT 0,Y:DRAWTO 319,Y:NEXT Y
625 COLOR 0
630 FOR X=32 TO 288 STEP 32
635 PLOT X,64:DRAWTO X,159:NEXT X
640 Y=64
650 FOR X=16 TO 48 STEP 32
655 GOSUB 700:NEXT X
660 FOR X=112 TO 176 STEP 32
665 GOSUB 700:NEXT X
670 FOR X=240 TO 272 STEP 32
675 GOSUB 700:NEXT X
680 ? :? "PRESS KEYS 1,2, OR 3 TO CHANGE OCTAVE"
690 ? :? "OCTAVE";:? " 2";
695 GOSUB 800:RETURN
700 REM PLOT BLACK KEY AT X,Y
710 FOR I=0 TO 14
720 PLOT X+I,Y:DRAWTO X+I,Y+48
730 NEXT I:RETURN
```

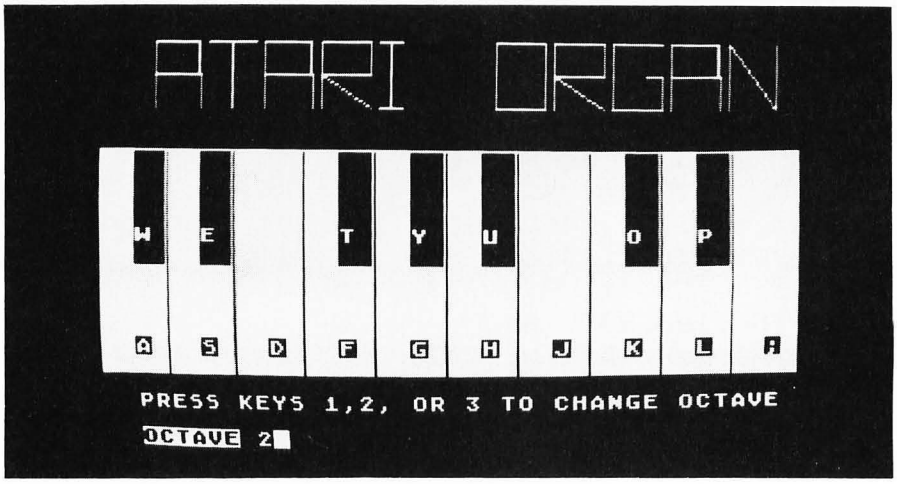


FIGURE 15.44 The keyboard for the organ produced by the subroutine shown in Figure 15.43.

FIGURE 15.45 Subroutine to display the lettering on the organ keyboard and the title.

```
800 REM KEY LABELS & TITLE
805 DATA 33,51,36,38,39,40,42,43,44,27,55,37,52,57,53,47,48
810 SA=PEEK(89)*256+PEEK(88)
815 FOR X=2 TO 38 STEP 4
820 READ C
825 LA=SA+40*144+X
830 CHAD=57344+C*8
835 FOR I=0 TO 7
840 POKE LA+I*40,PEEK(CHAD+I)
845 NEXT I:NEXT X
850 FOR X=2 TO 34 STEP 4
855 IF X=10 OR X=26 THEN 890
860 READ C
865 LA=SA+40*96+X
870 CHAD=57344+C*8
875 FOR I=0 TO 7
880 POKE LA+I*40,PEEK(CHAD+I)
885 NEXT I
890 NEXT X
895 GOSUB 900:REM PLOT TITLE
899 RETURN
900 REM TITLE
905 COLOR 1
910 Y=45:X=26:GOSUB 1010:REM A
915 X=X+26:GOSUB 1020:REM T
920 X=X+26:GOSUB 1010:REM A
925 X=X+26:GOSUB 1030:REM R
930 X=X+26:GOSUB 1040:REM I
935 X=X+52:GOSUB 1050:REM O
940 X=X+26:GOSUB 1030:REM R
945 X=X+26:GOSUB 1060:REM G
950 X=X+26:GOSUB 1010:REM A
955 X=X+26:GOSUB 1070:REM N
960 RETURN
1000 REM LETTERS
1010 PLOT X,Y:DRAWTO X,Y-28:DRAWTO X+21,Y-28:DRAWTO X+22,Y:PLOT X,Y-14:DRAWTO X+
22,Y-14:RETURN :REM A
1020 PLOT X,Y-28:DRAWTO X+22,Y-28:PLOT X+11,Y-28:DRAWTO X+11,Y:RETURN :REM T
1030 PLOT X,Y:DRAWTO X,Y-28:DRAWTO X+22,Y-28:DRAWTO X+22,Y-14:DRAWTO X,Y-14:DRAW
TO X+22,Y:RETURN :REM R
1040 PLOT X,Y:DRAWTO X+10,Y:PLOT X,Y-28:DRAWTO X+10,Y-28:PLOT X+5,Y-28:DRAWTO X+
5,Y:RETURN :REM I
1050 PLOT X,Y:DRAWTO X,Y-28:DRAWTO X+22,Y-28:DRAWTO X+22,Y:DRAWTO X,Y:RETURN :RE
M O
1060 PLOT X+22,Y-28:DRAWTO X,Y-28:DRAWTO X,Y:DRAWTO X+22,Y:DRAWTO X+22,Y-14:DRAW
TO X+15,Y-14:RETURN :REM G
1070 PLOT X,Y:DRAWTO X,Y-28:DRAWTO X+22,Y:DRAWTO X+22,Y-28:RETURN :REM N
```

```
1400 REM PLAY NOTE
1405 GOSUB 1600
1410 IF I<11 THEN Y=125:X=32*I-4:COLOR 0:GOSUB 1500:COLOR 1:GOTO 1450
1420 Y=75:X=TB(I-10)
1430 COLOR 1:GOSUB 1500:COLOR 0
1450 SOUND 0,P,10,8
1470 RETURN
1500 REM PLOT NOTE
1510 PLOT X,Y:DRAWTO X-3,Y-3
1520 DRAWTO X-3,Y+12:DRAWTO X-5,Y+10
1530 DRAWTO X-7,Y+10:DRAWTO X-9,Y+12
1540 DRAWTO X-9,Y+14:DRAWTO X-7,Y+16
1550 DRAWTO X-5,Y+16:DRAWTO X-3,Y+14
1560 DRAWTO X-3,Y+12:RETURN
1600 REM ERASE EIGTH NOTE
1610 IF Y=125 THEN COLOR 1:GOSUB 1500:RETURN
1620 IF Y=75 THEN COLOR 0:GOSUB 1500:RETURN
1630 RETURN
```

FIGURE 15.46 Subroutine to display the eighth note and produce the tone.

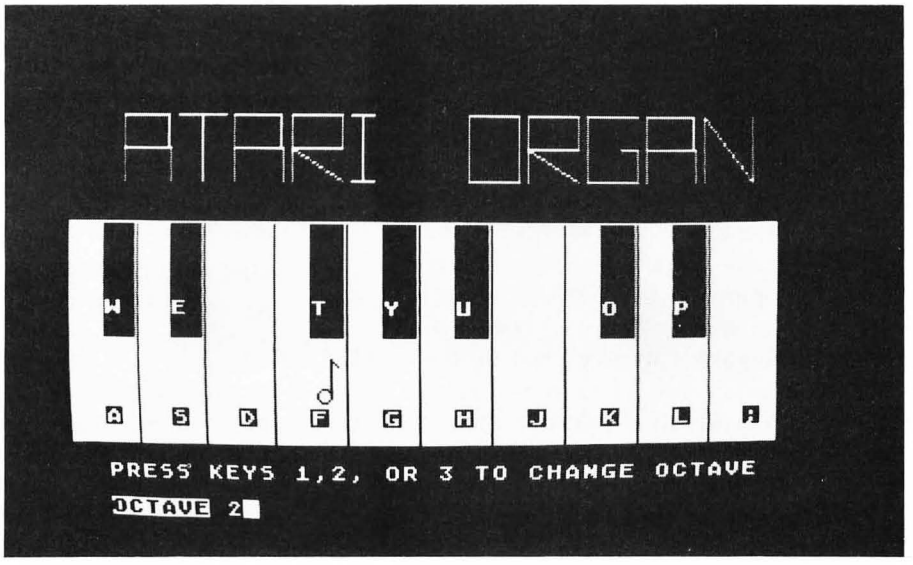
and Y values defined in line 1420 determine where on the screen the eighth note is plotted. The subroutine at line 1500 plots the eighth note. The X position is determined by the value in the tab array TB(I–10) that was initialized in lines 35 and 50 in Figure 15.42.

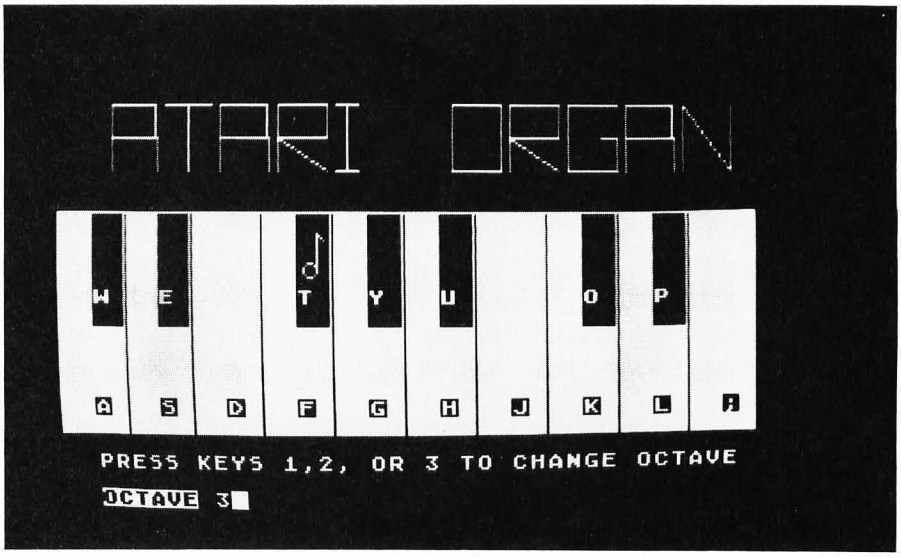
The coordinate X for plotting the note on the white keys is given in the THEN clause in line 1410. Since the spacing of the white keys is uniform, the position of the note on the line can be calculated by the equation X = 32 * I − 4, as given in line 1410.

On the black keys, the eighth note is plotted white by calling subroutine 1500 in line 1430 after executing the statement COLOR 1. On the white keys the eighth note is plotted black by calling subroutine 1500 after the statement COLOR 0 in line 1410. When the space bar is pressed, the sound is turned off and the eighth note will be erased by calling subroutine 1500 in line 82 (see Figure 15.42). The subroutine at line 1600 that is called at line 1405 will erase any existing eighth note when a new note is played. Figure 15.47 shows examples of the eighth note that is displayed when the ATARI organ is played.

FIGURE 15.47 (a) ATARI organ when a white note is played (key F); (b) ATARI organ when a black note is played (key T).

(a)





(b)

This ATARI organ program has only begun to use the sound capabilities in the ATARI. By playing more than one voice you can create chords. You can also vary the loudness of each note and create special sound effects by changing the distortion value in the SOUND statement. See Exercise 15.2 for some ideas on how to do this.

CONCLUSION

The HANGMAN and the ATARI 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 programs that you write. However, they are a good guide to use when you get stuck and don't know how to proceed. In the last analysis you will have to develop your own approach to writing computer programs. Programming is a skill that still requires insight, creativity, a knack for problem solving, and *practice*.

If you have read this entire book, typed all the examples on your ATARI, and worked a number of exercises, then you will have a good understanding of how to write BASIC programs on an ATARI computer. It is now time for you to start writing your own programs. Many useful programs can be written for the ATARI. Pick an area in which you are an expert. How can the ATARI help you in this area? Start by writing a short program, and then expand it into a longer, more complex one. You will find that writing computer programs is challenging, rewarding, and fun. Good luck!

EXERCISE 15.1
Modify the subroutine to find a word in Figure 15.14 so that no word is selected more than once.

EXERCISE 15.2
Modify the ATARI organ program to

1. play chords after key C has been pressed (A major chord can be played by multiplying the pitch of voice 0 by 0.79166 for voice 1 and by 0.66666 for voice 2)
2. vary the loudness of the notes by using the up and down keys
3. change the distortion value using keys 4–9.

EXERCISE 15.3
Write a program to play the game MASTERMIND. 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 and that are in the correct position in the number, and W is the number of digits guessed that are in the number but that were guessed in the wrong position. The player continues to guess numbers until the correct number is 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.

EXERCISE 15.4

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 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 10 and an ace can count either 1 or 11. Being dealt an ace and a face card is a *blackjack* and is an automatic winner. 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. No money is won or lost on a tie. Have the program continue playing and keep a running total of the player's winnings.

EXERCISE 15.5.

Write a program to play tic-tac-toe (see Exercise 7.5). The player should have the option to play against a second player or the computer.

APPENDICES

APPENDIX A

Reserved Words

None of the following *reserved words* should be used as part of a variable name in an ATARI BASIC program.

ABS
ADR
AND
ASC
ATN
BYE
CLOAD
CHR$
CLOG
CLOSE
CLR
COLOR
COM
CONT
COS
CSAVE
DATA
DEG
DIM
DOS
DRAWTO
END
ENTER
EXP
FOR
FRE
GET
GOSUB
GOTO
GRAPHICS
IF
INPUT
INT
LEN
LET
LIST
LOAD
LOCATE
LOG
LPRINT
NEW
NEXT
NOT
NOTE
ON
OPEN
OR
PADDLE
PEEK
PLOT
POINT
POKE
POP
POSITION
PRINT
PTRIG
PUT
RAD
READ
REM
RESTORE
RETURN
RND
RUN
SAVE
SETCOLOR
SGN
SIN
SOUND
SQR
STATUS
STEP
STICK
STRIG
STOP
STR$
THEN
TO
TRAP
USR
VAL
XIO

APPENDIX B

ATASCII Codes

Character C$ or Keystroke	*Keystroke*					
	Normal Video			*ATARI Key (Reverse Video)*		
	CAPS	*LOWR*	*CTRL*	*CAPS*	*LOWR*	*CTRL*
	ASC(C$)					
Blank (space)	32			160		
!	33			161		
"	34			162		
#	35			163		
$	36			164		
%	37			165		
&	38			166		
'	39			167		
(	40			168		
)	41			169		
*	42			170		
+	43			171		
,	44		0	172		128
-	45			173		
.	46		96	174		224
/	47			175		
0	48			176		
1	49			177		
2	50			178		
3	51			179		
4	52			180		
5	53			181		
6	54			182		
7	55			183		
8	56			184		
9	57			185		
:	58			186		
;	59		123	187		251
<	60			188		
=	61			189		
>	62			190		
?	63			191		
@	64			192		
A	65	97	1	193	225	129
B	66	98	2	194	226	130
C	67	99	3	195	227	131
D	68	100	4	196	228	132
E	69	101	5	197	229	133
F	70	102	6	198	230	134
G	71	103	7	199	231	135
H	72	104	8	200	232	136
I	73	105	9	201	233	137
J	74	106	10	202	234	138
K	75	107	11	203	235	139
L	76	108	12	204	236	140
M	77	109	13	205	237	141
N	78	110	14	206	238	142
O	79	111	15	207	239	143
P	80	112	16	208	240	144
Q	81	113	17	209	241	145
R	82	114	18	210	242	146
S	83	115	19	211	243	147
T	84	116	20	212	244	148
U	85	117	21	213	245	149
V	86	118	22	214	246	150
W	87	119	23	215	247	151
X	88	120	24	216	248	152
Y	89	121	25	217	249	153
Z	90	122	26	218	250	154
[	91			219		
\	92			220		
]	93			221		
^	94			222		
_	95			223		
|	124					252
ESC	27					
RETURN	155					

Special Control Characters

The following characters are only displayed if they are preceded by the ESC key, CHR$(27). Otherwise, the result takes place.

ATASCII Code	*Keystroke*	*Result*
28	ESC/CTRL −	cursor up
29	ESC/CTRL =	cursor down
30	ESC/CTRL +	cursor left
31	ESC/CTRL *	cursor right
125	ESC/CLEAR	clear screen
126	ESC/BACK S	backspace
127	ESC/TAB	move cursor to next tab
156	ESC/SHIFT BACK S	delete line
157	ESC/SHIFT >	insert line
158	ESC/CTRL TAB	clear tab stop
159	ESC/SHIFT TAB	set tab stop
253	ESC/CTRL 2	beep speaker
254	ESC/CTRL BACK S	delete character
255	ESC/CTRL >	insert character

APPENDIX C

ERROR CODES

Error code is stored in memory location 195.
EC = PEEK(195)

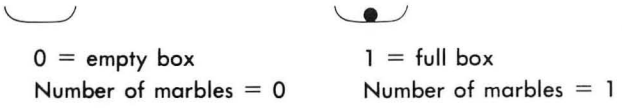
EC	*Error*
2	Memory insufficient
3	Value error
4	Too many variables
5	String length error
6	Out-of-data error
7	Number greater than 32767
8	INPUT statement error
9	Array or string DIM error
10	Argument stack overflow
11	Floating point overflow/underflow error
12	Line not found
13	NEXT without FOR
14	Line too long
15	GOSUB or FOR line deleted
16	RETURN without GOSUB
17	Garbage error
18	Invalid string character
19	LOAD program too long
20	Bad device number
21	LOAD file error
128	BREAK abort
129	IOCB already open
130	Unknown device
131	Opened for write only
132	Invalid command
133	Device or file not open
134	Bad channel number
135	Opened for read only
136	End of file
137	Record truncated
138	Device timeout
139	Device cannot perform a command
140	Serial bus input framing error
141	Cursor out of range
142	Serial bus data frame overrun
143	Serial bus data frame checksum error
144	Disk write-protected
145	Read-after-write compare error or bad screen mode
146	Function not implemented
147	Insufficient RAM for graphics
150	Port already open
151	Concurrent mode I/O not enabled
152	Illegal user-supplied buffer
153	Active concurrent mode I/O error
154	Concurrent mode inactive
160	Drive number error
161	Too many open files
162	Disk full
163	Fatal I/O error
164	File number mismatch
165	File name error
166	POINT data length error
167	File locked
168	Command invalid
169	Directory full (64 files)
170	File not found
171	POINT invalid

APPENDIX D

HEXADECIMAL NUMBERS

Consider a box containing one marble. If 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;* with 1 bit we can count from 0 (box empty) to 1 (box full), as shown in Figure D.1.

FIGURE D.1 You can count from 0 to 1 with 1 bit.

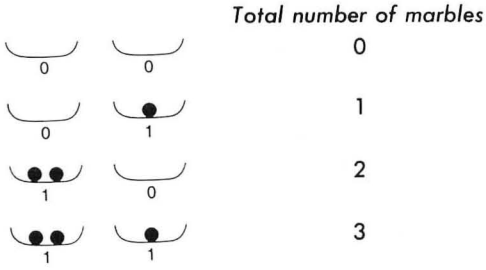


Consider now a second box that can also only be full (1) or empty (0). However, when this box is full it will contain *two* marbles, as shown in Figure D.2.

FIGURE D.2 This box can either contain two marbles (full) or no marbles.

With these two boxes (2 bits) we can now count from 0 to 3, as shown in Figure D.3. Note that the value of each 2-bit binary number shown in Figure D.3 is equal to the total number of marbles in the two boxes.

FIGURE D.3 You can count from 0 to 3 with 2 bits.

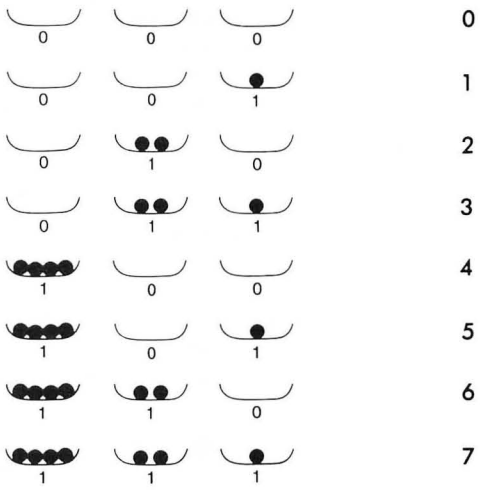


We can add a third bit to the binary number by adding a third box that is full (bit = 1) when it contains four marbles and is empty (bit = 0) when it contains no marbles. It must be either full (bit = 1) or empty (bit = 0). With this third box (3 bits) we can count from 0 to 7, as shown in Figure D.4.

If you want to count beyond 7, you must add another box. How many marbles should this fourth box contain when it is full (bit = 1)? It should be clear that this box must contain eight marbles. The binary number 8 would then be written as

1000

FIGURE D.4 You can count from 0 to 7 with 3 bits.



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 4 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 4 bits.

No. of marbles in each full box (bit=1)				*Total no.*	
8	*4*	*2*	*1*	*of marbles*	*Hex Digit*
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 the four boxes represented by the 4-bit binary numbers shown in Figure D.5 by a single digit.

We call this a *hexadecimal* digit. The 16 hexadecimal digits are shown in the right-hand column in Figure D.5. The hexadecimal digits 0–9 are the same as the decimal digits 0–9. However, the decimal numbers 10–15 are represented by the hexadecimal digits A–F. 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 8 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 of the marbles in the full boxes (the ones with binary digits = 1).

FIGURE D.6 You can count from 0 to 255 with 8 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

As the length of a binary number increases it becomes more cumbersome to work with. 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 4 bits starting at the right, and then represent each 4-bit group by its corresponding hexadecimal digit, given in Figure D.5. For example, the binary number

$$\underbrace{1001}_{9}\ \underbrace{1010}_{A}$$

is equivalent to the hexadecimal number 9A. You should verify that the total number of marbles represented by this binary number is 154. However, instead of counting the marbles in the "binary boxes," you can count the marbles in "hexadecimal" boxes, where 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

$$\underbrace{1000}_{8}\ \underbrace{0111}_{7}\ \underbrace{1100}_{C}\ \underbrace{1001}_{9}$$

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:

$$\begin{aligned} 8 \times 16^3 &= 8 \times 4{,}096 = 32{,}768 \\ 7 \times 16^2 &= 7 \times 256 = 1{,}792 \\ C \times 16^1 &= 12 \times 16 = 192 \\ 9 \times 16^0 &= 9 \times 1 = 9 \\ & \qquad\qquad\qquad\quad 34{,}761 \end{aligned}$$

You can see that by working with hexadecimal numbers you can reduce by a factor of 4 the number of digits that you have to work with.

Table D.1 will allow you to conveniently convert up to 4-digit hexadecimal numbers to their decimal equivalents. Note, for example, how the four terms in the conversion of 87C9 given here can be read directly from the table.

FIGURE 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	4,096	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

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, you can call this subroutine from BASIC and pass data values to and from the subroutine by using the USR function.

When the BASIC statement

W=USR(AD,X,Y)

is executed, the return address in the BASIC program is pushed on the stack and the program jumps to the machine language program starting at the decimal address AD. The arguments X,Y in this statement are optional and represent data values to be passed to the machine language program. These values are pushed on the stack as 2-byte, 16-bit integer values. The top of the stack contains a 1-byte integer equal to the number of arguments in the USR function. This value will be 0 if no arguments are passed to the subroutine. The machine language program must pull this count value plus all arguments off the stack before returning to the BASIC program by executing an RTS instruction. After the USR function given here is executed the stack will look like Figure E.1.

FIGURE E.1 Stack after executing W=USR (AD,X,Y).

Sp →	02	number of arguments
	X low	1st USR argument
	X high	
	Y low	2nd USR argument
	Y high	
	Return address low	
	Return address high	

To pass a 16-bit integer value V back to the BASIC program, store the most significant byte in memory location 213 ($D5) and the least significant byte in memory location 212 ($D4). Then execute an RTS instruction which will return control to the BASIC program with the value of USR equal to the 16-bit integer value V.

APPENDIX F

Formatting Your Diskette

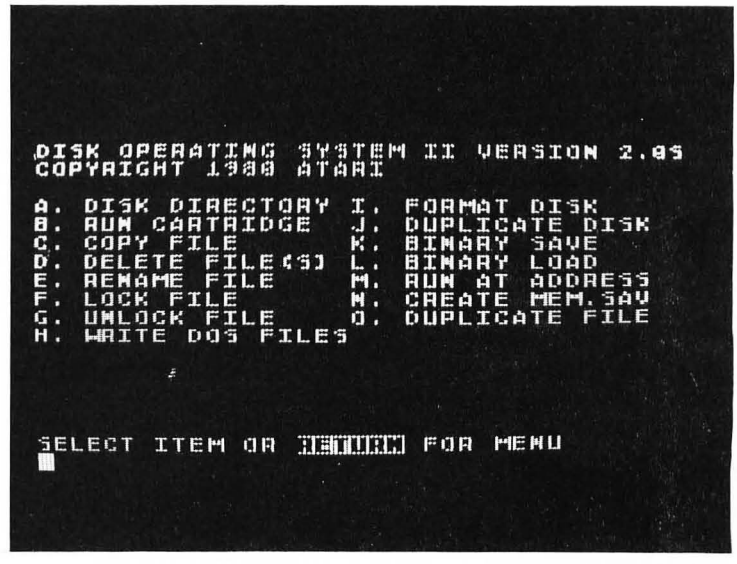
To format a new diskette you must use the *disk utility package* (DOS menu). This package is part of the DOS system files. These DOS system files are loaded into memory (*booted*) when the computer is turned on. If you type

DOS

the DOS menu shown in Figure F.1 will be displayed on the screen.

To format your diskette, insert it in the disk drive. Remember that formatting a diskette will *destroy* all information on the diskette. Then press key I. The message WHICH DRIVE TO FORMAT? will appear on the screen. Press 1. The message TYPE "Y" TO FORMAT DRIVE 1 will appear on the screen. If you now press key Y the disk drive will come on and format your disk in about one minute.

FIGURE F.1 DOS menu.



You should then write the DOS files to your diskette so that you can boot the system with your disk. To do this, press key H in response to the menu in Figure F.1. Follow the prompting messages on the screen to complete the process.

APPENDIX G

Summary of BASIC Statements

The following summary gives examples of various statements used in ATARI BASIC. For a more detailed discussion of each statement, refer to the pages cited.

Data Transfer Statements	*Page Ref.*
PRINT A$; B, C	22
INPUT C	31
GET #1,A	105
READ A, B, C$	84
DATA 5, 10, JOE	84
RESTORE	84
PRINT #2;W$	151
INPUT #2,W$	152
OPEN #2,8,O,F$	151
CLOSE #2	151
POKE 1536,75	128
X=PEEK(1024)	128

Branching and Looping Statements	*Page Ref.*
GOTO 40	15
IF M1 > M2 THEN PRINT "TOO SMALL": GOTO 20	43
FOR I=1 TO 10: PRINT I: NEXT I	36
GOSUB 500	75
RETURN	75
ON I GOSUB 100,200,300	150
ON NH GOTO 960,965,970,975,980,985	147
TRAP 9000	153

String Related Statements	*Page Ref.*
B$ = A$(1,I)	103
B$ = A$(LEN(A$)-I+1)	103
B$ = A$(I,I+J-1)	103
N = LEN(A$)	102
N = VAL(A$)	103
A$ = STR$(A)	104
N = ASC(A$)	105
A$ = CHR$(A)	105

BASIC Functions		*Page Ref.*
Z = SQR(X)	square root	27
Z = ABS(X)	absolute value	26
Z = INT(X)	integer value	26
Z = SGN(X)	sign	26
X = RND(O)	random number	27
Z = SIN(X)	sine	28
Z = COS(X)	cosine	28
Z = ATN(X)	arctangent	28
Z = LOG(X)	natural logarithm	28
Z = CLOG(X)	base 10 logarithm	28
Z = EXP(X)	exponential function	28
Z = PADDLE(O)	paddle function	81
Z = PTRIG(O)	paddle trigger function	82
Z = STICK(3)	joystick function	83
Z = STRIG(3)	joystick trigger function	83

Graphics Statements	*Page Ref.*
GR.5	55
SETCOLOR 0,0,14	56
COLOR 1	57
PLOT X,Y	57
DRAWTO X,Y	58
GR.0	57
GR.8	117
GR.1	131
GR.2	131

Other Statements and Commands	*Page Ref.*
DIM A(20)	95
? FRE(0)	95
NEW	14
SAVE	14
LOAD	14
RUN	7
CONT	15
STOP	16
END	16
LIST	7
REM REMARK	17
CLR	95
CLOAD	14
CSAVE	14
DEG	28
RAD	28
DOS	169
SOUND 0,P,10,8	35
X = USR(AD,Y)	169

SUBJECT INDEX

D

E

F

P

Q

R

S

T

U

V

W

Z

PERSONAL COMPUTERS

RICHARD HASKELL

atari basic

This practical, easy-to-use guidebook provides a solid introduction to programming an ATARI computer in BASIC.

Complete with examples illustrated by actual photographs taken from the computer's video screen, ATARI BASIC offers a hands-on, step-by-step approach to top-down programming that will enable you to master fundamental concepts and program a computer with ease and expertise in practically no time at all.

Written for the beginning and advanced programmer alike—for self-study or classroom instruction—ATARI BASIC tells you everything you need to know to make the most of your ATARI computer, including valuable information on:

- low-resolution graphics
- high-resolution graphics
- operation of the cassette tape recorder and floppy disk drive
- loops and arrays
- string variables and string functions
- how to include sound effects
- animated graphics
- and much more.

Richard Haskell holds a Ph.D. from Rensselaer Polytechnic Institute and is an engineering professor at Oakland University in Michigan. In addition, he has designed numerous microprocessor-based systems for industrial applications and written three other books in the Prentice-Hall computer series entitled PET/CBM BASIC, TRS-80 EXTENDED COLOR BASIC, and APPLE BASIC.

PRENTICE-HALL, Inc., Englewood Cliffs, New Jersey 07632

ISBN 0-13-049791-6