All About the Commodore 64

V·O·L·U·M·E O·N·E

Everything you need to know about BASIC to effectively program your Commodore 64®.

Craig Chamberlain

A **COMPUTE! Books** Publication $12.95

All About the Commodore 64

V·O·L·U·M·E O·N·E

Craig Chamberlain

COMPUTE!™ Publications, Inc. abc
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-40-X

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies and is not associated with any manufacturer of personal computers. Commodore 64 is a trademark of Commodore Electronics Limited.

Contents

Foreword

What can my computer do, and how do I make it do what I want? These two questions are asked by almost every beginning programmer. *All About the Commodore 64: Volume I* gives you the answers. This book emphasizes the powerful sound and graphics capabilities of the Commodore 64—so you can see and hear the results of your efforts. You'll quickly learn how to program your computer using BASIC, a computer language composed of simple, short statements similar to English words.

All About the Commodore 64: Volume I is your guide through 64 BASIC. Using simple examples, this book demonstrates how each command controls some special feature of your 64.

Craig Chamberlain understands what it's like to be a novice programmer. Like most beginning programmers, he faced the computer with some anxiety. So he doesn't simply tell you the right way to use BASIC. He talks about common mistakes you might make, what the computer does when you make mistakes, and what you can do to correct them.

Working at your own pace, you'll soon progress from putting a single character on the screen to creating a colorful animated display, from playing one note to composing an entire song on your computer.

After you're thoroughly familiar with BASIC's techniques, you'll learn how to design and write your own programs. There are even examples that show you how to search out and correct even the most subtle errors so that you can debug your own programs.

You may consider yourself a beginner now, but with *All About the Commodore 64: Volume I*, you'll quickly become an experienced BASIC programmer.

Acknowledgments

The author wishes to thank Harry Bratt, Glen Carbone, Mark Davids, Chet Gonterman, Sheldon Leemon, and especially Kurt Vogel for their assistance in the development of this book.

Chapter 1

Introduction

Introduction

What Is BASIC?

Computers communicate using numbers. Humans don't. Dealing with numbers is a major problem confronting people who use computers.

We are accustomed to communicating using words. But words alone are not enough to convey information; the order in which they are put together is also important. Words, plus some rules about how to combine them, are what form a language.

Unfortunately, English is too complicated to be implemented on a microcomputer like the Commodore 64, but it is possible to make a computer understand a simple language. The trick to making a language simple is to use a small vocabulary, and to limit the ways in which the words can be combined.

BASIC is a computer language which has these characteristics. BASIC, which stands for Beginners All-purpose Symbolic Instruction Code, was created at Dartmouth College by John Kemeny and Thomas Kurtz about 20 years ago. The BASIC on your Commodore 64 is known as Microsoft BASIC Version Two.

BASIC is a sort of interpreter between your words and the computer's numbers. If you know what you want the computer to do, it is rather easy to write instructions in BASIC for the computer to follow. Using the rules of the language, the computer interprets your instructions and performs whatever is necessary to produce the desired results. Now the difficulty in using a computer has been reduced to writing BASIC instructions. That is what programming is all about, and it is the subject of this book.

Why You Should Learn BASIC

With all of the programs already available for the Commodore 64, why would you ever want to write any yourself? As it turns out, there are many good reasons for learning BASIC.

The main reason is that learning how to program helps you get more out of your investment. As you learn more about how a computer works, you will realize its many applications and see how it can help you. You will be able to modify existing programs to better suit your needs. And when you need a program that nobody else has written, you can write it yourself.

Computers are playing an increasingly larger role in our society. Knowing how they work will make it easier for you to adapt to our changing world. By learning about computers, you are furthering your education, and you will have an advantage over those people who do not keep up with progress.

Finally, computers can be fun. Your Commodore 64 is a great entertainment machine, with fast action, high-resolution color displays, and music. If you stop and think about it, though, every time you play a game, you are enjoying the product of someone else's imagination. A computer can perform math calculations, display colorful pictures, and produce sounds, but original thinking is necessary to transform these mechanical operations into an entertaining game.

The computer lacks an imagination. That imagination can be supplied by a professional game programmer, or it can be supplied by you. By learning to program, you can turn your thoughts and ideas into reality, in ways not possible before. So, the computer provides a means for you to express your own creativity.

About This Book

This book has been written to give you an understanding of programming in general, and BASIC in particular. It presents all the essential things you need to know about BASIC so you can start learning and programming right away. We assume that you have never used a computer before. We also realize that you might learn languages besides BASIC. Many of the ideas and methods discussed in this book can be applied to other languages.

There are three main parts to this book. The first part, Chapters 2 through 6, presents fundamental concepts of computing. You will work only in what is called the *immediate* mode of the computer, for immediate results. You don't even have to bother with the Datassette or disk drive. The purpose of these chapters is to build a strong foundation in preparation for programming. When you move on to the second part, programming will seem to be a natural next step.

Chapters 7 through 11 contain all the essential information about programming. It is possible to write many programs using just the material in these five chapters.

The third part, comprising the last five chapters, covers more advanced topics. Here you will find some shortcuts that make programming even easier.

Each chapter is divided into several sections, and almost every section concludes with a summary which explores some feature of the computer while showing an application of the new material introduced in the chapter.

There are plenty of examples in each chapter, many of which make use of the graphics and sound capabilities of the Commodore 64. To remove as many obstacles from the learning process as possible, none of the examples is of a mathematical nature.

This book approaches topics from a conceptual standpoint, with the reasoning that if you understand the purpose of a procedure, you will better understand how to use it.

Here are some suggestions on how you might read this book. You could choose to read the entire book, in order. This might be a good strategy if you have never dealt with computers before. Or, if you have some experience, you may want to read only the summaries, referring to the main text for clarification or when you come to something new. Either way, you will probably find that you will have a more solid understanding of the subject if you try the examples, not just look at them. This book has been written so you can even read it in bed and still get a good idea of what is happening, but nothing can replace actual hands-on experience. The computer won't laugh at you if you make a mistake, and there is no way you can permanently damage the computer by typing, so feel free to experiment.

There are numerous program listings in this book. Each is designed to illustrate a point. In order to make the typing in of these programs easier, a special extra program has been included in the appendix called "The Automatic Proofreader," by Charles Brannon. Be sure to read this article before you try to type in any of the programs. Using the Proofreader will make program entry much easier.

You may also find it helpful to talk about some of the topics with somebody who knows about computers. We have tried to anticipate the most likely questions, but you are bound to have some questions we just couldn't cover.

Sneak Preview

Just so you can see what a program looks like, here is a short but fun program for you to try right now. Turn on your Commodore 64, LOAD and RUN the Automatic Proofreader (if you have not read the article Automatic Proofreader, Appendix I, now is the time to do so), type the word NEW, and press the key marked RETURN. Now type each of the lines exactly as listed below, pressing RETURN after every line.

If you make a typing mistake, you can easily correct it before pressing RETURN. Just press the DEL key until it backspaces to your error, then type your correction. When you are done, type LIST and RETURN. This will display your program on the screen so you can make sure the lines were entered correctly. If a line is wrong, simply retype it. Plug a joystick into port two. Type the word RUN, press RETURN, and start drawing.

```
10 PRINT CHR$(147) : POKE 53280,0 : POKE 53281,0 :
   S=500 : C=1                                  :rem 181
20 DIM JS(15) : FOR K=5 TO 14 : READ JS(K) : NEXT
                                                :rem 126
30 POKE 55296+S,C : POKE 1024+S,81              :rem 205
40 S=S+JS(PEEK(56320) AND 15)                    :rem 15
50 IF S<0 THEN S=S+1000                         :rem 177
60 IF S>999 THEN S=S-1000                        :rem 49
70 IF (PEEK(56320) AND 16) = 0 THEN C=C+1 : IF C>1
   5 THEN C=1                                    :rem 66
80 GOTO 30                                        :rem 4
90 DATA 41,-39,1,0,39,-41,-1,0,40,-40           :rem 239
```

By moving the joystick, you can doodle in eight directions. If you move off one side of the screen, the drawing reappears on the opposite side. To change the color of the line, press the trigger. Hold the trigger down to create a multicolored effect.

To stop the program and set the computer back to normal, hold down the key marked RUN/STOP and press the RESTORE key. Type LIST and press RETURN to see the program again. You can change the plotting shape from a ball to an asterisk by changing line 30. Type this line and press RETURN.

```
30 POKE 55296+S,C : POKE 1024+S,42
```

All that we did was change the number 81 to a 42. Type LIST and press RETURN to check that line 30 was changed.

Start the program again by typing RUN and pressing RETURN. This time, asterisks will be drawn on the screen. You might also try the number 160, for a square block.

Here is a complete breakdown of what is accomplished by each line in the program.

10 PRINT CHR$(147)
    clears the screen
    POKE 53280,0 : POKE 53281,0
    sets border and background colors to black
    S=500 : C=1
    sets the drawing start position to the center of the screen, and the color to white
20 DIM JS(15)
    creates space called an *array* for the joystick
    FOR K=5 TO 14 : READ JS(K) : NEXT
    forms a loop, to put numbers into the joystick array
30 POKE 55296+S,C : POKE 1024+S,81
    plots a shape and color on the screen
40 S=S+JS(PEEK(56320) AND 15)
    determines a new position, according to the joystick reading
50 IF S<0 THEN S=S+1000
60 IF S>999 THEN S=S−1000
    corrects the position if the drawing pointer moves off the screen
70 IF (PEEK(56320) AND 16) = 0 THEN C=C+1
    changes the color if the trigger is pressed
    IF C>15 THEN C=1
    sets the color back to white if all colors have been displayed.
80 GOTO 30
    go back to line 30 and do it all over again
90 DATA 41,−39,1,0,39,−41,−1,0,40,−40
    numbers for joystick array

Programs are a combination of three things: input/output, repetition, and conditional logic. These are the main concepts of the book. In the example, the input comes from the joystick—you send information into the computer, telling it what to do. The output is in the form of the pretty screen display. Every time the computer follows the instructions in lines 30–80, only one point is plotted. The effect of motion occurs because the computer repeatedly follows the instructions in those lines. That's the repetition. The conditional logic is needed when the drawing goes off the screen, or when the entire sequence of colors has been displayed.

Remember: This program is only a preview of what is to come. The details of how it works will become clear as you read the book.

How about one more program? This time, you can write it. The program will fill the screen with your name. Start the program by typing a line number. The number 10 will do. The BASIC instruction to make the computer write things on the screen is PRINT. Since you want it to write your name, put your name in quote marks after the PRINT. Put a semicolon at the end of the line, and press RETURN. The line should look something like this.

```
10 PRINT "CHRIS ";
```

We want the computer to print your name over and over again, so the second line should be as follows:

```
20 GOTO 10
```

Now type the command LIST and press RETURN. There is a listing of your first program. Type RUN, press RETURN, and you will see your name in print.

The Keyboard

Most of your interaction with the computer will require using the keyboard. Getting to know some of the special features of the keyboard will help streamline your learning.

When you turn on the Commodore 64, you are greeted with a message displayed in letters and numbers. These and other symbols that can be displayed by the computer are called *characters.* You will also see a flashing square, known as the *cursor.* You can put your own characters on the screen by typing on the keyboard. The space bar can be used with the letter and number keys to type the name of the computer.

```
COMMODORE 64
```

Every time you press a key, the corresponding character appears on the screen, at the location of the cursor. The cursor then moves one place to the right. To type the punctuation characters that are shown above the numbers on the digit keys, hold down one of the SHIFT keys while typing the digit keys. Using SHIFT with letter keys causes graphics characters to be displayed. Instead of holding down the SHIFT key all the time, press SHIFT LOCK. Now all characters typed will appear as if they are shifted. To release the SHIFT LOCK, press it a second time.

After you type 40 characters, you will have filled one screen row. When the cursor is pushed off the right edge of the screen, it will reappear at the left edge, one row lower. This is called *wraparound.* Usually you will type only a few characters on a line, and then press the RETURN key. This makes the cursor move back to the left edge of the screen, again one row lower.

If the computer displays the message SYNTAX ERROR when you press RETURN, don't worry about it at this point. Syntax refers to the grammar of the BASIC language. The computer displays this error message when you type something that does not belong in the language.

When you press RETURN without typing any characters, it just moves down one row. Do this enough times, and the entire screen display will move upward. This is called *scrolling,* and it's the vertical counterpart for horizontal wraparound. The screen will only scroll up. The writing which gets pushed off the top of the screen cannot be retrieved.

Hold down the SHIFT key and press the one marked with the Commodore symbol. The screen will change to lowercase. Now you will have to use the SHIFT key or SHIFT LOCK to get capital letters. To return to uppercase mode, press the SHIFT and Commodore keys again.

You can get some additional graphics characters by holding down on the Commodore key while you type the letter keys. Typing a digit key while holding down the Commodore key makes the cursor change color. All typing will now be in the new color. To get more colors, hold down on the key marked CTRL while typing a digit key.

If you press CTRL and type the key marked RVS ON, characters will appear in reverse mode, until you press CTRL and type the RVS OFF key or press RETURN.

Some keys do not put characters on the screen. Instead, they make the cursor move. The two cursor keys can make the cursor move in all four directions. Pressed alone, the cursor will move down or to the right. Hold down either the SHIFT or Commodore key while pressing the appropriate cursor key to move the cursor up or left.

The cursor keys will automatically repeat if you hold them down for about half a second. The space bar also has this feature.

Use the cursor keys to move the cursor into the middle of a word, and press the INST/DEL key to delete characters. To open up a gap to insert extra characters, hold down on either the SHIFT or Commodore key while typing INST/DEL. This key also repeats automatically.

Finally, the *home* position of the cursor is in the upper-left corner of the screen. Press the CLR/HOME key to make the cursor move to the home position. To erase every character on the screen when you home the cursor, use SHIFT or the Commodore key with CLR/HOME.

Numbers, Operators, Expressions, and Precedence

When you see the READY prompt, it means the computer is waiting for you to type something. Every time you typed a line and pressed RETURN in the last section, you got a SYNTAX ERROR. That meant you typed something which the computer could not understand. Here is a line that the computer will understand. Type this line and press RETURN:

```
?6+4
```

The computer responded by displaying the answer, 10, on the screen. The computer can also handle longer problems:

```
?345+98+4+7132+56
```

Now you see why the RETURN key is needed. It lets the computer know when you are done typing a line. Without it, the computer would not know whether you wanted to type something like ?6+4 or ?6+4+2. It also gives you the chance to correct typing mistakes before the computer starts interpreting the line. From now on, when we say *enter* a line, it means that you should type the line and press the RETURN key.

The computer supports the operations of addition, subtraction, multiplication, and division. The symbol used to indicate one of these operations is called an *operator.* In the examples, the plus sign was used for addition. Enter these three lines to see the other operators.

```
?9-6
?3*4
?12/6
```

Because letters of the alphabet have a special meaning to the computer, X or x cannot be used for multiplication, so the asterisk is used instead. The slash symbol is used for division. Incidentally, division by zero is illegal, so if you try to make the computer divide by zero, you will get the DIVISION BY ZERO error.

There is one other operation, exponentiation, but it will rarely be used in this book.

```
?3↑4
```

Exponentiation is used to raise a number to a power. The above line means to multiply the number 3 by itself four times, as in 3*3*3*3.

Operators can be mixed on one line. The result is called an *expression.* An expression is *evaluated* by the computer to produce an answer, or *value.*

```
?2*3+4
```

An important thing to remember when using expressions is that certain operators have precedence over others. Here is a list showing the hierarchy of operators, starting with the highest precedence. Operators which have the same precedence are evaluated in order, left to right.

↑	exponentiation
*,/	multiplication, division
+,−	addition, subtraction

To see precedence in action, enter this line:

```
?2+3*4
```

Because multiplication has precedence over addition, the computer will first multiply 3 and 4, then add 2 to the result. The correct answer to this expression is 14.

You can change the order in which an expression is evaluated by using parentheses. Parts of an expression in parentheses are always evaluated first.

```
?(2+3)*4
```

This time, the computer will add 2 and 3, then multiply the result by 4, and display the answer, 20.

Chapter 2

Statements

Statements

The PRINT Statement

So far you have typed several lines that started with a question mark and contained numbers and operators. Every time you pressed the RETURN key, the computer responded by displaying a number on the screen.

Whenever the computer displays something on the screen, it is said to be *printing* to the screen. You use the question mark to tell the computer to print to the screen.

Now we let you in on the question mark's secret identity. Try doing some of the earlier examples again, this time using the word PRINT in place of the question mark.

```
PRINT64
PRINT2+3*4
```

PRINT is the BASIC statement understood by the computer. The question mark is just a short form of that statement. For our purposes right now, PRINT and the question mark mean the same thing.

Why does the PRINT statement have a shorter form? Well, typing in a long program can be boring and time-consuming. Also, the more you type, the more likely you are to make errors. So, anything you can do to reduce the amount of typing you have to do is a real plus.

PRINT requires that you press five keys, and the PRINT statement is used often. The question mark requires only one keystroke, and that's about as short as you can get! So feel free to substitute the question mark for the longer form, PRINT, as you type in the examples in this book.

Incidentally, the term *print* is left over from the days before video terminals, when the computer's display device was a printer. Nowadays we can use our television set or monitor to see the information, and PRINT has come to mean "displaying information on the screen."

Summary

- The PRINT statement instructs the computer to display information on the screen. It is a frequently used statement in BASIC programming.
- The question mark is interpreted by the computer to mean the same thing as the PRINT statement.

Keywords and Statement Execution

Now we should take a moment to notice how statements are organized. When you type a line and press RETURN, the computer looks at the first thing in the line to see if it recognizes that word as a statement, or command. If so, the computer will do what the statement commands it to do.

Some statements require that additional information be placed after the statement name. In the case of PRINT, an expression is one example of this extra information.

Here is a typical PRINT statement: PRINT 5*3+9. In this statement, PRINT is the keyword. All the words that BASIC understands are called keywords, and all BASIC statements must begin with a keyword.

If the computer does not recognize the first word in a line, it will tell you so by reporting a SYNTAX ERROR. For example, if you type a line that starts with the word EXPLODE, the computer will reply with the error message.Your Commodore 64 does not know how to explode.

The 64 also expects perfect spelling. If you make a mistake while typing a keyword, even if you are off by just one letter, the computer will respond with a SYNTAX ERROR. The 64 can not make an educated guess as to what you intended to type. Therefore, PRITN64 will not work, and neither will PRINU64. Every letter must be correct for the computer to recognize the keyword.

On the other hand, you won't need to be as fussy about spaces. The following lines are legal and will work.

```
PRINT 64
PRINT         64
PRINT 2   +   3*   4
```

The computer does not recognize spaces, and will ignore them if they are used, provided you do not use them in the middle of a keyword (the statement name). The computer won't understand P R I N T.

A careful use of spacing simply makes it easier for you to read a line. The first line shown below looks nicer than the second one.

```
PRINT (3+4) * (5+2)
PRINT(3+4)*(5+2)
```

Summary

- All statements start with a keyword. Some statements require more information after the keyword. Placing a number or expression after the keyword PRINT tells the computer what to print on the screen.
- Keywords must be spelled correctly. Spaces cannot be inserted within a keyword.
- Spaces can be placed almost anywhere else in a line. They are used to make a line easier to read.

The POKE Statement

PRINT is used often because it is how your programs communicate with the people using them. POKE is used often because it lets your programs make changes in special locations inside the 64. POKE can be used to do a wide variety of things, so it is very powerful. For instance, you use POKE to create color graphics and sound on your Commodore 64.

The format, or *syntax,* of the POKE statement is the keyword, POKE, followed by a number (or expression), then a comma, and finally a second number (or expression). The first number designates a memory or hardware location in the computer. Memory locations are used to store information such as a program, the screen display, and system software. Hardware locations control things like the color of the screen, or the pitch of a music note. Each location contains a number that ranges in value from 0 to 255. The location is said to be changed when the value which it contains is changed to a new value.

Your Commodore 64 has a total of 65536 of these locations. Each location has a distinct number, from 0 to 65535. This means that location 0 is the first location. This idea may take a little getting used to, because when you count things, you usually start with 1. Just remember, zero is a number, too. And when you're counting memory locations or the values in those locations, you begin with zero.

Each location serves a different purpose. For example, location 53281 is a hardware location that controls the color of the screen background. There are 16 colors, represented by numbers in the range 0–15. (Remember, the first color is represented by 0.) If location 53281 contains a value of 6, the screen background is blue. That's the screen background color you see when you turn the computer on.

The color of the screen can be changed to red by putting a value of 2 into location 53281. The POKE statement is used to do this. First you may want to change the character color to white by holding down the CTRL key and pressing the 2 key. This will enable you to see the characters better. Then enter:

```
POKE 53281,2
```

The first number in the statement indicates the location to be changed, and the second number tells what new value the location should contain.

To change the screen color back to the original blue, enter this statement:

```
POKE 53281,6
```

Notice that when the screen background changed to red, the screen border remained light blue (represented by the value 14). The color of the screen border is controlled by location 53280. Change the screen border to red with this statement:

```
POKE 53280,2
```

While we have been using only numbers in our POKE statements, expressions can be used in place of the numbers, so the statement shown below is an alternate way of setting the background color to red.

```
POKE 53280+1, 6/3
```

You may want to experiment further and see what other colors can be put on the screen. Keep in mind, though, that your characters are white. So if you want to change the screen background to white, first change your character color by holding down CTRL and pressing the 1 key (for black). Otherwise, your characters will be the same color as the background, and the screen will appear to be blank.

Also remember that the 64 can display only 16 different colors, using the numbers 0–15. If you POKE locations 53280 and 53281 with a value larger than 15, you will not get any new colors. Instead, numbers beyond 15 simply repeat the colors obtained with values from 0 to 15. POKE 53280,2 and POKE 53280,18 both set the border color to red.

If you try to POKE a location with a value outside the range from 0 to 255, you will get an ILLEGAL QUANTITY error.

When you are through experimenting, set these locations back to their initial values by holding down the RUN/STOP key and pressing the RESTORE key.

Here is a very important word of caution. The POKE statement is very powerful, but with added power comes added responsibility. You should not carelessly change unknown locations. It is not possible to physically damage your Commodore 64 by randomly POKEing locations, but you can accidentally force the computer to ignore your keyboard entries, so that it appears to stop functioning completely. This is called a *system crash.*

To set everything back to normal, you must turn the computer off and then back on. Pressing RUN/STOP and RESTORE will reset some things, but it will not always get the computer back on track after a stray POKE. To be safe, you should POKE only those locations described in this book and in other Commodore 64 literature. Crashing the computer is really no big deal; just be thankful the damage is not permanent.

Summary

- The Commodore 64 has 65536 memory locations. They are numbered from 0 to 65535.
- Some locations are used to hold information for your use. Other locations are related to the hardware. The examples use hardware locations that control the screen border and background colors.
- Each location contains a value from 0 to 255.
- The POKE statement is used to replace the contents of a location with a new value.
- The syntax for the POKE statement is the keyword POKE, a number or expression indicating which location is to be changed, a comma, and a second number or expression which is the new value to be placed into the location.
- The wise programmer will avoid carelessly POKEing unknown locations. Although RUN/STOP and RESTORE will reset some things like screen color, they may not undo the results of stray POKE statements. The only sure way to restore the system in such a situation is to turn it off and then on again. Fortunately, crashing the computer does not cause permanent damage.

Multiple Statements on One Line

Now that you have given the PRINT and POKE statements a workout, you have seen how the computer accepts a line that you enter, looks for a keyword that it recognizes, and executes the statement. Thus far you have been putting only one statement on each line. But often you will want to put several statements on one line.

This is done using the colon (:). The purpose of the colon is to tell the computer where one statement ends and the next begins. Try this example to change the whole screen to a light green color:

```
POKE 53280, 13 : POKE 53281, 13
```

Here is an example of several statements on one line:

```
? 64 : ? 46 : ? 55 : ? 123 : ? 1024
```

After you pressed the RETURN key, the computer should have printed a column of numbers before it printed the READY prompt.

Notice that if you type the above line using PRINT instead of the question mark, the line will wrap around.

```
PRINT 64 : PRINT 46 : PRINT 55 : PRINT 123 : PRI
   NT 1024
```

However, the results are the same as before.

A line can be longer than one row, but it cannot be longer than two screen rows, which is 80 characters. If you type 81 or more characters without pressing the RETURN key, the computer will accept only the first 80 characters, and ignore the rest of the line. When your characters reach the end of the second row, they will wrap around to a third row, but when you press RETURN, none of the statements on the third line will be executed. And if the wraparound from the second to the third line occurs in the middle of a statement, that statement will be incomplete, and the computer will generate a SYNTAX ERROR. To prevent any problems, always type lines short enough so that they do not reach to the end of the second row.

Summary

- A colon (:) is used to separate several statements on one line.
- The maximum line length the computer can handle is two screen rows (80 characters). Characters that spill over into a third row are ignored.

Chapter 3

Variables

Variables

Introduction to Variables

So far, we have only used numbers with PRINT and POKE statements. And each time we wanted to change the number we were printing or change the screen color, we retyped the line. But there is an easier way to represent numbers.

While entering the demonstrations for PRINT and POKE, did you wonder what would happen if you typed something other than a number, such as a letter of the alphabet? Try it right now.

```
PRINT A
```

The number 0 is printed on the screen instead of an A. Try a couple more letters.

```
PRINT B : PRINT C
```

The same thing happens. We type a letter, and the computer prints a 0.

This does not mean that the computer interprets every letter as the number 0. Rather, the computer considers the letter to be something called a *variable*, and it is the variable which has the value of 0.

A variable consists of two things: a name and a value. It is called a variable because its value can *vary*, or change. The number 5 always has a value of 5, and PRINT 5 will always print a 5 on the screen. But a numeric variable can have different values at different times to represent any number.

The variables that you have seen so far (A, B, and C) have all had a value of 0. This is because you have not given them a value, and the computer assigns a *default* value of 0. The method by which a variable is given a new value is the subject of the next section.

Summary

- A variable consists of a name and a value.
- Letters of the alphabet may be used as names for variables.
- The value of a variable can be changed. This is what distinguishes a variable from a number.
- A numeric variable can have any value that is valid for a number.

Variables

Assigning Values to Numeric Variables with LET

All variables have a default value of 0, but you can change it by assigning a new value. Variable assignments are done with the LET statement.

```
LET A=6
```

The syntax for the LET statement is quite different from that for PRINT or POKE. The syntax is the keyword LET, the variable name, an equal sign, and then a number or expression. The variable name identifies which variable is being assigned, and the number or expression specifies the new value.

When the computer executes a LET statement, the first thing it does is look at the part of the statement to the right of the equal sign to get a value. If there is an expression, it will have to be evaluated. Once the computer has a value, it is assigned to the variable specified to the left of the equal sign. Now the variable can be used just like a regular number, as in the PRINT statement.

```
PRINT A
```

Now we have solid proof that a variable can have a value other than 0, and that the value can be changed by the LET statement.

When you entered the LET statement, the computer responded with the usual READY prompt. The LET statement itself does not cause anything to be printed. The value assignment takes place in the computer's memory.

Assigning a value to one variable does not affect other variables. The variable B still has its initial value of 0.

```
PRINT B
```

But we can easily change that.

```
LET B=4
PRINT B
```

Since variables are completely independent, A still has its value of 6, even though B has been assigned in the meantime.

```
PRINT A : PRINT B
```

Once a variable has been assigned a value, it doesn't mean that it has to have that value forever. The value can be changed by another LET statement.

```
LET A=5 : PRINT A
```

Once a variable has been assigned, however, it retains that value until it is changed again. Also, two different variables can have the same value, without creating a conflict.

```
LET B=5 : PRINT A : PRINT B
```

Although the value of a variable can be changed, a variable can have only one value at any given time.

Here are more examples:

```
LET A=66 : LET B=44
PRINT A+B
PRINT A-B
PRINT A/B
PRINT C
PRINT A+B-C
LET C=10 : PRINT A+B-C
```

A statement like LET 5=A is illegal, because the number 5 cannot be assigned a new value. The statement LET A+1=2 will not work, either, because only the variable name can be to the left of the equal sign.

Summary

- All variables have an initial, or *default,* value of 0.
- Variables are assigned new values with a LET statement.
- The syntax for the LET statement is the keyword LET, a variable name, an equal sign, and a number or expression.
- During execution of a LET statement, the value to the right of the equal sign is determined first, and then it is assigned to the variable specified to the left of the equal sign.
- Variables can be used just like numbers.
- The LET statement does not cause anything to be printed on the screen.
- Variables are independent, so assigning a value to one variable does not affect the values of other variables.
- There is no reason why two variables can't have the same value, even though the variables are independent.
- A variable keeps its value until it is assigned a new one.
- A variable can have only one value at a time.
- Numbers cannot be assigned new values.

- The computer will not print a variable name when it is instructed to print the value of a variable.

Uses of Variables

So we can assign a value to a variable. But why would anybody want to do this? As it turns out, variables are very useful, and many programs use variables more often than they use ordinary numbers. This section presents some reasons why.

Remember how you changed the screen color using the POKE statement? Each time you typed the POKE statement, you had to carefully type the correct location number. By assigning the location number to a variable, it becomes much easier to change the screen color, like this:

```
LET A=53281
POKE A,2
POKE A,6
```

Once you have assigned the value 53281 to the variable A, the variable name can be typed every time in place of 53281, with less chance of making a mistake.

Clear the screen and type the following statements. After each one, the screen will change color, and the computer will print the READY prompt. This is similar to an earlier demonstration, except that this time the variable A is used in place of the 53281.

```
POKE A,4
POKE A,7
POKE A,8
POKE A,13
```

The next step is to move the cursor back to the top of the screen, right on top of the first POKE statement. As before, press the RETURN key four times to rapidly change the screen color. Now that the cursor is at the bottom of the screen again, we are going to repeat the demonstration. But this time, before you move the cursor to the top of the screen, make one small change. While the cursor is still at the bottom, type this line:

```
LET A=53280
```

Okay, now you can continue by moving the cursor to the top of the screen and pressing RETURN repeatedly, just like last time.

Again the screen rapidly changes color, but this time it is the border that changes, not the background.

This demonstrates one of the most powerful uses of variables—changing the value of a variable can save a lot of extra time and work. If the number 53281 had been used instead of the variable A in this example, every line would have had to be retyped in order to make the statements change the border color.

This section has barely touched on the uses of variables; many more uses will become obvious to you as you program. Variables are a very important concept in computing, not just in BASIC. One small change to a variable can have a big effect.

Summary

- A variable can be assigned the value of long numbers. Typing becomes faster and less prone to error.
- The real power of variables lies in the fact that their values can be changed. Reassigning one variable can save the work of retyping several lines.

Variable Names

Only three examples of variable names have been shown thus far. Do variable names have to be only one letter long? Can other symbols besides letters of the alphabet be used? Just what constitutes a legal variable name?

Here are the rules governing valid choices for variable names.

1. A variable name can contain any number of characters. There is no restriction on length, other than the maximum line length (80 characters).

2. Only the first two characters are significant.

3. The first character in the name must be a letter of the alphabet.

4. Any characters after the first can be letters, or the digits 0 through 9.

5. Spaces can be put between the characters in a variable name, but this is not recommended.

6. The variable name can not contain any reserved words.

A variable name always starts with a letter because the computer must distinguish between numbers and variables. If the variable name started with a digit, the computer would think that a number was being used. This means that there are 26 different characters that can be used as the first character in a variable name.

Variables are used so often that 26 of them would not be enough. So a second character can be added to the variable name. This second character can be one of the 26 letters of the alphabet, or it can be one of the ten digits, from 0 to 9. Such a character is said to be *alphanumeric.* By combining these two characters, you can come up with 936 different variable names. This is many more than you will need. Examples of legal variable names are shown here.

A B C D Q X Y Z
AA BC DZ DQ QD XY ZZ WM
A1 B2 C1 D9 Q0 X7 Y3 Z4 G4 J8

Two-character names offer many possible variable names, but these names may not be very descriptive. Given variable names like those above, it would be hard to associate them with something. It would be nice if whole words could be used as variable names.

To allow this, additional alphanumeric characters can be added onto the first two characters in a variable name. Now you can use full words and even your own name as a variable name. Here are a few examples.

OBJECT, THING, COMPUTER
NUMBER, AGE, TAXRATE
UNDERDOG, TWEEDLEDEE, CHRIS

There is one problem, however, when using variable names longer than two characters. All characters after the first two are ignored by the computer. Only the first two characters are said to be *significant,* meaning the computer uses only the first two characters to distinguish between one variable and another. Additional characters are for cosmetic purposes only, to help you remember what the variable represents. A variable named TAXRATE is more quickly understood than the variable T. But be careful—names like CHRIS and CHARLIE may appear to be different variables, but as far as the computer is concerned, both are the same variable.

A common practice is to use variable names that have the same first letter, and then use different numbers as the second character, such as X1, X2, and X3. Notice that this would not work for longer names, such as RR1 and RR2 or S1 and S10.

In an attempt to keep the examples in this book as short and simple as possible, variable names are limited to two characters. For longer programs, however, the advantages to using longer variable names are worth the extra typing. If you choose to use longer variable names, check for conflicts when you use a variable name for the first time.

There is one last restriction that must be observed. The reserved words of the BASIC language must not be embedded in a variable name. This rule will be hard to follow at first, since you have seen only a few of the reserved BASIC words. To help you, the computer will print SYNTAX ERROR whenever you use a variable name that contains a reserved word. Reserved words include keywords, names of operators, names of functions, and predefined variables.

Here are four examples of variable names that are illegal because they contain a reserved word.

```
TOTAL COST NOTE COLOR
```

The name TOTAL is illegal because it contains the keyword TO. The trigonometric function COS (cosine) makes COST illegal. NOTE and COLOR are illegal because they contain the operator names NOT and OR, respectively.

There are two variable names which have special meaning to the computer and which you may not use. The first *reserved* variable is ST, which reports system status information. The other is TI, which stands for TIME. The value of variable TI reflects the value of the clock maintained by the computer. Because the clock is incremented every 1/60 second, the value of TI keeps changing. TI has a value of zero when you first turn the computer on, and it keeps increasing.

```
PRINT TI
PRINT TI
PRINT TI
```

If you try to assign a value to ST or TI, you will get a SYNTAX ERROR.

The guidelines presented here for choosing variable names may seem overwhelming, but with just a little practice, you won't even have to stop and think to choose a variable name.

Variables

Summary

- The first character in a variable name must be alphabetic.
- If a variable name contains more than one character, the others must be alphanumeric (a letter of the alphabet or a digit). Spaces are also allowed, but do not count as characters, so it is suggested that they not be used.
- A variable name can be any length, but only the first two characters are significant.
- It is best to choose a variable name which reflects the purpose and use of the variable.
- The computer will generate a SYNTAX ERROR if a variable name contains a reserved word.
- There are two reserved variables: ST and TI. They cannot be assigned.

Assigning Variables to Variables

Consider that variables can be used just like numbers. Now consider that the syntax for a LET statement specifies that a number or expression must be to the right of the equal sign. Could a variable be used in place of the number? Could variables be used in the expression? Yes.

Remember that when the computer executes a LET statement, it first looks to the right of the equal sign for a value. If the computer finds a number, that is the value. If it finds a variable, the value to be used is the value of the variable. Or if an expression is found, the expression is evaluated to produce a value. It doesn't matter whether the expression contains variables or not. All that matters is that a value can be determined by examining what is to the right of the equal sign.

The value having been determined, it is assigned to the variable specified to the left of the equal sign. The whole process really isn't that complicated. Just remember that the computer processes the right side, then the left.

With that in mind, the following statements should not present any difficulties.

```
LET A=3 : LET B=A+4
PRINT A : PRINT B
```

The first assignment is just like the ones you have seen before. The second is really no different. The computer sees A+4 to the right of the equal sign. The value of A is 3, so the value of A+4 is 7. The value 7 is assigned to B. This is verified by printing B.

Now enter this line:

```
LET A=B : PRINT A : PRINT B
```

Both A and B now have the same value, because the value that B contains has also been assigned to A. Notice that if B were now changed, A would still retain its value.

```
LET B=8 : PRINT A : PRINT B
```

Now that you have seen variable assignments involving other variables, the following variable assignment should not surprise you:

```
LET A=A+1
```

If you have a background in mathematics, this statement may bother you a little bit. There is something inherently wrong in saying that A equals itself plus something else. But remember, this statement is not an equation: It is an *assignment*. When the computer executes this statement, it takes the current value of A, adds 1 to it, and assigns that value to A. Recall that a computer executes the right side of the equal sign, then the left. To examine this more closely, enter this line:

```
LET A=1 : PRINT A
```

Now type:

```
LET A=A+1
```

The first thing the computer does is look at the A+1. Variable A still has a value of 1. Then A+1 becomes 1 plus 1, which is 2. Finally, the value 2 is assigned to the variable A.

```
PRINT A
```

Whenever this statement is executed, the value of variable A is *incremented* by 1. Remember this little statement, because you will see it again.

Summary

- When the computer executes a LET statement, it follows a two-step process. First the computer determines a value by examining the portion of the statement to the right of the equal sign. Then it assigns that value to the variable specified to the left of the equal sign.

- The portion to the right of the equal sign can be an expression containing one or more variables. The variables will be treated just like numbers. This means that the value of one variable can be assigned to another variable.
- Statements like LET A=A+1 are legal and in fact are very useful, as will be seen later.

Erasing Variables with CLR

When you turn on your Commodore 64, all variables will have the default value of 0, until changed by an assignment. After the variables have been assigned different values, they can all be set back to 0 by turning the computer off and back on again.

But that's a somewhat awkward way to erase all the variables, so the CLR statement is used to *clear* variables. Every variable has the value of 0 once the CLR statement has been executed. You could think of it as an instant assignment of 0 to every variable.

The syntax for CLR is simple; it consists of the keyword CLR, and nothing else.

```
LET A=1 : LET B=22 : LET C=333
PRINT A : PRINT B : PRINT C
CLR
PRINT A : PRINT B : PRINT C
```

This statement is used in situations where memory that has been allocated for variables needs to be reclaimed for other uses.

Every time you assign a variable for the first time, a small amount of the computer's memory is set aside to keep track of the variable's name and value. When variables are cleared with CLR, this memory is freed and can be used for something else. CLR is used only in programs that require a lot of memory. For our purposes at present, CLR is not very useful.

Summary

- A variable is *cleared* when it is reset to its initial value of 0.
- The statement CLR is used to clear all variables. Every variable is set to 0 by the CLR statement.

- The syntax for CLR consists only of the keyword CLR. Unlike PRINT, POKE, or LET, no numbers are needed in the statement.

The Optional Keyword

Variables are used very often in computer programs; therefore, variable assignments are needed quite often, too. We saw that the question mark could be used as a short form for the keyword PRINT. The shortcut in typing the keyword LET is to leave it out. The keyword LET is optional.

```
A=987 : PRINT A
```

Until now, we have always used the keyword LET to help emphasize the concept of statements—a keyword, sometimes followed by extra information. Now that you have seen and used four different statements, it is okay to reveal that the keyword is optional for this statement.

Of course, the LET statement is the only one which has an optional keyword. If the computer tries to execute a statement that does not have a keyword, it assumes that the statement is a variable assignment. This is called an *implied* LET statement. This also eliminates the possibility of any other statements having optional keywords. The other parts of the variable assignment statement are still required, including the variable name, the equal sign, and the new value.

Variable assignments without the keyword LET are no different from those that do have the keyword. For the remainder of this book, all variable assignments using the LET statement will be printed without the keyword, so you will not have to type it. This will save you three keystrokes every time you assign a variable, which is the whole idea behind making the keyword for this statement optional.

Summary

- As a shortcut, the keyword LET of the variable assignment statement does not have to be typed, it is optional.
- LET is the only optional keyword. All other statements require a keyword. If no keyword is provided, the computer assumes that the statement is a LET statement.
- A LET statement which does not include the keyword LET is called an *implied* LET statement.

- The rest of the syntax of the LET statement after the keyword is still required.
- The remaining examples in this book will not use the keyword LET.

Demonstration of Sound

This section explores the sounds that can be made by the Commodore 64. In the process, it will help you gain some experience in the use of variable names. To start, reset the computer by holding down RUN/STOP and pressing RESTORE. The screen should be cleared, and the READY prompt should appear near the upper-left corner of the screen.

To make sounds on the Commodore 64, it is necessary to change a lot of hardware locations, using POKE statements. Rather than typing the location numbers again and again, let's assign the location numbers to variables. Enter the first line, which assigns seven variables.

```
MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:
CT=54276:PW=54275
```

There are some numbers which will be used frequently as values in a POKE statement, so these numbers are also assigned to variables.

```
T=16:S=32:P=64:N=128
```

Now we are all set to begin the demonstration.

The first location controls the master volume. Volume levels range from 0 (no volume) to 15 (maximum volume). The variable MV represents this location, so POKE MV to set the master volume to the maximum level.

```
POKE MV,15
```

The master volume controls only the overall loudness of a voice, not the volume changes that occur during the duration of a note. When a note is played, it does not stay at full volume. After it reaches full volume, it begins to fade away.

Notes played on some instruments fade away faster than others. For example, the sound from a tuba stops when the person stops blowing, but the sound from a gong continues for quite a long time. These volume changes during the duration of a note can be analyzed in four parts: attack, decay, sustain, and release.

The rate of attack controls how quickly a note reaches full volume. After the volume peaks, it levels off. This is called the decay, which can also be controlled to happen at a faster or slower rate. The volume level after the decay is called the sustain level. From this level, the volume will fade, and the rate at which it does so is called the release rate.

The duration of a note can be divided into two parts. The first part, when the note starts, includes the attack and decay rates and the sustain level. The second part of the note is when it is released—when the volume starts to fade. This is controlled by the rate of release. These changes in volume during the duration of a note combine to form the note's *envelope.*

Different musical instruments have different characteristic envelopes. The fact that your Commodore 64 can change the envelope of a voice is one way in which it can effectively simulate various musical instruments.

The rates of attack and decay are controlled by the hardware location which we labeled AD. The attack and decay rates can range from 0 to 15, with 15 being the slowest rate. To POKE both numbers into the one location, multiply the attack rate by 16 and add it to the decay rate. Type the next line, but do not press RETURN.

```
POKE AD, 8*16+7
```

The sustain level ranges from 0 to 15, with 15 meaning that the note will stay at full volume. The release rate is from 0 to 15, just like the attack and decay rates. To POKE both values into the location, you must multiply the sustain level by 16 and add it to the release rate. Finish typing the line so that it looks like the one below.

```
POKE AD, 8*16+7 : POKE SR,12*16+9
```

These two POKEs set an attack rate of 8, decay rate of 7, sustain level of 12, and release rate of 9.

Another characteristic of a note is its pitch, also called frequency, controlled by two locations, FL (frequency low) and FH (frequency high). Enter this line to set the frequency.

```
POKE FL,0 : POKE FH,20
```

Just about everything has been set to play a note. The only thing left is to choose a waveform.

Variables

The notes produced by different instruments have different waveforms, so having a selectable waveform is another way of simulating various musical instruments. For our first waveform selection, we will use what is called the *triangle* wave, which is represented by the variable T. We will use the variable WF to represent our waveform selection, so we assign the value of T to WF. When you add one to the waveform and POKE the result into the control location, CT, the note is started. You should hear a note upon entering this next line (you may have to adjust the volume on your television set).

```
WF=T : POKE CT,WF+1
```

You heard the quick attack and decay, and you still hear the note because it is playing at the sustain level. To release the note, enter the following line:

```
POKE CT,WF
```

Admittedly, it took a little bit of work just to make the computer produce one note. This is not necessary every time you want to play a note, however, because all of the settings stay in effect until they are changed. To play the note again, all you have to do is move the cursor up to the line which said WF=T : POKE CT,WF+1 and press RETURN. Then, to finish the note, just press RETURN a second time.

After you have played the note a few times, move the cursor up to the line which set the envelope. You might reduce the sustain level.

```
POKE AD, 8*16+7 : POKE SR, 4*16+9
```

Move the cursor back to the first line which POKEs CT, and press RETURN. After the attack and decay, the low sustain level causes the note to be very quiet. Release the note, then change the level back to 12 and make the attack and release rates slower.

```
POKE AD,12*16+7 : POKE SR,12*16+12
```

After you have experimented with the envelope a little, start the note playing again, but this time do not let it release. While the constant tone is playing, move the cursor up to the line which set the frequency locations. Change the value POKEd into FH to a 30, and press RETURN. You should hear the pitch get a little higher. Try different values, from 2 to 255, for lower and higher frequencies. The smaller the number, the lower the frequency. (Values of 0 and 1 are almost inaudible.)

Set the value in FH back to 20, listen to the tone, and then notice the change in pitch when you change the value to 21. Now change the value in FL from 0 to 1: You will not be able to hear any difference. But when you change the value to 40, you will hear a slight increase in pitch. A value of 80 again causes only a small change in the frequency. Also try values of 120 and 240.

Location FL, *frequency low,* is like a fine-tuning control for the frequency. *Frequency high* changes the frequency by bigger intervals. The names frequency low and frequency high are a little misleading, because they do not mean that one location is used for low pitches, and the other for high pitches. Think of a car odometer displaying a number like 10753. The lower digits (the ones to the right) change as you travel a short distance. The higher digits represent bigger numbers, and do not change as often. The highest digit, 1, will change to 2 only after the odometer reaches 19999.

The terms *low* and *high* apply in the same way to the frequency locations. If you now set FL back to 0 and FH to one greater, there will be just a small change in pitch.

```
POKE FL,0 : POKE FH,22
```

To change the waveform, change the assignment of WF from T to S, for the sawtooth wave, and press RETURN. Because the note had already been started, the attack and decay will not happen again, but you will hear a difference in the tone due to the changed waveform. Now press RETURN the second time to release the note. To better hear the difference in waveforms, keep alternating between the values T and S for WF at different frequency levels. Notice that because you used the variable WF, you have to change only one line every time you change the waveform. The value in WF affects both lines which POKE to the control location, CT.

Another waveform is called the pulse wave, which we have labeled P. Type the variable name P in the place where T and S have been used, and press RETURN. This time, you won't hear anything, because one other location must be set before the pulse wave can be used. The width of the pulse wave can be varied to make the wave square or rectangular. Values in the range from 0 to 15 correspond to different degrees of rectangles, with a value of 8 creating a perfect square wave.

At a free spot on the screen, POKE location PW (pulse width) with a value of 8, and you will hear the tone. Change the width for different effects, but keep the waveform as P. Location PW affects only the pulse wave, and will have no effect when other waveforms are used.

```
POKE PW,8
```

To make a note play with only one keypress, first set the attack rate to 0, and then combine the two lines which POKE the control location.

```
POKE AD, 0*16+7 : POKE SR,12*16+9
WF=P:POKE CT,WF+1 : POKE CT,WF
```

You can play the note several times by moving the cursor back up to this line and pressing RETURN. Just let the volume fade completely before you try to start the note again.

The sound that you hear when a television station goes off the air is called *white noise,* because it is the combination of all frequencies, just as white light is the combination of all colors. Many percussion instruments, like a snare drum, generate a white noise sound. The last waveform is called *noise,* which has been labeled N. Change WF to N and press RETURN for an example of white noise.

Chapter 4

Output

Output

The PRINT Statement Revisited

PRINT is one of the most useful and versatile output statements in the BASIC language. In this chapter, we will explore some of the more powerful ways the PRINT statement can be used, such as screen formatting and displaying messages.

Thus far, we have used PRINT to display numbers on the screen one line at the time. You have noticed that after you PRINT a value, the cursor automatically moves to the leftmost column on the next screen row.

This is just like what happens when you type a line and press the RETURN key. You could say that the computer prints a RETURN at the end of the line. This is a simple example of how you can use PRINT to format the screen, or place your output characters where you want them.

You can use this same general idea to print blank lines. The syntax for PRINT allows the keyword to stand alone. When the computer executes a PRINT that is not followed by anything, it will move the cursor back to the left edge if it isn't already there, and then move it down one row. This prints a blank line on the screen:

```
PRINT 6 : PRINT : PRINT 4
```

The statement PRINT without an operand is used to separate blocks of printed information on the screen. By moving the cursor to the bottom of the screen using repeated RETURNs, you can use 25 empty PRINT statements to clear the screen, although it would be much faster to simply hold down the SHIFT key and press the HOME key.

```
?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:?
```

Summary

- PRINT is a versatile statement used to output information to the screen.
- If a value is placed after the keyword, it will be printed beginning at the place on the screen currently pointed to by the cursor.
- After a PRINT statement is executed, the computer prints a RETURN, which moves the cursor back to the left edge and down one row.
- If no value is specified after the PRINT keyword, the automatic RETURN leaves a blank line on the screen.

Formatted PRINTing Using the Comma and Semicolon

We have learned that we can use the automatic RETURN built into the PRINT statement to skip to the next line, and to leave a blank line between values printed on the screen. There will be occasions when you want to print more than one number on the same row, with spaces between the numbers instead of between the lines. You can achieve this by using just one PRINT statement followed by several numbers separated by commas. When printed, the numbers are shown separated by spaces. Take a moment to try these examples.

```
PRINT 41276, 987
? 64, 46, 55
PRINT 1,2,3,4,5,6,7,8,9
```

In the last example, you'll notice that there are too many numbers to fit on one line. The wraparound feature introduced in the first chapter also applies to printing done by the computer. When the computer runs out of room on one row, it will continue the printing on the next row, starting back at the left edge.

You will also see that the numbers on the lower lines appear right below the numbers on the first line. The comma can be used with the PRINT statement to make the computer print information in columns.

The comma makes the PRINT statement more versatile by allowing several numbers to be printed at once, in ordered columns.

Sometimes you may want to print several numbers on one line, but you won't want them to be spaced so far apart. To accommodate this need, the semicolon (;) can be used in place of the comma. Numbers separated by semicolons in a PRINT statement will be printed closer together on the same line.

```
PRINT 41276; 987
PRINT 64; 46; 55
PRINT 1; 2; 3; 4; 5; 6; 7; 8; 9
```

With the closer spacing provided by semicolons, wraparound will not occur as often. You can also mix semicolons with commas.

```
PRINT 64; 46, 55; 20
```

Here is one more feature of using commas and semicolons. If a PRINT statement ends with either of these two symbols, the computer will not print the RETURN after the other information. The cursor will remain in place, and will not move until more printing to the screen is done. In this way, several PRINT statements can display information on the same row.

```
PRINT 64; 46, : PRINT 55; 20
```

Notice that when the computer prints a message such as the READY prompt, the effect of the comma or semicolon is lost.

Summary

- To print more than one value on a line, separate the items to be printed with a comma or semicolon.
- A semicolon is used for close spacing, and a comma is used for far spacing.
- The far spacing of the comma can be used to print numbers in columns.
- If output exceeds the length of one line, printing will wrap around to the next line.
- If a PRINT statement ends with a comma or semicolon, the computer will not print the RETURN, and the cursor will not move.

PRINTing Character Strings

So far we have used the PRINT statement only to print numbers on the screen. What if we wanted to print something like a message or a name?

```
PRINT CHRIS
```

As we have learned, the computer interprets CHRIS as the variable CH, and prints the value of CH. To make the computer print text verbatim, without trying to interpret words as variables, a special symbol must be used in the PRINT statement. This special symbol is the double quote mark, obtained by pressing a SHIFTed 2 key. This symbol is used to indicate the beginning and end of text in a PRINT statement. To print a word or name, enclose it in quote marks.

```
PRINT "CHRIS"
```

Output

Voilà! Finally, the computer has printed something other than a number. When the computer sees a quote mark in a PRINT statement, it copies all succeeding characters directly to the screen until it encounters a second quote mark. A group of characters like this is called a *character string*. The quote marks are called *delimiters* because they indicate the beginning and end of the character string. Remember to type the closing quote mark.

```
PRINT "CHRIS" : PRINT 64
```

The printed text does not have to be just one word. A full name can be printed.

```
PRINT "CHRIS MAKEPEACE"
```

Or the computer can print a whole sentence.

```
PRINT "THE COMMODORE 64 FEATURES SOUND AND
    COLOR GRAPHICS"
```

As with numbers, printing of text will wrap around, but it does not automatically split a sentence only at a space. The computer will split a word at any point. To make sure that long sentences do not cross the right edge of the screen, use several PRINT statements.

```
PRINT "THE COMMODORE 64 FEATURES" : PRINT
    "SOUND AND COLOR GRAPHICS"
```

Professionally written software never splits sentences at any place other than at a space.

Since the computer does not care which characters are printed, you can print a character string of digits.

```
PRINT "1234567890"
```

Take a moment to experiment with printing character strings. You will discover that the editing keys do not work when you are typing a line that contains a character string. After you type an opening quote mark, the only editing key that still works is DELete. All the others cause characters to be printed. Try this with CTRL and a number. In the next line, press CTRL-4 right after you type the opening quote mark.

```
PRINT "COMMODORE 64"
```

The color change did not take place until the character string was printed. This trick can also be used with other keys, such as those which control reverse video and cursor movement. Type the line again, but before you type 64, type a couple of cursor downs. When the string is printed, the 64 will appear two rows below the rest of the line. This editing feature is called *quote mode editing,* and is canceled when you type the second quote mark.

The only thing that you cannot put in a character string is the double quote mark. (A method for getting around this problem is given in a later chapter.)

Go ahead now—give the PRINT statement a workout!

Summary

- A character string is a sequence of characters.
- A character string can be printed by the PRINT statement if it is enclosed by double quote marks.
- The quote marks are needed so the computer can distinguish between a word of text and a variable name.
- Printed text will wrap around, possibly causing a word to be split.
- Multiple PRINT statements can be used to prevent the splitting of words. It adds a professional touch to a program if a sentence is split only at a space.

Mixing Numbers and Character Strings

You can print numeric values and character strings with one PRINT statement. It is just as simple as this:

```
MEM=32000 : PRINT "THERE ARE" MEM "BYTES FREE"
```

This statement prints a message like the one that appears when you turn the computer on. The computer knows that MEM is a variable in the PRINT statement because it is not enclosed by quote marks. Only the words THERE ARE and BYTES FREE are character strings.

When necessary, commas and semicolons can also be used between numbers and character strings.

```
NUMBER=6 : ? "TWICE" NUMBER "IS"; : NUMBER=
    NUMBER*2 : PRINT NUMBER
```

The output is only one line long, and appears as though only one PRINT statement was used. Two PRINT statements were actually used, but the semicolon was used to cancel the normal carriage return. The above example is simple, but it would be simplified even further by using only one PRINT statement.

```
NUMBER=6 : ? "TWICE" NUMBER "IS" NUMBER*2
```

Although numbers and character strings can be used together in PRINT statements, there are some limits on how they can be used. You cannot use character strings in place of numbers for math operations.

```
PRINT "CHRIS"+7
PRINT 124-"1"
```

Neither of these statements will work. The computer generates a TYPE MISMATCH error, which means it has detected an attempt to mix unlike things incorrectly. The things being mixed incorrectly are numbers and character strings. "CHRIS" is a character string. It is of the *type* character string. A type identifies kinds of values. The character string "CHRIS" does not have a numeric value. Character strings are not one of the types which can be mixed in math operations.

Summary

- One PRINT statement can print both numbers and character strings.
- Commas and semicolons can be used regardless of whether numbers are mixed with character strings.
- The word *type* refers to the quality of a value that determines what sort of information is being represented. The two types that have been introduced thus far are numbers and character strings.
- Attempts to use the arithmetic operators with character strings cause the computer to generate a TYPE MISMATCH error.

Chapter 5

Functions

Functions

Introduction to Functions

Functions belong in a class all by themselves. They are not statements, they are not operators, and they are not variables. Functions are used to produce values which can be used by statements, such as the value to PRINT on the screen or POKE into a location. You cannot use a function by itself; it must be used as if it were a number in a statement. The correct terminology is to say that a function is *called* by a statement.

Functions are like variables in that they have names, but functions are not assigned values. A function is a process; it must be given a value in order to produce a second value. The function is said to *return* a value.

This chapter introduces six functions which you will be using later in your programs. Even though functions may seem to deal only with arithmetic, you will learn that they are important tools for developing good programming logic.

In mathematics, a function is often written using the notation Y=F(X). The F stands for function, the X is the value given to the function to work on, and the Y is the result. This is a good example of the input/output concept; a value goes in, it is processed, and a value comes out. Different functions process numbers in different ways.

Functions in BASIC work in a similar way. Here are three examples showing how functions are used.

```
V=RND(34)
PRINT SGN(Y-Y0)
POKE 53280,ABS(C)
```

The arithmetic operators (+ − * /) require two or more values to produce a result. A function performs a process on only one value, but because this process can be broken down to a lot of operations, a function is much more complicated than an operator.

The names of BASIC functions consist of at least three letters. A function's name is immediately followed by a pair of parentheses containing a value which can be a number, a variable, or an expression. This value in parentheses is the input value, or *argument,* for the function. No spaces are allowed in the function name or between the name and the opening parenthesis.

Although they are used with only moderate frequency, functions have a wide range of uses. The functions introduced in the next few sections are easy to understand, and they'll be used in later chapters.

Summary

- A function does not have a value. Rather, a function is a process which produces a value. The value produced by the function can then be used like a number. When a function produces a value, it is said to *return* the value.
- A function is not executed. Rather, it is *called* by a statement.
- A BASIC function has a name which is made up of at least three letters of the alphabet.
- The function name is followed by a pair of parentheses. There can be no spaces in the function name.
- The parentheses enclose the input value, or *argument,* for the function. A function uses only one argument.

Absolute Value

The absolute value function is a common function in mathematics, usually written as two vertical bars enclosing a number. The absolute value of a number is the number's value without the sign.

A number contains two parts: an absolute value, and a sign (plus or minus). The absolute value function removes the sign from a number. If the argument is positive, the value is returned unchanged as the function value. If the argument is negative, it is negated first, making it positive, and then the value is returned. The absolute value function always returns a positive value.

In BASIC, the absolute value function is named ABS.

```
PRINT ABS(5)
```

This example prints the absolute value of 5. Since this value is already positive, the number will not be changed, and the value 5 will be returned.

```
PRINT ABS(−5)
```

Here, the negative 5 is converted to a positive value. The computer prints a 5 on the screen again.

```
PRINT ABS(0)
```

This one is easy. Zero can never be negative. The computer prints 0.

Try a few other numbers of your own until you are confident that you understand the absolute value function. As you will see soon, many other functions and statements cannot accept negative numbers, so you will use the absolute value function to insure that values are positive.

Signum

This function also has to do with the sign of a number. Also called the sign function, signum returns one of three different values. If the argument is 0, the signum function returns a 0. If the argument is greater than 0 (positive), a value of 1 is returned, and arguments less than 0 (negative) cause signum to return a −1. Here are some examples.

```
PRINT SGN(0)
PRINT SGN (6)
PRINT SGN (−4)
```

Although this function is not used as much as many others, one valuable use of signum is to determine an object's direction of movement. Let's say that the variables X1 and X2 record the position of an object on the number line. The variable X1 is the old position, and X2 is the new position. The value SGN(X2−X1) tells which direction the object moved to get from the old position to the new position. If the function returns a 1, the object must have moved to the right. A −1 indicates movement to the left. A value of 0 indicates that the object did not move.

Integer

An integer number is one which has no fractional portion—it has no digits to the right of the decimal point. The numbers 0, 1, 2, 3, and so on, along with their negative counterparts, constitute the integer numbers. The BASIC integer function, named INT, takes a value and returns the nearest integer that is less than or equal to the given value.

```
PRINT INT(5)
```

The number 5 is already an integer, so it is not changed.

```
PRINT INT(4.6)
```

The number 4.6 is not an integer. The integer portion is 4, and the decimal portion is 0.6. The first integer that is less than 4.6 is 4.

It may seem that the integer function merely chops off, or *truncates*, all the digits to the right of the decimal point. But when we apply the function to a negative number that is not an integer, the value returned will be the greatest integer less than the given negative value. This is different from simply removing the decimal portion.

```
PRINT INT(-5)
PRINT INT(-6.4)
```

In the first case, the negative 5 was already an integer, so the computer printed that value unchanged. In the second case, however, the first integer that is less than −6.4 is not −6. On the number line, −6 is greater than −6.4. The correct answer, −7, was printed by the computer.

The integer function will never return a value larger than its argument.

The value returned by a function can be used just like any other number. Therefore, it should be possible to use the value of one function as the argument for another. This is called *nesting* the functions.

```
PRINT ABS(INT(5))
PRINT INT(ABS(6.4))
PRINT ABS(INT(-5))
PRINT INT(ABS(-6.4))
```

Notice that two pairs of parentheses are needed. There must be just as many opening parentheses as closing parentheses, or a SYNTAX ERROR will be generated.

The order in which functions are nested affects the value returned. Functions inside a nest are processed first, then used as arguments for the functions on the outside. In the next example, the order of processing is integer, signum, and then absolute value.

```
PRINT ABS(SGN(INT(-4.6)))
```

One use for the integer function is to determine whether a number is odd or even. An even number can be divided by 2 and leave no remainder, or decimal portion. The number 6, when divided by 2, gives 3 with no remainder. Dividing 7 by 2 gives the noninteger answer 3.5. Examine the following line to see how the integer function is used to remove the integer portion of a number, leaving only the decimal portion.

```
NUMBER=6
PRINT NUMBER/2 – INT(NUMBER/2)
```

If the computer prints a 0, there is no remainder, and NUMBER is even. Otherwise, NUMBER is odd. The next chapter will show you how to make the computer print the words EVEN or ODD, depending on the result.

Another use for the INT function is to take a large number and break it down into smaller parts. The frequency number for middle C is 4389, but that is too large a value to POKE into one location (the maximum value is 255). You can use the integer function to divide the number into the low and high frequency parts, each of which is less than 256.

```
F=4389 : FL=54272 : FH=54273
POKE FL,F–256*INT(F/256) : POKE FH,INT(F/256)
```

A 37 is POKEd into location FL, and a 17 is POKEd into location FH. These are the low and high frequency numbers for middle C, which you would hear playing if all the other POKEs for sound had been executed. To reconstruct the frequency number, multiply 17 by 256 and add 37. The result is 4389.

Random

This function is very different from the others in the way that the argument affects the output value. The random function returns a decimal value between 0 and 1. The value returned, however, is different every time the function is called, even when the same argument is used.

The random function, RND, is used to add an element of chance to a program. It might be used in a game, to determine which player moves first, or in scientific experiments. A study of growth patterns could require a random number to simulate a different reproduction count for every generation. A random number might be used to choose a winning lottery ticket. The random function has a wide variety of applications. To see how it works, have the computer execute the following statement several times.

```
PRINT RND(0)
```

The usefulness of this function is hampered by only one thing—the values it generates are decimal numbers between

0 and 1. It would be much more convenient if the function could generate random integers in a range of something like 0 to 9. This is easily done. Look at the first digit to the right of the decimal point in each of the random numbers printed in the last example. That digit itself is a random number, and it has values ranging from 0 to 9. Multiplying a number by 10 is like moving the decimal point one place to the right. If you multiply the random decimal number by 10, the digit just mentioned appears to the left of the decimal point in the product. By taking the integer of this value, we have our random number from 0 to 9.

```
PRINT INT(RND(0)*10)
```

Repeat this line several times until you are satisfied that it does indeed work.

Perhaps you're in a situation where you don't want 0 to be one of the possible random numbers. Let's make another modification so that the range is all integers from 1 to 9. We can change the statement to read PRINT INT(RND(0)*10)+1, but the range is now 1 to 10 instead of 0 to 9. The lower limit (1) is what you want, but the upper limit (10) is too high. That is easily corrected by changing from multiplication by 10 to multiplication by 9. Now our statement will print random integers from 1 to 9, inclusive.

```
PRINT INT(RND(0)*9)+1
```

The random function can be made to generate random integers in any positive range according to the formula INT(RND(0)*(B−A+1))+A. Replace A with the lower limit of the range, and B with the upper limit. Note that these limits are inclusive. If you want random numbers in the range from 34 to 36, use values of 34 and 36 for A and B, respectively. The statement below will print only the numbers 34, 35, and 36. (The number 3 comes from B−A+1.)

```
PRINT INT(RND(0)*3)+34
```

You may have noticed that we've always used 0 as the argument for the random function. The numbers produced by RND(0) are not completely random and form certain patterns, but they are suitable for use in programs like games. The examples in this book will use only RND(0).

Free Memory

This function is used to determine how much memory is available for use by BASIC. Your Commodore 64 gets its name from the fact that it has 64K of memory. K, a computer term for measuring groups of memory units, is similar in use to the word dozen, which stands for 12 units. One K is used to indicate 1024 units. The computer has 64*1024, or 65536, units of memory. This is why the POKE locations are limited to the range 0 to 65535. (Remember that we count memory units beginning with zero.)

Some of these locations are reserved for the special computer chips that handle graphics, sound, and other operations. More locations are reserved for the system software, including the BASIC language which accounts for 8K of the locations. After all of these reserved areas are provided for, there are about 38K memory locations remaining. This memory is called *free memory,* which means it can be used to hold program information.

As a program gets longer and longer, more of this free memory is used up. The free memory function, FRE, is used to determine just how much memory remains. Type the following line to see how the function works.

```
PRINT FRE(0)
```

If the number printed is negative, a small adjustment must be made to get the correct value.

```
PRINT FRE(0)+65536
```

When you turn on your Commodore 64, a message is displayed which reads something like 38911 BASIC BYTES FREE. A *byte* is another name for one memory unit, so you could say that the computer has 64K bytes of memory, and about 38K bytes which can be used for a BASIC program. These bytes are used for things like program lines and variables. A variable takes up seven bytes of free memory. To verify this, choose a new variable name, one that you haven't yet given a value. Print the amount of free memory, assign a value to the variable for the first time, and print the free memory byte count again.

```
PRINT FRE(0)+65536 : WQ=1 : PRINT FRE(0)+65536
```

The difference in the numbers is seven, the number of bytes needed to hold the name and value of a variable. If you clear the variables, the free memory will be almost equal to the amount displayed by the power-up message.

```
CLR : PRINT FRE(0)+65536
```

The free memory function is different from the previous ones in that the argument has absolutely no effect on the value returned. Such an argument is called a *dummy* argument. You can use any number you want.

If you ever get the OUT OF MEMORY error, see how many bytes are free by using this function. Always remember that when the result is negative, you must add 65536 to the result to get the correct answer.

Screen Memory

Let's take a little time to learn how the computer displays information on the screen. You have seen that there are a lot of characters, and that each character can be displayed in one of 16 colors. The screen consists of 25 rows of 40 columns each, for a total of 1000 positions at which characters can be displayed. That's a lot of information. To keep track of that much information, some of the computer memory has to be used. Therefore, 1000 memory locations have been reserved just for displaying characters. The memory locations are reserved in locations 1024–2023.

You can use these locations to display characters on the screen without typing them in. First press RUN/STOP and RESTORE to reset the screen. The cursor should appear right under the READY prompt. Now, without moving the cursor, enter this line:

```
POKE 1104,81
```

The letter P of the POKE statement should have changed to the ball character.

```
POKE 1105,82
```

This time a horizontal bar appears, covering up the letter 0. The addresses being used in the POKE statements are within the range of 1024 to 2023, which means you are POKEing characters directly onto the screen. Location 1024 corresponds to the upper-left corner of the screen, 1025 to the character immediately to its right, and so on. Forty locations past 1024 is at the left edge of the screen again, one row lower. The number 1104 is 1024 plus 80, which is the leftmost character, two rows down. That's where the letter P was, so that's why it was changed.

Remember that the number after the address in a POKE statement must be an integer from 0 to 255. There is a different character for each number, so you can put a total of 256 different characters on the screen. Experiment a little by POKEing various numbers into location 1106, one at a time. Do not let the screen scroll; keep moving the cursor back up to the same line to change the POKE value.

We still haven't discovered how to set the color for each character. There are another thousand locations, starting at 55296, that are always used only for specifying color. Each location in this *color memory* corresponds to one location in the screen memory. The address in color memory for two rows down is 55296+80 or 55376, so POKEing that location with an 8 will change the color of the ball to orange.

```
POKE 55376,8
```

Change the colors of some more characters by entering the following lines:

```
POKE 55377,4
POKE 55378,3
```

If you experiment a little more, you will notice that if you use POKE to change the color of a screen position which contains the blank character, you will not see the color change. There has to be something displayed at a position in order for the color to show. Also, if you use POKE to change a blank character to something else, the character seems not to appear. This is because the color at that position is blue, the same as the background. (This is not true for older 64s, which will default to white.)

```
POKE 1144,81
```

Nothing should appear. But you will see the ball when you change the background color.

```
POKE 53281,1
```

It is a good practice to always set the color for a position whenever you POKE a new character to the position.

You may have noticed that regardless of the color in which a character is printed, the background color is always the same. All characters have the same background color. The graphics chip inside your Commodore 64 supports another display mode which lets you use one of four background colors with each character. This display mode is called *extended background color mode.* To see how it works, follow this sequence of instructions.

1. Press RUN/STOP and RESTORE.
2. Type the word COMMODORE, but do not press RETURN yet.
3. Press the SHIFT LOCK in.
4. Type COMMODORE again. This time the graphics symbols will appear.
5. Press RETURN.
6. Release the SHIFT LOCK.
7. Press CTRL and RVS ON.
8. Type COMMODORE. It should appear in the reversed characters.
9. Press the SHIFT LOCK in again.
10. Type COMMODORE one last time.
11. Press RETURN.
12. Release the SHIFT LOCK.

Now enter the statement POKE 53265,91. You should see the word COMMODORE displayed four times, each time on a different background color.

The first background color is the one we have been using all along, and can be changed by using POKE with the address 53281. To change the other background colors, enter these lines.

```
POKE 53282,4
POKE 53283,7
POKE 53284,13
```

Just think—a character can be printed in any of 16 colors, and on any of four background colors. That's a lot of color combinations! The only disadvantage to using the extended background color mode is that instead of printing 256 characters, you can now print only 64. But those 64 characters include the alphabet, so this should not be a serious restriction. Take the time now to experiment with this new display mode. When you want to return to the normal mode, just POKE 53265 with the value 27.

The PEEK Function

The POKE statement replaces the value already in a designated location with a new value from 0 to 255. The old value that had been in the location is lost when POKE is used. But sometimes it is useful to know what value is in a location before the POKE statement changes it. This is the purpose of the PEEK function. PEEK is often used with POKE.

The PEEK function works in the same way as the five arithmetic functions introduced. With PEEK, the argument in parentheses is the location number that is to be examined. The function returns whatever value is in the designated location. Thus, PEEK returns integer numbers 0–255, inclusive.

Press RUN/STOP and RESTORE, then type this line:

```
? PEEK(1064)
```

Location 1064 is the first character in the second row on the screen, where the R of the READY prompt is printed. The value returned by the function, 18, represents R, which is the eighteenth letter of the alphabet. Try this next line:

```
? PEEK(1065),PEEK(1066),PEEK(1067),PEEK(1068)
```

The numbers printed—5, 1, 4, and 25—match the numbers that would be POKEd to put the characters E, A, D, and Y on the screen. PEEK can be used to determine the character at any position on the screen.

In a later chapter, we will return to the PEEK function and learn how to use it to examine hardware locations. But hardware locations sometimes behave in unusual ways, and a special trick must be used when PEEKing them in order for the correct value to be returned. For now, let it suffice that PEEK should be used only with memory locations.

Chapter 6

Conditional Logic

Conditional Logic

The True/False Concept

Before we begin programming, we must examine a fundamental concept in computing, conditional logic. Conditional logic is what transforms your Commodore 64 from a calculator into a decision-making machine.

Humans are capable of making complex decisions, involving all sorts of variables. Your computer, on the other hand, is capable of making only the simplest decision. It can decide only whether a statement is *true* or *false*.

This may have you wondering just how you translate a question from English into a form the computer will understand and answer. The computer only understands numbers. How do you ask a question using numbers?

The answer to this dilemma is to pose the question as a true/false mathematical phrase which can be understood by both the computer and the programmer.

Here is an example of such a mathematical phrase: 2+2=4.

When the computer sees a phrase like 2+2=4, first it examines the equal sign. This equal sign does not mean the same thing as the equal sign of the LET statement. In that usage, the equal sign means "be assigned the value," as in LET A=5. In the phrase 2+2=4, no value is being assigned. The equal sign in this phrase is being used in an entirely different way—as a relational operator.

When the computer sees the equal sign, it checks to see if the two values are equal. If they are, the phrase is true. Otherwise, the phrase must be false. Notice that *almost true* is not an option. The computer cannot examine the phrase 2+2=4.0001 and decide that it is close to the correct answer.

You are already familiar with true/false conditional logic. Consider the sentence "If there are some tickets left, we will go see the movie." Seeing the movie depends on the availability of tickets. Examine the phrase "there are some tickets left." If tickets are available, the phrase is true, and you will see the movie. On the other hand, when all the tickets have been sold, if someone said to you, "There are some tickets left," that would be a false statement.

Conditional Logic

The conditional logic in the sentence becomes obvious if you reword it to read, "We will go to the movie on the condition that there are some tickets left."

The computer uses conditional logic by analyzing a mathematical phrase. If the phrase is true, a designated statement is executed. If it is false, the statement is not executed. The next section shows how to implement this logic on the computer.

Summary

- The computer is capable of making very simple decisions, based on whether a phrase is true or false.
- The computer decides things by analyzing conditional mathematical phrases, such as 2+2=4.
- The *condition* in a mathematical phrase is based on a relationship between numbers in the phrase. The relationship being checked is established by a *relational operator.*
- The equal sign can be used as a relational operator. In this use, it is not assigning values to variables.
- Conditional logic is used by the computer to decide whether or not a designated statement should be executed.

The IF-THEN Statement

The syntax for the IF-THEN statement is the keyword IF, a conditional expression, the keyword THEN, and a statement. Notice that this statement contains two keywords, while the previous statements we examined required only one. The conditional expression appears between the two keywords, and is a mathematical phrase like the 2+2=4 shown earlier. The statement after the keyword THEN can be any statement, such as PRINT, POKE, or LET.

The IF-THEN statement can be used in several ways, so let's see it in action before we examine it further.

```
IF 2+2=5 THEN PRINT "COMMODORE 64"
```

Not much action, was there? The condition was false, so the statement after the THEN, in this case a PRINT, was not executed. Notice that a conditional expression having the value *false* does not cause an error to be printed. The computer simply does nothing, and responds with the READY prompt. Now change the condition so that it is true, causing the PRINT statement to be executed.

```
IF 2+3=5 THEN PRINT "COMMODORE 64"
```

The execution of the IF-THEN statement works like this. The first thing the computer sees is the keyword IF. Having found IF, it searches for the keyword THEN. The computer needs these two keywords to delimit the conditional expression which is between them. Once THEN is found, the conditional expression is analyzed to see whether it has a value of *true* or *false*. If the condition is true, the statement to the right of the THEN keyword is immediately executed. If the condition is false, no action is taken.

Of course, the conditional expression 2+3=5 is always true, which defeats the purpose of an IF-THEN statement. Since the condition is always true, the PRINT is always executed, and there is no need for the IF-THEN logic. The conditional expression of an IF-THEN statement usually includes values that can change, such as variables.

The next example shows how conditional logic might be used in a more realistic situation. To begin, we assign some arbitrary value to the variable C, which we will use to mean color.

```
C=9
```

Here is the conditional statement, which you should now type.

```
IF SGN(C)=1 THEN POKE 53280,C
```

The border color was changed, because SGN(9) is 1. Now we assign another arbitrary value to C. This time the value happens to be negative.

```
C=-4
```

Now type the conditional statement again.

```
IF SGN(C)=1 THEN POKE 53280,C
```

The border color was not changed. The idea behind this conditional statement was to execute the POKE to change the border color only if the color value was greater than 0. Remember: Trying to POKE a location with a negative value causes an ILLEGAL QUANTITY error. The IF-THEN statement was designed to prevent that from happening. In the last example, the border color was not changed because the POKE value was negative, and the signum of a negative number is −1, making the condition false. There is one small flaw in the conditional phrase, though. It will not allow the border to be set to color 0, even though 0 is a legal POKE value. The signum of 0 is 0, making the condition false. We can rewrite the conditional statement so that it will POKE the border color to 0 or any positive integer, yet still exclude negative values.

```
IF C=ABS(C) THEN POKE 53280,C
```

The next example simulates the tossing of a coin. The computer will print the word HEAD or TAIL to indicate the result of the toss. The random function is rigged so that it will produce only two different values, 0 and 1, which are used to mean *head* and *tail*, respectively. Clear the screen first. Then, as you type these lines, leave several blank lines between them.

```
FLIP=INT(RND(0)*2)
IF FLIP=0 THEN PRINT "HEAD"
IF FLIP=1 THEN PRINT "TAIL"
```

Because there is a separate conditional statement for each possible value of C, either HEAD or TAIL will always be printed, but it is impossible for both words to be printed in one flip. Move the cursor back to the top, and have the computer execute these statements several times. Remember to erase the leftover word from the previous time.

The coin toss demonstration uses conditional logic to arrive at one of two possible results. In this next example, we use the signum function to select one of *three* choices, involving the direction of an object's movement. Assign values of your own choice to the variables X1 and X2. Variable X1 is the old position of the object, and X2 is the new position. One of the three following conditional statements will be executed, to print the appropriate message.

```
IF SGN(X2-X1)=1 THEN PRINT "MOVING TO THE
    RIGHT"
IF SGN(X2-X1)=-1 THEN PRINT "MOVING TO THE
    LEFT"
IF SGN(X2-X1)=0 THEN PRINT "NOT MOVING"
```

Incidentally, when there are a lot of conditional statements all dependent on the same value, like SGN(X2−X1), you can assign the value to a variable once, and then use the variable in the conditional expression, to save yourself some typing.

```
D=SGN(X2-X1)
```

```
IF D=1 THEN PRINT "MOVING TO THE RIGHT"
IF D=-1 THEN PRINT "MOVING TO THE LEFT"
IF D=0 THEN PRINT "NOT MOVING"
```

In order to make the computer do several things if a condition is true, multiple statements can be placed after the keyword THEN. One use might be to change both the border and background colors simultaneously.

```
IF C=ABC(C) THEN POKE 53280,C : POKE 53281,C
```

If the conditional expression has a value of false, the computer never makes it past the keyword THEN, and the POKE statements are not executed.

You can even combine conditional statements. Before typing this line, assign arbitrary values to the variables A and B.

```
IF A=ABS(A) THEN IF B=ABS(B) THEN POKE 53280,A :
    POKE 53281,B
```

This line is designed to individually set the border and background colors only if both values are legal.

It is a common mistake to omit the second keyword of the IF-THEN statement. This next line will generate a SYNTAX ERROR.

```
IF 2+2=4 PRINT "COMMODORE 64"
```

The IF and THEN keywords work together. When you use one, you must use the other. Just remember: If you have an IF, then you must have a THEN.

The only exception to this IF-THEN rule is when you are using the GOTO statement after the THEN.

We'll discuss more about GOTO in chapter 8.

Summary

- Conditional logic is implemented on the computer by means of the IF-THEN statement.
- The syntax for the IF-THEN statement is the keyword IF, a conditional expression, a second keyword THEN, and a statement.
- The conditional expression can be any mathematical phrase.
- Any statement may be placed after the THEN, even another IF-THEN statement.

- The statement after the THEN is executed if the conditional expression is true. If it is false, nothing happens. All of the statements on the rest of the line are ignored. No error message is printed.
- A missing THEN keyword causes a SYNTAX ERROR (see GOTO exception above).
- The IF-THEN statement controls the execution of statements within a line.

Relational Operators

The decision-making capability of your Commodore 64 would be severely limited if conditions were restricted to analyzing equality alone. To return to our movie ticket analogy, let's change the condition to "If the price of a ticket is under $2, we will go see a movie." It is impossible to write a mathematical phrase equivalent to this condition using only the equal sign.

Thus far we have considered conditions to be true only when something has equaled something else, but it is possible for unequal conditions to be true. To illustrate this, look at the phrase 3=2. The phrase is false, of course. But the comparison needn't stop there. In order for the two values to be unequal, one must be larger than the other. By expanding our relational operators to include cases in which a value is less than or greater than another value, we can make a decision in our latest movie ticket problem.

These two new relational operations, < (less than) and > (greater than) are used in much the same way as the equal sign. The phrase "the price of a ticket is under $2" can be reworded as "the price of a ticket is less than $2." Now assign a value to the variable TP, which we will use to mean "ticket price."

```
TP=1
```

With the new relational operator < (less than), the reworded phrase can be written as a conditional expression.

```
IF TP<2 THEN PRINT "WE WILL SEE THE MOVIE"
```

Now the price goes up to $4.

```
TP=4
```

Execute the conditional statement again, and the message will not be printed.

Even if you have the price of a ticket, the law may prohibit persons 17 years old and under from seeing an R-rated movie unless they are accompanied by an adult. Persons over 17 don't have to worry.

```
AGE=19
IF AGE>17 THEN PRINT "YOU CAN SEE THE MOVIE
  ALONE"
```

We can make our conditional logic even more flexible by combining relational operators. Go back to the example where the condition was true if the ticket price was under $2.

```
IF TP<2 THEN PRINT "WE WILL SEE THE MOVIE"
```

Notice that the condition is false if the price is exactly $2. Let's say that we will see the movie "if the ticket price is $2 or less." Written mathematically, the condition is that the ticket price must be less than or equal to $2.

```
IF TP<=2 THEN PRINT "WE WILL SEE THE MOVIE"
```

In the last example, the PRINT statement will be executed if TP is less than 2 or if it is equal to 2. The order of the operators does not affect the meaning of the condition. They can be combined in the opposite order with the same result.

```
IF TP=<2 THEN PRINT "WE WILL SEE THE MOVIE"
```

Usually the greater-than and less-than signs are put in front of the equal sign, giving the orders <= and >=.

We introduced conditional logic by showing how a condition could be true if one numeric value was equal to another numeric value. Now we reverse that situation and let the value of the conditional expression be true if the values are not equal.

```
IF C<>ABS(C) THEN PRINT "BAD POKE VALUE"
```

If one numeric value is less than another, or greater than the other, the two values are anything but equal. The relation <> means *not equal,* and is often used to cover all remaining possibilities after a check for equality. In the following example, the variable MC stands for "movie count" and tells how many movies will be shown.

```
IF MC=2 THEN PRINT "DOUBLE FEATURE"
IF MC<>2 THEN PRINT "NOT A DOUBLE FEATURE"
```

In a previous example we checked whether a number was even or odd. The number was divided by 2 because all even numbers are evenly divisible by 2, and odd numbers are not. This could be extended to check if a number is a multiple of 3. When a number is divided by 3 and the remainder is 0, the number is a multiple of 3.

A remainder other than 0 means that the number is not a multiple of 3. The problem is that this time there are more than two possible remainders, so you can't check for a remainder of either 0 or 1. The remainder can be 0, 1, or 2 (the result will show a decimal portion of .333333333 or .666666667). Fortunately, both of these values can be handled by using the not-equal relation. Assign an arbitrary value to the variable N.

```
IF N/3=INT(N/3) THEN PRINT "THE NUMBER " N " IS A
    MULTIPLE OF THREE"
IF N/3<>INT(N/3) THEN PRINT "THE NUMBER " N " IS
    NOT A MULTIPLE OF THREE"
```

Here is a list of relational operators:

SYMBOL	MEANING
=	equal
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal

For the symbols < and >, if you have trouble remembering which is which, here is a phrase to help. Just think of the symbol as an alligator's open jaw, and remember that "the alligator always eats the bigger number."

Summary

- Two numbers can be either equal or unequal. When they are unequal, one of them must be larger than the other.
- The relational operators < (less than) and > (greater than) work just like the equal sign, except that they are used in cases of inequality.
- Relational operators can be combined, as in <= and >=. The combination <> means *not equal.*
- The advantage of these new operators is that they can check for conditions which cannot be expressed using only the equal sign. They increase the flexibility of conditional logic.

Logical Operators

So far, we have examined two types of operators. Arithmetic operators are used to do things like add and multiply numbers and variables. The computer uses relational operators to evaluate conditional expressions. We now introduce the third type, the logical operators AND, OR, and NOT, which further expand the computer's ability to evaluate conditional expressions.

The first two logical operators, AND and OR, play a major role in conditional logic and are the main topic of this section. The third operator, NOT, is seldom used and is dealt with last.

In our movie ticket examples, we identified two conditions for tickets that determined whether you would go see a movie. In one example, you would go to the movie if tickets were available. In the second example, we assumed that tickets were available, but you would go only if the ticket price was less than $2.

By using the logical operators, you can evaluate two or more conditions in a single IF-THEN statement. The AND operator requires that both conditions be true. This time, tickets must be available for not more than $2, or you won't go to the movies.

Here is an example of the AND operator. The variable TA represents the number of tickets available, and TP represents the price of each ticket.

```
IF TA>0 AND TP<=2 THEN PRINT "MOVIES, HERE WE
    COME!"
```

The PRINT statement will be executed only if TA is greater than 0 and TP is less than or equal to 2 as well.

The AND operator is often used to determine if a number is in a specified range. The following example changes the border color only if the POKE value is from 0 to 15, the range for 16 different colors.

```
IF C>=0 AND C<=15 THEN POKE 53280,C
```

You can link a lot of conditions together using multiple AND operators. Let's modify the last example so it also checks that the POKE value is an integer.

```
IF C>=0 AND C<=15 AND C=INT(C) THEN POKE
    53280,C
```

Conditional Logic

Here is a detailed look at how the AND operator works. The AND is placed between two conditional expressions. The conditional expressions, plus the AND operator, are treated as one conditional expression. The true/false value of the big expression depends on the values of the two smaller conditional expressions. This *truth table* shows all possible outcomes:

First Condition	**Second Condition**	**Result**
True	True	True
True	False	False
False	True	False
False	False	False

When the AND operator is used, all of the smaller conditions must be true in order for the final value to be true. Conversely, the big expression is false if just one of the smaller conditional expressions is false. This also applies to cases when several conditional expressions are linked by several AND operators.

The logical operator OR works differently.

```
IF C=5 OR C=13 THEN PRINT "THE COLOR IS GREEN"
```

With the OR operator, all it takes is for one of the conditional expressions to be true, and the whole expression is true. Here's the truth table for the OR operator.

First Condition	**Second Condition**	**Result**
True	True	True
True	False	True
False	True	True
False	False	False

Like AND, the OR operator can be used to link several conditional expressions.

```
IF C=11 OR C=12 OR C=15 THEN PRINT "GRAY TONE"
```

The operators AND or OR can be mixed in the same expression, although the logic may get to be a little confusing. The following statement allows only the colors cyan, purple, green, and brown to be accepted as POKE values for changing the border color.

```
IF C>=3 AND C<=5 OR C=9 THEN POKE 53280,C
```

The last operator to be introduced is the rarely used NOT operator. Unlike AND and OR, which are binary operators (they operate on two values), NOT is a unary operator (it operates on only one value). This operator changes a value to the opposite value. It causes a true value to become false and a false value to become true. Whereas the PRINT statement will be executed in the first example, it won't be executed in the second one:

```
IF 2+2=4 THEN PRINT "COMMODORE 64"
IF NOT 2+2=4 THEN PRINT "COMMODORE 64"
```

This can be used in cases like

```
IF NOT C=ABS(C) THEN PRINT "BAD POKE VALUE"
```

However, we have already seen the preceding example written without the NOT operator.

```
IF C<>ABS(C) THEN PRINT "BAD POKE VALUE"
```

Because everything that the NOT operator does can be accomplished by rewriting a conditional expression, NOT is the least frequently used logical operator. You will rarely see this operator except in advanced applications, which are beyond the scope of this book.

One other common mistake is shown in the line below.

```
IF C=5 OR 13 THEN PRINT "GREEN"
```

This one is tricky because everything sounds okay if you read the line out loud. But according to IF-THEN syntax, the 13 should be replaced with C=13 in order to have the statement print GREEN if C equals either 5 or 13. As the line is written, OR is performing an operation on two numbers (an advanced application discussed in the next section).

Always keep the order of precedence in mind when using the various operators. Generally, arithmetic operators have precedence over relational operators, which have precedence over logical operators. Here is a list showing the order of precedence for all operators in the BASIC language. Remember that parentheses can be used to change the order of evaluation.

Operators	Names
↑	exponentiation
—	negation
*,/	multiplication and division

+,−	addition and subtraction
>,=,<	relational
NOT	logical
AND	logical
OR	logical

Summary

- Unlike arithmetic and relational operators, which operate on numbers, the logical operators AND, OR, and NOT operate on the values true and false.
- The operators AND or OR are called binary operators because they operate on two values. When you place AND or OR between two conditional expressions, the operator and expressions form one big expression.
- When using AND, the value of this big expression is true only when both of the smaller expressions are true. All it takes is one false value to make the whole expression false.
- When using OR, the value of the big expression is false only when both of the smaller expressions are false. All it takes is one true value to make the whole expression true.
- The NOT operator is called a unary operator because it operates on only one value. This operator takes a true/false value and changes it to the opposite value.
- Logical operators have the lowest precedence of all the operators.

Binary Numbers

In order to use the graphics features of the Commodore 64, you should have an understanding of the binary number system. This section presents the information you will need to perform high-resolution plotting and sprite animation. It will also give you some insight into why conditional logic is based on the values true and false. This section is not intended to be a full treatment of binary numbers; it contains background information. Also, rest assured that you can still use the rest of the book even if you don't understand this section.

No matter what numbering system is used, the quantities represented by numbers always stay the same. For example, the number 4, which we write as 4, was written IV by the Romans.

The binary number system is simply a different way of representing numbers. One difference is that there are only two digits in a binary number, 0 and 1 (instead of the ten we are used to, 0–9, in the decimal system).

Binary	Decimal
0	0
1	1

There is no numeral 2 in binary numbers, just as there is no single numeral for 10 in decimal numbers. To express the value 10 in decimal, a second digit must be used.

```
 8
 9
10
```

When the second digit is added, the first one starts back at 0, and starts building up to 9 again.

```
11
12
```

The same trick is used with binary numbers, except that another digit must be added after every use of the numeral 1, not 9.

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Note that the symbol 10, which means ten in the decimal system, is 2 in the binary system.

Conditional Logic

Each digit in a binary number is called a *bit*. If a bit has the value 0, it is *clear*. If the bit has a value of 1, it is *set*. The largest number that can be represented using only four bits is 15.

Eight bits form a *byte*. Here are some examples of bytes. It is a common practice to show the full eight bits, even though the number may be small enough that not all of them are needed.

Binary	**Decimal**
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
...	
00001111	15
...	
01111111	127
10000000	128
10000001	129
10000010	130
...	
11111110	254
11111111	255

A byte can have any value from 0 to 255. Hardware and memory locations consist of eight bits, which is why attempting to POKE a value outside the range 0 to 255 produces the ILLEGAL QUANTITY error.

For convenience, the eight columns of a byte are numbered, from 0 to 7. Bit 0 is the rightmost bit, and bit 7 is the leftmost bit. In the number 00100001, bits 0 and 5 are set.

76543210	Bit Numbers
00100001	Byte

Those bytes which have a bit set in only one column require special attention.

Binary	**Decimal**
00000001	1
00000010	2
00000100	4
00001000	8

00010000	16
00100000	32
01000000	64
10000000	128

Although each bit by itself can be either 0 or 1, it can have a much larger value within the context of a byte. Bit 7 corresponds to the value 128. This provides an easy way to convert a binary number to decimal, by adding the values for the bits which are set. The example shows how to find the decimal equivalent for the number 01100101.

```
128  64  32  16   8   4   2    1   Value
 0    1   1   0   0   1   0    1   Byte
____________________________________
   64+32              4  +     1   =101 Decimal
```

Sometimes different things are controlled by the various bits in a hardware location. Location 53265 controls, among other things, the extended background color mode. Normally this location contains the value 27, or 00011011. To invoke the extended background color mode, we POKE 53265 with a 91, or 01011011. The extended mode is in effect only when bit 6 is set. Bit 6 corresponds to the number 64, so 64 was added to 27 to get the POKE value, 91.

The other bits in location 53265 control other features. One problem in POKEing the value 91 is that the other bits in location 53265 may change, so 53265 will not always contain a 27. One solution would be to PEEK 53265, add the number 64, and POKE the result back into 53265.

```
POKE 53265,PEEK(53265)+64
```

Unfortunately, this method also has a drawback. What if the statement is executed when bit 6 is already set? The value at 53265 will already include the 64, so adding another 64 would bring the total to 128, which corresponds with bit 7. You should avoid this method, because it can cause you to accidentally set the wrong bit.

The solution to our dilemma lies with the operators AND, OR, and NOT. These are logical operators when used with true and false values, and they can also be used on bytes. These three operators are used to set and clear the individual bits in a byte.

The OR operator works on bits in much the same way that it works on true and false values. Just think of 0 as corresponding to false, and 1 as corresponding to true. When you OR two bits, the result will be set if either of the two bits is set.

```
   1        1        0        0
OR 1     OR 0     OR 1     OR 0
----     ----     ----     ----
   1        1        1        0
```

When you OR one byte with another, the operation is done for each bit column.

```
     Binary      Decimal
   00011011           27
OR 01000000        OR 64
-----------        -----
   01011011           91
```

Because 1 OR 1 is 1, it does not matter if bit 6 is already set when you OR the value in location 53265 with 64.

```
     Binary      Decimal
   01011011           91
OR 01000000        OR 64
-----------        -----
   01011011           91
```

The best way to invoke the extended background color mode is to use the following statement:

```
POKE 53265,PEEK(53265) OR 64
```

We have seen that the OR operation can be used to set a bit in a byte without affecting the other bits. To clear a bit, use the AND operation. With AND, the result will be set only if both bits are set.

```
    1         1         0         0
AND 1     AND 0     AND 1     AND 0
-----     -----     -----     -----
    1         0         0         0
```

Using AND to turn off the extended background color mode is not as easy as turning it on, however. Look what happens if you AND the contents of location 53265 with 64.

```
         Binary            Decimal
       01011011                 91
   AND 01000000             AND 64
   ------------              -----
       01000000                 64
```

Instead of turning off the extended mode, we turned off everything else! Bit 6 is the only one which survives, because it is the only one which is set in the second byte. All the other bits are forced to 0. Although this does not turn off the extended mode, it does provide a way for a program to check if the mode is on.

```
IF (PEEK(53265) AND 64) = 64 THEN PRINT "EXTENDED
    MODE IS ON"
IF (PEEK(53265) AND 64) = 0 THEN PRINT "EXTENDED
    MODE IS OFF"
```

To turn off the extended mode, a little trick must be used. The secret is to AND the value in location 53265 with a byte which has every bit set except bit 6.

```
         Binary            Decimal
       01011011                 91
   AND 10111111            AND 191
   ------------              -----
       00011011                 27
```

The second byte is called a *mask,* because it lets only certain bits show through. All bits other than bit 6 retain their value; bit 6 is forced to 0. The best way to turn off the extended background color mode is to use the following statement.

```
POKE 53265,PEEK(53265) AND 191
```

The Commodore 64 supports 16 colors, so four bits are needed to represent a color number. The locations for the border and background colors use only bits 0 through 3. Bits 4 through 7 are always set, which is why you cannot use the PEEK function to determine the color number. If you PEEK location 53281 when the background is dark blue, you get the value 246 (11110110), when it should be 6 (00000110). This will happen even if you previously POKEd 53281 with a 6. Those unused four bits are getting in the way, so let's get rid of them.

```
        Binary          Decimal
      11110110              246
  AND 00001111           AND 15
  ------------            -----
      00000110                6
```

Here is the BASIC statement that should be used to print the color number.

```
PRINT PEEK(53281) AND 15
```

The operations AND and OR work on every bit, so a mask can have more than one bit set. This masking technique should also be used when PEEKing color memory, which starts at 55296.

Finally, the NOT operator can be used with numbers, but it is not used as often as AND and OR. Being a unary operator, NOT requires only one operand.

```
NOT 0        NOT 1        NOT 01000110
-----        -----        ------------
    1            0            10111001
```

The NOT operator flips every bit in the number. Every bit that was set is cleared, and every bit that was clear is set.

Summary

- Binary numbers consist only of the digits 0 and 1.
- The binary system does not allow the number 2 to be represented in a single digit, just as the decimal system does not allow the value 10 to be represented by a single digit. To write larger values, multiple digits must be used.
- A *bit* is a binary digit. A bit is *clear* if it is 0, and *set* if it is 1.
- A *byte* is a number made up of eight bits. The values 0 through 255 can be represented by one byte. It is impossible for a byte to represent values outside this range.
- The eight bits in a byte are numbered from 0 to 7, with the rightmost bit being numbered 0.
- By itself, a bit can only be 0 or 1. In relation to the other bits in a byte, however, a bit can have much larger values.
- It is an easy matter to convert a binary number to decimal. Given the binary number, look at only those bits which are set. Add the values that correspond to each of these bits. The sum is the decimal equivalent of the binary number.

- The operators AND, OR, and NOT are used with numbers to set and clear individual bits in a byte.
- When using these operators on numbers, the value *false* is replaced by the value 0, and the value *true* is replaced by the value 1.
- To set a bit in a byte without affecting the other bits, use the OR operator. OR sets the bit in the result if either of the bits in the two numbers is set.
- The AND operator sets the bit in the result only if both bits in the two numbers are set.
- To clear one or more bits in a byte, AND the byte with another byte which has all bits set except those which are to be cleared. Such a byte is called a *mask.*
- Whenever you PEEK a hardware color or color memory location, AND the value with 15 to eliminate the unused bits. This applies only to color locations.
- The NOT operator takes one number and flips each bit to the opposite value.

Using the Joystick

The joystick has four cardinal directions, four diagonal directions, a center position, and a trigger. A hardware location keeps track of which direction the stick is pointing to, and whether the trigger is being pressed. Port 2 is easier to use in BASIC than port 1, so we will always assume that the joystick is plugged into port 2. The hardware location for port 2 is 56320.

Bits 0 through 3 contain the direction information. The other bits in the location may be set, so eliminate them using the AND operator. Enter the following line several times while pushing the stick in different directions.

```
PRINT PEEK(56320) AND 15
```

Here is a table showing the numbers for each direction.

Binary	Decimal	Direction
1110	14	up
1101	13	down
1011	11	left
0111	7	right
1010	10	up left
0110	6	up right

1001	9	down left
0101	5	down right
1111	15	no direction (center)

To find out if the stick is being pushed in a specific direction, such as left, use a line like the one below.

```
IF (PEEK(56320) AND 15) = 11 THEN PRINT "STICK
PUSHED LEFT"
```

If you examine the bit patterns, you will notice that each bit corresponds to a main direction.

BIT	VALUE	DIRECTION
0	1	up
1	2	down
2	4	left
3	8	right

Normally, when the stick is not being pushed in any direction, all four bits are set. A bit is cleared only when the stick is pushed in that direction. Diagonals are obtained by pushing in both a horizontal and a vertical direction. To check if the stick is being pushed left, use this line.

```
IF (PEEK(56320) AND 4) = 0 THEN PRINT "STICK PUSHED
    LEFT"
```

The trigger information is kept in bit 4, which has the value 16. This bit is clear if the trigger is being pressed, and set if it is released. The following line will print a message only when the trigger is pressed.

```
IF (PEEK(56320) AND 16) = 0 THEN PRINT "TRIGGER IS
    PRESSED"
```

After we have covered a few more of the basics, you will write a game program that shows how the joystick can be used to make objects move on the screen.

Chapter 7

The BASIC Program

The BASIC Program

Immediate Versus Deferred Mode

This is it—the chapter you've been waiting for! In the next few pages you will discover what a program is, how it works, how to create and edit one, how to execute it, and how to store it on cassette or disk for later retrieval.

Up until now, you have typed in a new line every time you wanted to make the computer do something. As soon as you pressed the RETURN key, the computer executed the line and printed the READY prompt. This is called the *immediate* mode, or *direct* mode, of the computer; the line is executed immediately after you type it and press RETURN.

To make the computer rapidly execute several lines in a sequence, you had to clear the screen, type each line in its proper place on the screen, move the cursor back to the top line, and repeatedly press the RETURN key.

This last method of executing lines is very similar to writing and executing a program. After all, that's basically what a program is—a sequence of lines containing one or more BASIC statements which are executed in a certain order. But there are some important differences.

In the immediate mode, every time you want to execute the sequence, you have to position the cursor again, and press all those RETURNs again. And if you clear the screen, everything you typed is lost. However, a program is not executed as you type it in. It is put into a special place in the computer's memory, where it resides until you instruct the computer to execute it. Clearing the screen does not erase the program from its place in memory. You can call it back to the screen as many times as you want without retyping it.

This means that you can add to it and edit it until you are satisfied, and then tell the computer to execute it. While the computer is executing a program, it is said to be operating in the *deferred* mode, also called the *program* mode.

When you type a line, you must let the computer know whether to execute it immediately, or place it in memory as a program line. And once you have specified that a line is to be included in the program, you must tell the computer where to insert the line. Is it the first line, or third, or last?

Both these tasks are accomplished by the same thing—a line number. If you type a line that starts with a number, it will be placed in memory as one of the lines of a program. The line number determines the position of each line in the program.

The next section gives more information about line numbers, shows you how to build a program, and introduces three commands to help you manage the program.

Summary

- When a line is executed as soon as it is typed, the computer is operating in the *immediate* or *direct* mode.
- When the computer executes a program, it is said to be operating in the *deferred* or *program* mode.
- If a number is placed at the beginning of a line, the computer will not execute it immediately, but instead will place it in computer memory as a program line. A program is a sequence of numbered lines.
- The position of a line in a program is determined by the value of the number at the beginning of the line.

Writing a Program

Turn your Commodore 64 off and back on, and enter the following line:

```
10 ? "COMMODORE 64"
```

Nothing happened. Not even a READY prompt was displayed. The cursor moved to the next line when you pressed the RETURN key, and that was it. The line was not executed because it was placed in memory as a program line. To see that it is now part of a program, type this command.

```
LIST
```

The computer printed the line that you had typed before, including the line number. The line number 10 does not mean that this line is the tenth one in the program. Right now, line 10 is the first, and in fact the only, line in the program.

Now add another line to the program.

```
20 ? "I AM A FRIENDLY COMPUTER"
```

Follow this with another LIST command.

```
LIST
```

The computer lists both program lines on the screen, printing line 10 first, and then line 20. The computer did not print any blank lines for the missing line numbers, such as 11, 12, and so on. The computer arranges lines in the same way you arrange a list of names in alphabetical order. When you alphabetize Adam, Carrie, and Zelda, you don't leave blank spaces for missing names or letters.

You instruct the computer to execute this short program by typing the RUN command.

```
RUN
```

Both program lines were executed, in the correct order, and then the computer printed the READY prompt. Do it again.

```
RUN
```

This is one of the advantages to using a program—you type the lines only once, but you can execute them again and again.

Although they look like statements, LIST and RUN are classified as commands. Commands are used in the immediate mode to manage a program, and are not usually included in a program. The command LIST instructs the computer to print on the screen every line that is currently in the program. Lines are listed by order of increasing line numbers. The command RUN instructs the computer to execute the program, starting at the first line. While a program is in the process of being executed, it is said to be *running*. Execution ends when the last line has been executed, after which the READY prompt is printed.

We are going to add a third line to the program, but this time we want it to be placed between the first two. This is accomplished by using a line number between 10 and 20. Enter the following line, then LIST and RUN the program.

```
15 POKE 53280, INT(RND(0)*16)
```

Now you see why we didn't use the line numbers 1 and 2. It would have been impossible to put a line between them, because line numbers have to be integers. The number 1.5 is not a legal line.

Line numbers are used only when you enter a line in the program, to indicate where in the program the line should be placed. They are ignored while the program is being executed. Therefore, to make editing easier, many people choose to start line numbering at 100 and increment by 10. This leaves plenty of room to insert lines at a later time. Line numbers can range from 0 to 63999. The only problem with using larger numbers is that more digits require more typing. In this book, the longer examples will start at 100, but the shorter ones will start at 10 for your typing convenience.

Program lines are always placed in memory exactly as you type them, including spacing, with two exceptions. If you type a line and use the question mark in place of PRINT, the question mark will be converted to PRINT before the line is placed in memory. This shows up when you use LIST. Lines 10 and 20 of the current program were first typed using the question mark, but the keyword PRINT is displayed when the lines are listed. The second exception is that extra spaces placed between the line number and the first statement are ignored. These are called *leading spaces.* Only one leading space is printed when the line is listed.

After you enter a line, you may need to correct a mistake or add a new statement. You can change a line by retyping the line, using the same line number and the new statements. The next example changes line 10 so that the digits "64" are spelled in the long fashion.

```
10 ?"COMMODORE SIXTY-FOUR"
```

Use LIST and RUN to confirm that the line has been changed. The old line 10 no longer exists, having been replaced by the new line 10.

Note that although there does not have to be a line for every line number, no two lines can have the same line number.

By using the screen editor and cursor control keys, you can make changes without retyping the whole line. LIST the program and then move the cursor to the line that is to be changed. Use the editing keys to INSerT or DELete some characters in the line. When you are done, press the RETURN key while the cursor is still on the changed line. The computer stores the line in memory, just as if you had typed the whole line.

Give this a try by using the editing keys to change line 10 back to its original state. Position the cursor on top of the S in SIXTY, type the digits 6 and 4, and then hold down on the SPACE bar until all the other characters have been erased. Press the RETURN key. The changes will not be made unless you press this key. When you LIST the program, it will show line 10 restored to its original state.

Now let's say that we want to get rid of line 10 altogether. To delete a line, move the cursor to a blank line, type the number of the line you want to delete, and press RETURN. Don't type anything after the line number; just press RETURN.

```
10
```

When you use LIST, you will see that line 10 is no longer part of the program. If, however, a listing of the line is still somewhere on the screen, all you have to do is move the cursor to that line and press RETURN. The computer thinks that you just typed the whole line, and it will again be a part of the program, which can be verified by using LIST. Please note that a line may be displayed on the screen even though it is not part of the program.

There will be times when you want to move a line from one place in the program to another. To do this, you can delete the line (type the old line number and press RETURN) and then enter the same line, but using the new line number. Let's move line 15 of our current program to line 30. Enter the following lines

```
15
30 POKE 53280,INT(RND(0)*16)
```

Confirm that the change was made by listing the program.

There is a better way to move a line from one place to another in a program: using the screen editor keys. We'll do this to move line 20 to line 45. First list the program. Next, move the cursor so that it is on top of the 2 in line 20. All you have to do now is type the number 45 and press RETURN. List the program again. You will see the new line numbered 45, but you will also see the old line 20. By typing the line number 45 on top of the 20, all we did was add a new line, numbered 45, to the program.

To delete a line, you have to type the line number with nothing after it, and then press RETURN. Therefore, even when moving a program line using the editing keys, a two-step process is necessary. First you copy the old line to a new line, then you delete the old line.

Here is one other word of caution about editing lines. Whether you are changing a line, or moving it to a new place, you must remember to press RETURN while the cursor is still on that line; otherwise, the changes will not be made.

Our example program has been thoroughly garbled by all our experiments, so let's write a new program. Before we can do that, the old program has to be erased. This could be done by deleting each line in the program, but that's a slow process, especially if the program contains a lot of lines. Fortunately, there is a faster way to erase a program.

Just as the statement CLR erases all variables, the command NEW deletes every line in a program. LIST the current program, type NEW, and then LIST the program again, to get a before and after picture of what the command NEW does.

```
LIST
NEW
LIST
```

The entire program is gone. As you can see, NEW is a rather destructive command, and you should exercise caution when using it.

Now you are all set to enter this next program:

```
10 PRINT N
20 N=64
30 PRINT N
```

RUN the program. The computer should print a 0, then a 64, then the READY prompt when it is done executing the program. Variables still retain their values after a program has ended. Use PRINT in the immediate mode to print the value for N.

```
PRINT N
```

The variable N still has the value 64. RUN the program a second time. The variable N started out with the value zero again. This is because the RUN command makes the computer first do a CLR before starting program execution. An automatic CLR is also performed every time you modify the program. Right now the variable N has the value 64. If you delete line

10 and then PRINT the value of N, you will see that N has been cleared.

```
PRINT N
10
PRINT N
```

The NEW command also clears all variables when it erases a program.

Add the following line to the current program, typing the line exactly as it's printed here.

```
40 IF N=64 PRINT "COMMODORE 64"
```

This time when you run the program, a SYNTAX ERROR is generated. The error message also says IN 40, so you know in which line the error occurred. Examine line 40. You will notice that there is a missing THEN keyword. The point of this demonstration is that errors in a program line are not detected when the line is entered. Errors are detected only when a line is executed. A program will stop executing as soon as an error occurs. Fix line 40 by inserting the THEN keyword in the appropriate place. Then add the following lines to the program:

```
40 IF N=64 THEN PRINT "COMMODORE 64"
10 PRINT "COLOR GRAPHICS"
50 PRINT "SOUND SYNTHESIS"
```

Let's conclude this section by looking at the syntax for the three new commands LIST, RUN, and NEW.

The LIST command causes every line in the program to be printed in order on the screen. The printing scrolls up the screen so quickly that you can't read a program while it is being listed. Sometimes, when you are listing a long program, it would be nice to slow down the listing, so you can look for a specific thing, such as a variable name. By holding down the CTRL key while program lines are being listed, you can cause a pause after each line is printed. This still isn't slow enough to read every line, but it is handy for scanning program lines. Try it with the current program. Remember that this is most useful for longer programs with more lines than will fit on the screen at once.

Another problem with longer programs is that sometimes you may only want to look at one line near the end of the program. When you use LIST, you have to wait for all the earlier lines to be listed first. To avoid this, you can use LIST to display specific lines in a program. Try it to look at line 20.

```
LIST 20
```

Or you can request that a range of line numbers be listed.

```
LIST 20-30
```

Use the minus sign as a dash between the first and last line numbers to be listed. All lines in the specified range will be printed on the screen.

For even more flexibility, you can use the dash with only one line number. If you type a line number and then a dash, the requested line will be listed, along with all lines that come after it, to the end of the program. If you type a dash and then a line number, the computer will print all lines up to and including the requested line. This is especially helpful when you don't remember the line numbers for the first or last lines in the program.

```
LIST 20-
LIST -30
```

Because of the way Commodore 64 BASIC handles line numbers, if you try to list only line 0, the computer will list every line in the program, as if you had typed just plain LIST.

```
LIST 0
```

The RUN command clears all variables, and then executes the program starting at the first line. As with LIST, you can put a line number after the command RUN, to start execution at a line other than the first. You cannot, however, use a line number range. Once a program has started execution, it will stop when it reaches the end of the program, or when an error occurs. Type the following line to see how RUN works with a line number:

```
RUN 40
```

The NEW command clears all variables and erases the current program. If you print the free memory after a NEW, the byte count will almost match the byte count printed when you first turned the computer on. No line numbers can be used with this command.

Summary

- When a program line is placed in memory for deferred execution, the cursor moves to the next line, but no READY prompt is printed.
- The LIST command is used to print every line currently in the program.
- Line numbers must be integers in the range from 0 to 63999. Any other numbers cause the SYNTAX ERROR to be printed.
- The line number is used only to indicate the position in the program in which a line is to be placed. A line numbered 10 is not necessarily the tenth line in a program.
- The computer ignores unused line numbers.
- Using a large line number increment makes it easier to insert lines between the existing ones in a program. It is a good practice to increment by at least 10.
- The RUN command is used to execute a program. Execution starts at the first line and ends when there are no more lines.
- While a program is being executed, it is said to be *running*.
- One of the advantages of a program is that lines typed once can be executed several times.
- Commands are used to help manage a program. They are used in the immediate mode, but are not usually included in a program.
- A program line is placed in memory exactly as it was typed, except when abbreviations or leading spaces are used. The question mark is converted to the keyword PRINT before the line is stored in memory. Only one leading space is put after the line number, no matter how many were typed when the line was entered.
- A program line can be changed by typing the same line number and new statements. The RETURN key must be pressed to enter the line.
- A program line also can be changed by using the editing keys to move the cursor to a listing of the line to be changed so that the INSerT and DELete keys may be used to modify the line. After the changes have been made, the RETURN key must be pressed while the cursor is still on the line.

- A line is deleted by typing only the line number and pressing RETURN.
- To retrieve a line that has been deleted, the line must be retyped unless a listing of the line is still on the screen. In that case, the cursor can be moved on top of the line, and the RETURN key pressed. The line will again be part of the program.
- Even though a line may appear on the screen, it is not necessarily a part of the program.
- To move a line from one place in the program to another, type the line using the new line number, and delete the old line.
- A line can also be moved by using the editing keys. List the line, position the cursor on top of the line number, type the new line number, and press RETURN. You must still delete the old line, though.
- A fast way to erase a program is to use the NEW command, which deletes every line in the program, and clears all variables at the same time.
- Variables retain their values after a program has stopped executing.
- Variables are cleared as soon as the program is modified in any way.
- The RUN command also automatically clears all variables before program execution starts.
- Errors in a program line are detected only when the line is executed, not when the line is entered and placed in memory.
- If an error occurs while a program is running, execution stops immediately. The error message is printed, along with an indication of which line contains the error.
- Holding down the CTRL key while a program is listing will cause the lines to scroll more slowly.
- The LIST command can be used to list a specific line, by typing the line number after the keyword.
- A line number range can be specified after LIST. A dash must be placed between the starting and ending line numbers.
- Using LIST with a line number followed by a dash causes the listing to start at the line number and continue to the

end of the program. Using LIST with a dash followed by a line number causes the listing to start at the first line in the program and continue to the specified line number.

- The RUN command usually starts executing a program at the first line. To start execution at a line other than the first, that line number must be specified after the keyword RUN. A line number range may not be used with this command.

Program Storage and Retrieval

Programs can get to be very long. While it is convenient to be able to execute the whole program just by typing RUN, it is not so convenient if you have to type in a program every time you want to use it. As soon as you type NEW or turn the computer off, the program is gone for good. Since you will often want to change from one program to another, or turn the computer off, it is important that you be able to store programs on floppy disk or cassette tape. These are two examples of *storage media* used to keep permanent copies of programs. Once a program has been stored on tape or disk, you can turn the computer off or use a different program, and retrieve your stored program anytime you want to use it.

The two commands used to store and retrieve programs are SAVE and LOAD.

First we will see how to SAVE a program to tape or disk. Type NEW, enter the following program, and RUN it so you are satisfied that it works correctly. Then follow one of the procedures described below, depending on whether you are using tape or disk.

Poem

Please read the article called "Automatic Proofreader," Appendix I, before typing in any of the programs in this book. Do not type in the rem statement at the end of the program lines. For example, do not type :rem 240 from line 10.

```
10 PRINT "SILVER BOX"                          :rem 240
15 PRINT "BY EDWARD CHU" : PRINT                :rem 48
20 PRINT "A SMALL SILVER BOX"                  :rem 171
30 PRINT "WITH MOUTHS FLAMING RED,"             :rem 85
40 PRINT "SITS ON MY SHELF"                     :rem 45
50 PRINT "DEVOURING BREAD." : PRINT             :rem 60
60 PRINT "I FED HIM SOME SLICES"                :rem 36
70 PRINT "TWO AT A TIME,"                       :rem 99
80 PRINT "AND HE HANDS THEM BACK."             :rem 116
90 PRINT "BLACK."                              :rem 197
```

If you are storing programs on cassette tape, here is the procedure to use:

1. Rewind the tape to the beginning by pressing the REWIND key on the Datassette. Press STOP when the tape is rewound completely.
2. Type the command SAVE "POEM" and press RETURN.
3. The computer will respond with the message PRESS RECORD & PLAY ON TAPE. This is your cue to simultaneously press the RECORD and PLAY keys on your Datassette. On some models of the Datassette, the PLAY key will automatically be pressed when you press RECORD.
4. As soon as you press RECORD and PLAY on the Datassette, the computer will print the message OK, followed by the message SAVING POEM. You will not be able to see these messages, however, because the screen background color will be changed to match the border color. This is called *screen blanking*.
5. The tape will start to move. During this time the computer is taking a copy of the program in memory and storing it on the tape.
6. When the computer is done, it will print the READY prompt, and restore the screen.
7. The tape will have stopped moving, and you can press STOP on the Datassette.

Here is the equivalent procedure for using a disk drive.

1. Insert a properly formatted disk into the disk drive. Read the disk drive user's manual for instructions on how to format a disk.
2. Type the command SAVE "POEM",8 and press RETURN. The number 8 is called a *device number*. If you do not type the device number, the computer will try to save the program on tape.
3. The computer will print the message SAVING POEM.
4. The disk drive will be activated, and the computer will take a copy of the program currently in memory and store it on the disk.
5. Upon completion of the SAVE operation, the computer will print the READY prompt, and the disk drive will automatically stop.

Remember that the SAVE command only stores a copy of the program on the tape or disk. The program is still in memory after you use the SAVE command, as can be verified by entering the command LIST.

Now you have a copy of the program safely stored on tape or disk for future reference. At this point you are free to type NEW, or even turn the computer off. If you are using disk, be sure to remove your disk from the drive before you turn your computer off.

To retrieve the program, use the LOAD command. Again, this procedure differs slightly for tape and disk.

If you are using cassette tape, follow these steps:

1. Rewind the tape back to the beginning.

2. Type the command LOAD "POEM" and press RETURN.

3. The computer will respond with the message PRESS PLAY ON TAPE. Be sure to press only the PLAY button this time. Do *not* press RECORD.

The computer will then print the messages OK and SEARCHING FOR POEM on the screen, but you won't be able to see them because once again the screen will be blanked.

5. The tape will start moving as the computer searches for the program.

6. When the computer finds the program, it will stop the tape, print the message FOUND POEM, and restore the screen so you can see the message.

7. After about ten seconds, the computer will print the message LOADING. (On some 64s, you must first press the key labeled with the Commodore logo.) The screen will be blanked again, and the tape will start to move, as the computer copies the program from tape into memory.

8. When the program is done loading, the READY prompt will be printed, the screen will be restored, and the tape will stop moving. You may want to press the STOP key on the Datassette.

Here are the steps for loading a program on disk.

1. Type the command LOAD "POEM",8 and press RETURN.

2. The computer will print the message SEARCHING FOR POEM.

3. The disk drive will be activated. When the program has

been found, the computer will print LOADING.

4. The disk drive will stop, the READY prompt will be printed, and the program will be in memory.

To see that the program is indeed back in memory, use the commands LIST or RUN. Every line of the program should be back, exactly as it was saved. Once a program has been saved, it can be loaded again and again.

There is one other thing you should be aware of concerning the LOAD command. When you load a program from tape or disk, it replaces the one currently in memory. Enter the following lines.

```
NEW
18 PRINT "USE A MICROWAVE INSTEAD"
```

Now continue with the appropriate loading procedure. Once the program has been loaded, you will see that line 18 no longer exists. Always be careful that you do not inadvertently erase a program in memory by loading a new one.

This brings up another important topic: storing multiple programs on tape or disk. To store a second program on a tape, start the tape at the position where it stopped when you saved the first program. Type the SAVE command again, but this time use a different filename. When programs are stored on tape or disk, they are called *files*. The character string after the SAVE command is the name of the file. Filenames can be up to 16 characters long, and they should be chosen so that they describe the program being saved. If the second program saved was the demonstration in the first chapter, a good filename would be DOODLE.

Let's say that you have saved two programs to tape, named POEM and DOODLE, and want to load the second one. Rewind the tape, enter the command LOAD "DOODLE", and follow the normal loading procedure. The first program to be found will be POEM, but the computer will skip this program. Only when it reaches the file named DOODLE will a program be loaded. Many programs can be stored on one tape in this way. The Datassette also has a tape counter which can be used to help you find programs.

The advantages of using a disk drive include greater speed and simplified loading and saving, but perhaps the biggest advantage is the ease with which multiple programs can be handled. To save a second program to a disk, just use the

SAVE command with a different filename. There are no positioning problems, as there are with tape. However, you still use the same device number, 8. The computer will save the second program to the drive. You can later load either program, by typing LOAD, the desired filename, and the device number.

Sometimes you may want to load a program, make some changes, and save the updated version in place of the old one. When using tape, however, updating can be a dangerous practice. Updated programs are often longer than their original versions. If you save an updated program at the same position on the tape as the old copy, you take the risk of erasing the beginning of the next program. You will get the LOAD error if you try to load a program ruined in this way. The best policy: Never save a program on top of an old one. Always save a program at an unused portion of the tape, after the other programs.

The disk drive will not let you save a program in the place of another one. Before the computer saves a program to disk, it checks the disk directory to see if the filename already exists. If so, the red light on the drive will start to flash. Once you have used a specific filename to save a program to disk, you can load from that filename repeatedly, but you cannot save to that filename again. (See the user manual for information about using "@0" to get around this.)

There are a few other errors that can occur when you're working with tape or disk. The device number for the disk drive is 8. If you do not specify a device number, the computer uses a default number of 1, which is for the Datassette. Other numbers like 3 do not correspond to valid devices for saving and loading, and attempting to use them will cause the ILLEGAL DEVICE NUMBER error. If you try to access the disk drive when it is not connected to the computer, you will get the DEVICE NOT PRESENT error. The FILE NOT FOUND error occurs when you try to load a file from disk using a filename that does not exist. Whenever an error occurs on the disk drive, the red light will flash.

You can cancel a SAVE or LOAD command by pressing the RUN/STOP key. The operation will be aborted, and the error message BREAK will be printed.

Loading and saving are called *input/output* operations. Remember that input and output are always described from the viewpoint of the computer. Loading a program means that some information coming from outside the computer (in this case a tape or disk) will be brought into the computer. When you save a program, information in the computer is being sent outside the computer. Loading is an input operation, and saving is an output operation. In computer terminology, the word *read* is often used in place of the word *input*. Likewise, the word *write* is a common substitute for *output*. Therefore, when your Commodore 64 is loading a program, it is reading from the tape or disk, and when it is saving a program, it is writing to the tape or disk.

Although cassette tapes and floppy disks are reliable storage media, they are by no means indestructible, and you should exercise care in handling them. Both types of media work on the principle of magnetism, and information stored on a tape or disk will be lost if the media is placed near a magnet or source of radiation, such as a television set or sunlight. They should not be exposed to extreme temperature and should not be directly touched. The outer shell of the cassette and the square envelope of the disk serve as protection for the sensitive media inside, but they are not enough to guard against magnetic or radiation forces. In addition, a disk should be removed from the drive before turning the drive off or on.

You should acquaint yourself with the other care suggestions described in the user's manual for the Datassette or disk drive. Following a few simple commonsense rules can help prevent the loss of important programs.

Even when utmost caution is used, it is still possible for information stored on a tape or disk to be incorrect. The Datassette and disk drive are mechanical devices, and thus are subject to such problems as static and changing motor speed. There is no way you can prevent errors caused by mechanical problems, but you can double-check that your program was written correctly by using the VERIFY command.

After you save a program, you can use the VERIFY command to reread the program on tape or disk to see if it matches the program in memory. If there is any discrepancy, the computer will report an error.

Let's say that you just saved a program to tape, and want to make sure that the copy of the tape was written correctly. Rewind the tape to the beginning of the program, type the command VERIFY, and press RETURN. Just as for the LOAD command, you will be prompted to PRESS PLAY ON TAPE. The computer will print OK and SEARCHING, and blank the screen. When the program is found, the FOUND message will be displayed. After ten seconds the message VERIFYING will be printed, and the screen will be blanked again (some 64s require that you press the Commodore key).

The next time the screen is restored, one of two messages will have been printed. Either you will see the message OK, which indicates that the program on tape matches the program in memory perfectly, or you will see a VERIFY ERROR, which means that there is a difference between the tape copy and the program in memory, and the save was bad. In the event of a bad save, you should try saving the program again. The procedure is virtually the same when using a disk.

The LOAD and VERIFY commands work much the same way. In fact, there are only two significant differences: With VERIFY, the program in memory is not erased, and the program on tape or disk is not loaded into memory.

To find out the names of all the files on a disk, the LOAD command can be used with the filename "$". Enter the following command.

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

The computer will load into memory the name of every program on the disk. To see the names, use the command LIST.

```
LIST
```

The list of filenames is called a *directory* of the disk. Just like a telephone directory lists names and phone numbers, the disk directory contains the name and other information about each file.

This is a somewhat unusual application of the LOAD command, but it still erases the program currently in memory. Always be careful when you request a disk directory. Be sure that the program in memory has been properly saved.

The Datassette does not support a directory for cassette filenames.

The rest of this section discusses shortcuts for use with saving and loading programs on tape. The first thing you may have noticed is that the ten-second delay between searching and loading can be rather long. All the computer is doing during this time is letting you see that it has found a program and is loading. You can shorten the waiting period by pressing the Commodore key. As soon as this key is pressed, the computer will start loading the program. The SPACE bar, CTRL, and left-arrow keys do the same thing.

Although filenames are required for disk files, they are optional when using tapes. You can type just LOAD and press RETURN, and the computer will load the next program it finds on the tape. If you use SAVE with no filename, the program will be saved on tape, but the only way to retrieve such a file is to use LOAD without a filename.

The LOAD command is more frequently used than SAVE or VERIFY. The last shortcut is an easier way to load and run a program. Just press the Commodore and RUN/STOP keys together, and the command LOAD will be automatically entered (but with no filename). The PRESS PLAY ON TAPE prompt will appear, and you are all set to start loading a program. As soon as the program is loaded, the command RUN will be entered, and the program will start executing. The key combination of SHIFT and RUN/STOP will accomplish the same thing.

Summary

- Programs can be stored on cassette tape or floppy disks with the SAVE command, and retrieved later by using the LOAD command.
- The SAVE command stores a copy of the program in memory onto tape or disk. The program is still in memory after a SAVE command is used.
- The syntax for SAVE is the keyword SAVE followed by a character string which is called a *filename*. When saving to the disk drive, a comma and device number of 8 (a second drive would be device number 9) must be put after the character string.
- The LOAD command copies a program from tape or disk back into the computer's memory. Any program that was already in memory is erased when the new program is loaded.
- The syntax for LOAD is the keyword LOAD, the filename, and a comma and device number of 8 if the disk drive is being used.

- A filename can be up to 16 characters, all of which are significant.
- Multiple programs can be stored on tape or disk by using different filenames. When saving a program to tape, position the tape right at the end of the last program on the tape. The disk drive automatically determines where on the disk the program will be stored.
- It is a dangerous practice to save a revised program to tape on top of an earlier version, because the next program on the tape may be overwritten.
- The disk drive will not normally let you save a program with a filename that is already in use.
- The ILLEGAL DEVICE NUMBER error means that an attempt was made to use a device which does not support loading or saving. The only device numbers normally used are 1 (cassette) and 8 (disk). If a device number is not specified after a filename, the computer assumes that the cassette is to be used.
- The DEVICE NOT PRESENT error occurs when a referenced device does not respond to the computer, because it is either not hooked up or not turned on.
- If you try to load a disk file using a filename that does not exist, you will get the FILE NOT FOUND error.
- When errors occur on the disk drive, the red light flashes.
- Storage and retrieval commands can be aborted by pressing the RUN/STOP key.
- LOADing is an input operation, and SAVEing is an output operation.
- The terms *read* and *write* are often used to mean *input* and *output.*
- Care should be exercised in handling cassette tapes and floppy disks. They should not be exposed to radiation, magnets, temperature changes, or fingers.
- The VERIFY command is used to read a program on tape or disk and check if it matches the program in memory. If it doesn't, the VERIFY ERROR message is printed.
- The main difference between the LOAD and VERIFY commands is that with VERIFY, the program in memory is not touched. It is only compared against the program on the tape or disk.

- A disk directory is a list of filenames. To get a directory, use the LOAD command with the filename "$", then use LIST. This works only for disk files.
- The ten-second delay between the searching and loading of a tape file can be shortened by pressing the SPACE bar, left arrow, CTRL, or Commodore key.
- Filenames are optional for tape files. Using LOAD with no filename will make the computer load the next program found on the tape. Files saved without a filename can be retrieved only by using LOAD without a filename.
- To have the computer automatically load and run the next program found on a tape, press the RUN/STOP key with either the Commodore or SHIFT key.

Documenting a Program Using REM

When you type a statement like POKE 53280,14 into a program, you may know exactly what it means and what it does at the time. But after running a program for a week, you may decide to modify it. Will you still remember the function of every statement in your program a week after you type it in? As you write longer and more complex programs, you will need to leave notes to yourself, to remind you of what variables mean, and where special sequences begin and end. BASIC provides a special statement for this purpose.

The REM statement lets you embed *remarks* right in the program. Your remarks do not appear when the program is loaded or executed, only when it is listed. They might provide commentary about a program, telling how the program was written and how it works, or they might contain notes to yourself, marking lines that you want to revise later.

Here is an example of how remarks might be used.

```
10 REM RANDOM COLOR PROGRAM                      :rem 157
20 REM CRAIG CHAMBERLAIN                         :rem 194
30 POKE 53280, INT(RND(0)*16) : REM CHANGE BORDER
   {SPACE}COLOR                                  :rem 243
40 POKE 53281, INT(RND(0)*16) : REM CHANGE BACKGRO
   UND COLOR                                      :rem 23
```

The syntax for the REM statement is the keyword REM, optionally followed by any kind of information. The example shows many ways in which the REM statement can be put to use. REM is often used to provide general information, like the name of a program, the author's name, and the date the program was written. For longer programs, a program version number may also be given. For programs that are to be distributed to others, the author may want to put in an address or telephone number. And if the program is protected by copyright, this is one place where the copyright notice should be displayed.

Lines 10 and 20 use REM to identify the title and author of the program. The text after the REM is optional. A line with only a REM on it can be used to separate parts of a program. Longer programs are often divided into several sections, and a REM statement can be used before each section to describe the purpose of that particular section.

A REM statement does not have to take a whole line. You can put a REM after a few other statements on the same line. The only requirement is that the REM be the last statement on the line. REM statements used in this way usually explain a particularly tricky line of the program, or draw attention to a key part in the processing. The REM statements in lines 30 and 40 are simplistic, but they do explain what is happening on those lines.

If the computer encounters a REM statement in a line while executing a program, it ignores the rest of the line and moves on to the next line. That is why a REM statement has to be the last thing on each line. In the next example, the second POKE statement will never be executed.

```
10 POKE 53280,0 : REM BORDER : POKE 53281,0 : REM
    BACKGROUND
```

The computer stops executing the line as soon as it reaches the first REM statement. If a line starts with a REM, the whole line will be ignored.

The listings in this book use another type of rem statement. The rems at the end of each program line which are written in lowercase letters should not be typed. The lowercase rem statements are not part of the program, but rather a tool to help you enter programs correctly (see Appendix I for more information).

Summary

- The REM statement lets you embed messages in a program listing.

- These messages are called comments or remarks because they are often short descriptive phrases which explain something about the program.
- The syntax for a REM statement is the keyword REM optionally followed by additional text.
- The computer stops executing a line as soon as it comes to a REM statement.
- REM statements can be placed at the beginning of a line or after other statements. The REM statement should be the last statement on a line.
- REM statements are often used as the first lines of a program to state the program title, identify the author, and give the version number of the program.
- Empty REM statements are used to put blank lines in a program listing.
- REM statements can be used as a title before each section, giving pertinent information about that section.
- REM statements used after other statements on a line often highlight critical parts of a program, or parts which require special attention before they can be understood.
- In a broad sense, remarks are used to document a program, presenting information that would be helpful to a programmer.

The END Statement

When you enter the RUN command, the computer stops operating in the immediate mode and starts working in the deferred mode. It sequentially executes every line in the program and returns to the immediate mode only when an error occurs, or when the last line of the program has been executed. As a formality, there is an optional statement, END, that can be used to terminate program execution.

```
10 REM DEMONSTRATION OF END STATEMENT        :rem 77
20 REM TOASTER POEM                         :rem 153
30 PRINT "SILVER BOX"                       :rem 242
40 PRINT "BY EDWARD CHU"                    :rem 103
50 END                                       :rem 60
```

The syntax for the END statement is the keyword END alone.

Whenever the computer encounters an END statement, it stops executing the current program and returns to the immediate mode. To see that it really does stop program execution, move the END statement to line 35, and run the program again.

Admittedly, there is not much to be gained by putting an END statement as the last line of a program, since the program will end anyway. And putting an END statement in the middle of a program doesn't seem to make much sense, either. But the END statement does have its uses.

```
10 REM SECOND DEMO OF END STATEMENT          :rem 71
20 REM                                        :rem 70
30 N=2                                        :rem 32
40 IF N<0 OR N>15 THEN PRINT "ERROR! NUMBER OUT OF
    RANGE!" : END                            :rem 224
50 POKE 53280,N                               :rem 16
60 POKE 53281,N                               :rem 18
```

In the above example, the END statement is used to terminate program execution early if the value of variable N is out of the range of 16 colors. If not, the program continues.

Of course, there would never be any reason to place statements after an END statement on the same line. As soon as the computer reaches the END statement, execution stops, and the computer will never get to the other statements.

Summary

- The END statement causes the computer to stop executing a program, and return to the immediate mode.
- The syntax for the END statement consists only of the keyword END.
- A program automatically ends after the last line has been executed, so there is no real need to put an END as the last line. Using END in this way is a formality, not a necessity.
- END can be placed on a line by itself, or after other statements on a line. This is often done in conjunction with the IF-THEN statement. It is sometimes desirable for a program to end early, such as when a potential error has been detected.
- Statements placed after an END statement on the same line will never be executed.

Chapter 8

Controlling Program Execution

Controlling Program Execution

The GOTO Statement

You have seen that when the computer runs a program, it executes the first line, then the second, and so on until there are no more lines, or the END statement is encountered. The lines are executed sequentially, according to the line numbers.

Remember when we introduced conditional logic? We showed how the IF-THEN statement provided control over the execution of statements on one line. Now, with a program, we could use something to provide control over the execution of program lines. This is available with the GOTO statement. Enter and run the following program. (Be sure to first type NEW if there is an old program in memory. Future examples will not include a reminder.)

```
10 PRINT "CO"                                  :rem 196
20 PRINT "COMM"                                 :rem 95
30 PRINT "COMMOD"                              :rem 243
40 PRINT "COMMODOR"                            :rem 149
50 PRINT "COMMODORE"                           :rem 219
60 PRINT "COMMODORE 64"                         :rem 70
70 GOTO 10                                       :rem 1
80 PRINT "A FRIENDLY COMPUTER"                  :rem 70
```

The computer not only executes the lines in the program, but it keeps on executing them. When the computer sees the statement GOTO 10 on line 70, it ignores anything after line 70, and continues execution by going back to the first statement of line 10. This will go on forever, and line 80 never gets executed.

The GOTO statement alters the flow of execution from the normal course. When GOTO is used to make the computer execute a bunch of lines several times, the lines form a *loop*. A loop which continues forever, like the one in the example, is called an *infinite loop*.

Since the computer executes statements so quickly, it can be difficult to see exactly what the computer is doing when it is caught in an infinite loop. To slow down the scrolling, press the CTRL key, just as you did to make the computer list programs more slowly. At least now you can see what is being printed on the screen. The printing will return to normal speed as soon as you release the CTRL key.

A program isn't of much use if it's stuck in an infinite loop. To stop the computer when it's in an infinite loop, or at any time it's executing a program, press the RUN/STOP key. The computer will immediately stop executing the program, and will print the message BREAK IN followed by a line number. The line number indicates which line was being executed when the RUN/STOP key was pressed. The computer will now be in the immediate mode.

The GOTO statement does not have to make the computer repeat some lines. It can also be used to make execution skip ahead in the program.

```
10 C=INT(RND(0)*16)                           :rem 19
20 PRINT "THE NUMBER" C "IS ";                :rem 59
30 IF C/2 = INT(C/2) THEN PRINT "EVEN" : GOTO 50
                                              :rem 185
40 PRINT "ODD"                                :rem 12
50 POKE 53280, C                               :rem 5
```

In this example, the GOTO in line 30 is used to bypass line 40. If the number is even, the GOTO 50 will be executed. The only other possibility is that the number is odd, and line 40 handles that. If there were a lot of lines between 30 and 50, the GOTO 50 would skip around all of them.

With just a couple of changes to the program, you can make the computer rapidly change the background and border colors for a psychedelic effect.

```
30 IF C/2 = INT(C/2) THEN PRINT "EVEN" : POKE 5328
   1, C : GOTO 10                             :rem 144
60 GOTO 10                                     :rem 0
```

The syntax for the GOTO statement consists of the keyword GOTO followed by a line number. This is one case where a variable, expression, or function cannot be used. Only a line number is acceptable.

You can also spell GOTO with a space between GO and TO. This is the only keyword in which this is allowed. This is done to maintain compatibility with other versions of BASIC which spell the keyword with a space. The GO TO statement works just like GOTO.

Because the GOTO statement is used to move program execution forward or backward to a different line, we say that the GOTO statement makes BASIC *jump* to a new line.

GOTO can jump only to the beginning of a line. If the computer has to GOTO a line which contains many statements, execution will start at the first statement. Also, GOTO should always be the last statement on a line because, just like the END statement, anything after the GOTO will never be executed.

The most common error people make with GOTO is attempting to GOTO a line which is not in the program. When the computer tries to execute the statement GOTO 15 and there is no line 15 in the program, the error message UNDEF'D STATEMENT will be printed. This is the computer's way of telling you that line 15 does not exist. Another way to get this error is to use a variable in place of the line number or to forget the line number altogether in the GOTO statement. In this case the computer will assume a line number of 0. If a line 0 exists, the program will jump there; if it doesn't exist, you'll get the error message.

Summary

- The GOTO statement changes the order in which program lines are executed.
- Normally, after the computer executes one line, it will execute the next one in the sequence. The GOTO statement causes execution to *jump* to a different line.
- GOTO can move execution forward or backward in the program.
- When GOTO is used to skip backward so that lines can be executed again, the lines that are repeated form a loop. If these lines are executed continuously, without ever stopping, the loop is called an *infinite loop.*
- If your program enters an infinite loop, execution can be aborted at any time by pressing the RUN/STOP key.
- The GOTO statement makes the computer jump to the beginning of the new line. Execution starts with the first statement on the line.
- You should not put statements after a GOTO on the same line, because they will never be executed.

- The syntax for this statement is the keyword GOTO and a line number. A number must be used; variables and expressions are not valid.
- An alternate spelling for the name of the GOTO statement is GO TO.
- If an attempt is made to GOTO a line which is not in the program, or if a variable is used instead of a line number, the UNDEF'D STATEMENT error will occur, which means that the referenced line is undefined.

Loops

Enter and run the following one-line program.

```
10 GOTO 10                                         :rem 251
```

You might call this the ultimate infinite loop. It certainly doesn't accomplish very much. All it does is prevent the READY prompt from ever being printed. Loops like this have only a few uses, and even infinite loops like the one in the last section are not too useful. Repetition is an essential part of programming, but the problem with using GOTO to form loops is that the loops are endless. To solve the problem of infinite looping, loops are often used with conditional logic. The IF-THEN and GOTO statements are a powerful combination. The next program shows how these statements are used together.

```
10 A=TIME                                           :rem 14
20 PRINT "STARTING"                                :rem 159
30 IF TIME-A < 60 THEN GOTO 30                     :rem 252
40 PRINT "TIME'S UP!"                              :rem 164
```

This example uses the reserved variable, TI or TIME, which counts in sixtieths of a second, to make the computer wait one second. The program notes the time when it starts, and stores this value in A. Notice that the value of TI keeps increasing. As long as the difference betwen TI and A is less than 60, execution keeps looping at line 30. Execution falls out of the loop when the expression TI − A is greater than or equal to 60. Add this line to the program to see that the delay is one second.

```
50 GOTO 10                                         :rem 255
```

By using a value of 600 instead of 60, you create a ten-second delay.

In the example above, we used the variable TI to let the computer count automatically. You can also use the IF-THEN GOTO combination to have your program count things other than time.

```
10 A=A+1                                          :rem 124
20 PRINT A                                         :rem 48
30 GOTO 10                                        :rem 253
```

When you run this program, the computer prints a column of numbers, starting with 1 and increasing forever. Every time the loop is executed, the value of A increases by one. The loop is infinite, but at least you can tell how many times the loop has been executed, just by looking at the value of A. Now, add one more step, and you can make the program do the counting itself.

```
30 IF A<5 THEN GOTO 10                            :rem 109
```

Introducing the conditional logic based on the counting variable A means that the loop can be executed a specific number of times—in this case, five. Loops controlled by a counting variable normally do something other than print the counting variable. Here we replace line 20 with another PRINT statement:

```
20 PRINT "COMMODORE 64"                            :rem 66
```

Having the computer print its name instead of the value in the counting variable is not a big change, but it does demonstrate that you can place any statements inside the loop and cause them to be executed any number of times. You change the number of executions by changing the value compared to A in IF A<5 THEN GOTO 10.

Now that we have the means to execute a loop a certain number of times, let's have the loop do something different every time it is executed. The next example uses the variable F not only to control how many times a loop is executed, but also to change the pitch of a note that is played. This time, however, F is changed by 4 instead of 1. The variable A is again used in a delay loop.

```
10 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:CT
   =54276:F=4                                     :rem 148
20 POKE MV,15 : POKE AD,15 : POKE SR,168 : POKE FL
   ,0                                             :rem 118
30 POKE FH,F : PRINT F                            :rem 159
```

```
40 POKE CT,17 : A=TI : REM START NOTE         :rem 245
50 IF TI-A < 60 THEN GOTO 50 : REM WAIT ONE SECOND
                                               :rem 95
60 POKE CT,16 : A=TI : REM RELEASE NOTE       :rem 105
70 IF TI-A < 15 THEN GOTO 70 : REM WAIT A QUARTER
   {SPACE}OF A SECOND                          :rem 188
80 F=F+4 : REM INCREASE FREQUENCY              :rem 170
90 IF F<=120 THEN GOTO 30                       :rem 21
```

As with many other frequently used statements in BASIC, there are a couple of shortcuts that can make loops easier to use. Both shortcuts involve the syntax of the IF-THEN statement. When combined with a GOTO statement, the whole thing looks like:

IF conditional expression THEN GOTO line number

The idea is to find some way to shorten the line. This may seem difficult because every part of the line is necessary. However, in this case the keyword GOTO is optional. The reasoning is that the only thing which should ever come after the THEN is a keyword for a statement (or a variable name for an implied LET statement), so if a number is found there, it is assumed to be a line number, and the GOTO statement is implied. You can write the above line as:

IF conditional expression THEN line number

This is not a great savings in terms of typing, but it does simplify the line a little, and simplicity is a key to good programming.

An alternative shortcut is to make the THEN optional instead of the GOTO, giving you:

IF conditional expression GOTO line number

Since this method requires just as much typing as the first one, the distinction between the two shortcuts is merely one of personal taste. It all depends on whether you prefer to emphasize the THEN or the GOTO when the two are used together. To see the shortcuts in use, change lines 50 and 70 in the example. The program will work just as before.

```
50 IF TI-A < 60 THEN 50                        :rem 53
70 IF TI-A < 15 GOTO 70                        :rem 67
```

Note that the second shortcut is the only case when the conditional expression after an IF is followed by anything other than a THEN.

Summary

- To avoid the problem of GOTO forming an infinite loop, conditional logic must be used. The combination of IF-THEN with GOTO is one way to achieve this.
- A variable can be used in a LET statement such as A=A+1 to count how many times a loop is executed. A variable used in this manner is called a *counting variable.*
- By basing conditional logic on a counting variable, you can have a program control exactly how many times the loop is executed.
- Sometimes the value of a counting variable is used within a loop, so that something is done differently every time a loop is executed.
- Because it is used so often, there are two short forms to the IF-THEN and GOTO combination. The first is to make the GOTO optional, and the second is to make the THEN optional.

Game Demonstration

As a demonstration of how to use all the concepts introduced thus far, here is a simple game program called "Hurkle" (origin unknown). The object is to fire missiles to hit a falling Hurkle before it crashes on the ground. You get ten chances, and you cannot fire a new missile until the last one has moved off the screen. Type in the program and try it.

```
110 S=1024:C=55296:T=0:HX=0:HS=0:PX=0:MS=0:DH=0:JS
    =0:PT=56320:R=40:HC=1                       :rem 186
120 B=32:H=81:P=98:M=30:SC=0                     :rem 59
200 HX=INT(RND(0)*38)+1:HS=HX:DH=SGN(RND(0)-0.5):P
    X=20:MS=-1:MH=1                              :rem 71
210 PRINT "{CLR}":T=56256                       :rem 202
220 POKE T,3:T=T+1:IF T<56296 GOTO 220           :rem 35
230 POKE 1984+PX,P:POKE 53280,0                 :rem 176
300 MH=MH-1:IF MH>0 GOTO 340                    :rem 244
305 MH=4:T=HS:HS=HS+R+DH:HX=HX+DH:IF HX=0 OR HX=39
     THEN DH=-DH                                :rem 196
310 IF HS>959 GOTO 500                          :rem 113
320 IF PEEK(S+HS)=M GOTO 400                     :rem 10
330 POKE C+HS,8:POKE S+HS,H:POKE S+T,B          :rem 209
340 IF MS<0 GOTO 370                              :rem 5
350 T=MS:MS=MS-R:IF MS<0 THEN POKE S+T,B:GOTO 370
                                                :rem 127
360 IF PEEK(S+MS)=H GOTO 400                     :rem 14
```

```
365 POKE C+MS,13:POKE S+MS,M:POKE S+T,B     :rem 20
370 JS=PEEK(PT):IF (JS AND 16)=16 OR MS>=0 GOTO 38
    0                                        :rem 227
375 MS=920+PX:POKE C+MS,13:POKE S+MS,M     :rem 241
380 T=PX:IF (JS AND 4)=0 AND PX>0 THEN PX=PX-1
                                               :rem 2
385 IF (JS AND 8)=0 AND PX<39 THEN PX=PX+1:rem 208
390 IF PX<>T THEN POKE 1984+PX,P:POKE 1984+T,B
                                              :rem 80
395 GOTO 300                                 :rem 109
400 POKE S+T,B:SC=SC+1:T=65                   :rem 56
410 POKE S+HS,T:POKE C+HS,T:T=T+1:IF T<120 GOTO 41
    0                                        :rem 132
500 HC=HC+1:IF HC<=10 GOTO 200                :rem 61
510 PRINT "YOU GOT" SC "OUT OF 10 HURKLES" :rem 52
520 IF SC=0 THEN PRINT "BETTER LUCK NEXT TIME"
                                             :rem 140
530 IF SC=10 THEN PRINT "YOU ARE AN ACE HURKLE HUN
    TER!"                                     :rem 74
```

Now that you have played the game a few times, let's take it apart and see how it works.

Here are the variables used in the program. The numbers in parentheses tell the possible values of each variable. Some variables are assigned once and never change; others have a range of values.

S	(1024) beginning of screen memory
C	(55296) beginning of color memory
HS	(0 to 999) Hurkle screen position—offset from top of screen
MS	(−40 to 999) missile screen position; negative value means no missile on screen
T	(0 to 999) temporary; holds old position of Hurkle, missile, or player
HX	(0 to 39) Hurkle horizontal (X axis) position
PX	(0 to 39) player (base) horizontal position
DH	(−1, 0, 1) direction of Hurkle (−1 for left, 1 for right, 0 for straight down)
MH	(0 to 4) move Hurkle delay; counter to move Hurkle once for every four player moves
HC	(1 to 11) Hurkle count, number of Hurkles that have fallen
SC	(0 to 10) score—number of Hurkles that have been hit
PT	(56320) hardware location for port 2

JS	(0 to 255) joystick information; contents of port 2
R	(40) row offset—number of bytes per row; add to or subtract from position for vertical movement
B	(32) blank; POKE value to erase screen image
H	(81) POKE value for Hurkle
P	(98) POKE value for player
M	(30) POKE value for missile

The following description shows the logic used in the program.

Program
110 program initialization (setup); executed only once per run
200 setup before Hurkle starts falling
300 main loop to move Hurkle, missile, player, and check for collision
300–330 process Hurkle movement
340–365 process missile movement
370–375 fire missile
380–395 player movement
400 handles a Hurkle hit
500 prepares for next Hurkle, or ends game if last one

Lines
110–120 establish initial values for variables
200 HX=INT(RND(0)*38)+1 : HS=HX
start Hurkle at random position at top of screen
DH=SGN(RND(0)−0.5)
randomly choose Hurkle falling direction to the left, right, or straight down
PX=20
set player's base at middle of bottom row
MS=−1
indicate that no missile has been fired
MH=1
prepare to move Hurkle first time through main loop
210 PRINT "{CLR}"
clear the screen
T=56256
point to color memory for bottom screen row
220 POKE T,3 : T=T+1 : IF T<56296 GOTO 220
put the player color at each position of the bottom row

230 POKE 1984+PX,P
    POKE the player base onto the screen
    POKE 53280,0
    set border color to black
300 MH=MH−1
    decrement the Hurkle movement counter
    IF MH>0 GOTO 340
    if still not time to move Hurkle, skip past Hurkle
    movement lines
305 MH=4
    reset Hurkle to not move until another four times
    through main loop
    T=HS
    remember current Hurkle position
    HS=HS+R+DH
    position the Hurkle down one row, and to the left or right
    HX=HX+DH
    update the horizontal position of the Hurkle
    IF HX=0 OR HX=39 THEN DH=−DH
    if the Hurkle is at either side of the screen, change to
    opposite direction
310 IF HS>959 GOTO 500
    if Hurkle's new position is somewhere in the bottom
    screen row, Hurkle has crashed
320 IF PEEK(S+HS)=M GOTO 400
    if there is already a missile at the screen position where
    the Hurkle is going to move, the Hurkle is hit
330 POKE C+HS,8 : POKE S+HS,H
    okay to move Hurkle, so POKE color and screen memory
    for new position
    POKE S+T,B
    erase old position so Hurkle leaves no trail
340 IF MS<0 GOTO 370
    if there is no missile on the screen, skip the code to move
    the missile
350 T=MS
    remember the old position for the missile
    MS=MS−R
    change the missile position to one row higher
    IF MS<0 THEN POKE S+T,B : GOTO 370
    if new position is off the screen, erase the current missile
    and bypass remaining code

360 IF PEEK(S+MS)=H GOTO 400
    if there is a Hurkle where the missile is going to move, the Hurkle has been hit
365 POKE C+MS,13 : POKE S+MS,M
    move missile to new position
    POKE S+T,B
    erase old missile image
370 JS=PEEK(PT)
    get the current joystick information
    IF (JS AND 16)=16 OR MS>=0 GOTO 380
    if the trigger is not being pressed, or a missile is still on the screen, skip around the code to fire a missile
375 MS=920+PX
    missile starting position is at same horizontal position as player, but one row higher
    POKE C+MS,13 : POKE S+MS,M
    POKE missile onto screen
380 T=PX
    remember the old position of the player base
    IF (JS AND 4)=0 AND PX>0 THEN PX=PX−1
    if the stick is being pushed left, and the player base is not already at the left edge of the screen, new player position is one position to left
385 IF (JS AND 8)=0 AND PX<39 THEN PX=PX+1
    if stick is being pushed right, and the player is not at right edge of screen, player position is changed one position to right
390 IF PX<>T THEN POKE 1984+PX,P : POKE 1984+T.B
    if the player position has been changed by line 380 or 385, update the screen by POKEing new position and erasing old
395 GOTO 300
    end of main loop
400 POKE S+T,B
    erase either Hurkle or missile
    SC=SC+1
    update score to show another Hurkle hit
    T=65
    prepare for explosion
410 POKE S+HS,T : POKE C+HS,T
    rapidly change character and color for explosion effect
    T=T+1 : IF T<120 GOTO 410
    repeat for characters 65 to 119

500 HC=HC+1
    increment Hurkle count to show one more Hurkle has crashed or been hit
    IF HC<=10 GOTO 200
    if more Hurkles left, go prepare for main loop again
510 PRINT "YOU GOT" SC "OUT OF 10 HURKLES"
    display score
520–530 print special messages

Here are some important notes about the program. The program is organized into three main parts: initialization, main loop, and termination. The main loop lets three objects move independently, and repeats indefinitely until the Hurkle crashes or is hit. Program execution moves from the explosion-handling sequence right into the section which starts another Hurkle. This is the basic structure of many programs, and is suitable for other games.

The easiest way to make an object move is to POKE the number for a blank into its current position, calculate the movement, and POKE the character into the new position. Unfortunately, objects moved in this manner seem to blink because for a fraction of a second between the two POKEs, there is no object displayed. For a better visual effect, this program always POKEs the character to the new position first, and then erases the character at the old position. This requires that the variable T be used to keep track of the old position when the new position is calculated.

Another technique that makes the movement look better is to always POKE the color value before POKEing the screen value. POKEing the character first means that there will be a fraction of a second when the character is displayed in the wrong color. If the color is POKEd first, it will not appear until there is a character at the screen position, so there's no flash of a wrong color.

One of the best ways to gain experience in programming is to modify existing programs. You could change the characters used for the objects by changing the assignments to H, P, and M in line 120. A slightly more difficult task would be to change the color for each object. To see the Hurkle leave a trail, delete the statement POKE S+T,B in line 330. The game can be made more difficult by assigning a smaller number to MH in line 305. Finally, for a real workout, try adding sound effects to the game.

Chapter 9

Data Storage

Data Storage

Assigning Variables with READ and DATA

Up until now we have used only the LET statement to assign values to variables. You may recall how we had to assign a lot of hardware addresses to variables for use in sound demonstrations. This entailed one LET statement after another. Another way to do this would be to use the statements READ and DATA.

The READ statement tells which variables are to be assigned values. The DATA statement contains the values to be assigned. These two statements work together to assign values to variables. Here is an example of how a variable could be assigned using LET.

```
10 MV=54296                                :rem 75
```

Using READ and DATA, two statements would be required.

```
10 READ MV                                 :rem 32
20 DATA 54296                             :rem 134
```

The READ tells the computer that the variable MV is to be assigned. The computer scans the whole program, starting at the first line, until it comes to a DATA statement. The value after the keyword DATA is assigned to the variable specified by READ.

The DATA statement does not have to be after the READ. The example would work just the same if it were rewritten with the DATA statement before the READ.

```
10 DATA 54296                             :rem 133
20 READ MV                                 :rem 33
```

By itself, a DATA statement does nothing. When the computer comes across a DATA statement in the course of executing a program, it ignores the statement, just as it ignores a REM statement. Unlike REM, however, other statements can be put after a DATA statement on the same line.

The advantage to using READ and DATA instead of LET is that several variables can be assigned with one READ statement. This is because READ can be followed by a *variable list,* a bunch of variable names separated by commas. The next line shows the assignment of six variables using LET.

```
10 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:CT
   =54276                                 :rem 163
```

Using READ and DATA, the assignment is again accomplished with two statements.

```
10 READ MV,AD,SR,FL,FH,CT                        :rem 221
20 DATA 54296,54277,54278,54272,54273,54276
                                                 :rem 134
```

Here is what happens. When the computer finds the READ statement, it looks at the first variable name after the keyword READ. Then it looks at the first number following the keyword DATA in the DATA statement. This value is assigned to the specified variable. If there are more variables indicated by the READ statement, the second variable is assigned the second value in the DATA statement, and so on. The whole process keeps repeating, until all of the variables have been assigned. Each time, the next number in the DATA statement is used.

A single READ statement does not have to be followed by as many variable names as there are numbers in the DATA statement. There can be several READ statements in a program, with each one assigning just a few variables. Values which are not read the first time that a READ is executed might get read the second time.

```
10 READ A,B : PRINT A : PRINT B                   :rem 61
20 DATA 1,2,3,4                                  :rem 202
30 READ C,D : PRINT C : PRINT D                   :rem 71
```

Only the first two numbers of the DATA are read by line 10. But the computer remembers at which point reading stopped. The next time a READ is executed, the computer picks up where it left off in the DATA statement. In the above example, variables C and D get assigned the values 3 and 4.

There can also be several DATA statements in one program. When more than one is used, the first one in the program gets priority. Adding this line to the example program will completely change the values assigned to A, B, C, and D.

```
15 DATA 7,8,9,10                                  :rem 13
```

Because this line is numbered 15, these data values will be read before those in line 20. But when all of the values of line 15 have been read, the DATA statement of that line is no longer of any use, and future data will be read from line 20. When you add the following line to the program, the variable E should be assigned the value 1.

```
40 READ E : PRINT E                          :rem 209
```

It is okay for a program to end without all of the data having been read. No error message will be printed.

In this example, the data values 2, 3, and 4 are left unread when the program ends.

On the other hand, if a program tries to read more values than are available in DATA statements, there is a definite problem.

```
10 READ A : PRINT A                          :rem 198
20 READ B : PRINT B                          :rem 201
30 READ C : PRINT C                          :rem 204
40 DATA 6,4                                   :rem 20
```

All of the data values have been read after line 20 is executed. When the READ in line 30 attempts to assign the variable C, an OUT OF DATA error message is printed.

When reading numeric data, no variable names or expressions are allowed in DATA statements. Only numbers can be used, and they must be separated by commas. A line like DATA 2+3 will produce a SYNTAX ERROR and report the line number of the DATA statement. If the computer reads two commas with no number in between, it assumes the number 0.

Summary

- The READ and DATA statements offer an alternative method for assigning values to variables.
- The syntax for the READ statement is the keyword READ followed by a list of variable names, called a *variable list.* Each variable name would be to the left of the equal sign if a LET statement were used. When more than one variable name is used in a READ statement, the variable names are separated by commas.
- The syntax for the DATA statement is the keyword DATA followed by one or more numbers. If several numbers are used, they are separated by commas. Two consecutive commas are read as the number 0. Only constant values can be used; variables or expressions cannot be placed in DATA statements.
- The READ statement is the one that does all the action. The variables are assigned when the READ is executed. The

DATA statement just supplies values for READ. The computer ignores the DATA statement unless a READ statement is being executed. For that reason, DATA statements are often placed at the end of a program, out of the way of other statements.

- The first value in a DATA statement is the first one that is assigned to a variable. The computer remembers which values in a DATA statement have already been read. If more than one READ statement is executed, the second READ will start with the next value not yet read.
- If there is more than one DATA statement, reading will start with the one which has the lowest line number. Once all of the values in that statement have been read, future values will be read from the next DATA statement.
- An attempt to assign more values than are available in DATA statements will result in the OUT OF DATA error.

READ and DATA in Loops

The ability to quickly assign values to different variables is handy, but this is not the most important use of READ and DATA. The real power of these statements becomes evident when they are used to assign different values to the same variable, one value after another. This is done in a loop.

```
10 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:CT
   =54276                                    :rem 163
20 POKE MV,15:POKE AD,0:POKE SR,168          :rem 233
30 READ F                                    :rem 197
40 POKE FL,F-256*INT(F/256) : POKE FH,INT(F/256)
                                             :rem 74
50 POKE CT,33 : A=TI : POKE CT,32            :rem 163
60 IF TI-A < 30 GOTO 60                      :rem 62
70 GOTO 30                                    :rem 3
90 DATA 1607,2025,2145,1804,2408,2145,2025,1804
                                             :rem 23
```

Here is a case where READ is being used to do something which could never be done with LET. The program plays notes with different frequencies for a musical bass line. An equivalent program that did not use READ and DATA would be considerably longer, and a loop could not be used.

There is a problem with this program, though. Because it uses GOTO to create an infinite loop, an error occurs when the program runs out of data. This is a sloppy way of ending a program. We can avoid the error by using conditional logic to check when the last data value has been used. A simple change to line 70 will do.

```
70 IF F<>1804 GOTO 30                             :rem 31
```

With this change, the program should loop back to line 30 every time except the last time, when the program will end normally. Try it.

Even though the OUT OF DATA error never occurs in the modified program, there is still a problem. The program ends prematurely, after playing only four notes. This is because the last data value is the same as the fourth one. The way the program is written, it will end after the first 1804, and never get to the last one. In solving the problem of running out of data, we have created a new one: not using all of the data. The idea of using IF-THEN to end the loop is a good one; the problem is in the value being checked. The solution is to use a value that is called a *flag* as the last number in the DATA statement. Make the following changes to the program:

```
30 READ F : IF F=-999 THEN END                   :rem 239
70 GOTO 30                                         :rem 3
90 DATA 1607,2025,2145,1804,2408,2145,2025,1804,-9
   99                                             :rem 27
```

The value −999 is never used in the POKE statement. Its only purpose is to indicate when all of the data has been read. Unlikely values are often used as flags; in the example, −999 would never be used as a POKE value. By using the technique of flag data values, we've resolved the problems of using READ and DATA in loops.

Summary

- The real advantage of using READ to assign variables is that it can be used in a loop to assign different values to the same variable. The LET statement cannot do this.
- Using infinite loops does create the problem of running out of data, though, so conditional logic must be used to determine when there is no more data to be read.
- The best technique for finding the end of data is to use a flag value. This is an out-of-range value that would normally not be used as data. As soon as the program reads this value, the loop is done.

The RESTORE Statement

Once all values in all DATA statements have been read, trying to read any more causes the OUT OF DATA error. So if you want a program to read the same sequence of numbers more than once, you need a way to reset the pointer to the first DATA statement, as if the program had just started running. That is the purpose of the RESTORE statement. RESTORE causes the next READ statement to take the first value of the first DATA statement in the program, and further reading will continue from that point. Make the following change to the demonstration program from the last section:

```
30 READ F : IF F=-999 THEN RESTORE : GOTO 30
                                             :rem 18
```

Now the bass line repeats forever, with no OUT OF DATA error.

All the data in a program does not have to be read before RESTORE is used. RESTORE will reset the data pointer back to the first value in the program no matter how much data has been read.

The statement CLR automatically performs a RESTORE, and so does the command RUN. By itself, RESTORE is not a frequently used statement.

Summary

- The RESTORE statement resets the data pointer so that reading will start from the first value in the first DATA statement of the program. This makes it possible to use READ in an infinite loop without getting an OUT OF DATA error.
- It does not matter how much data has already been read when RESTORE is used. The data pointer will always be restored to the initial state when the program started running.
- The CLR statement automatically performs a RESTORE. Because the RUN command does a CLR before starting program execution, RUN also performs a RESTORE.
- The RESTORE statement is used only with READ and DATA, and even then it's not used very often.

Music Demonstration

To demonstrate one application of READ and DATA in loops, here are some short tunes for your listening pleasure. The first listing is the main playing program. Enter this short program and save it using the filename "PLAYER". Then, for each tune, load the player, enter the appropriate DATA statements, and type RUN to hear the music.

Player

```
100 PRINT "MUSIC PLAYER" : PRINT              :rem 119
110 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:C
    T=54276:PW=54275                          :rem 249
120 READ T1,T2,E1,E2,WF : POKE MV,15          :rem 146
130 POKE AD,E1 : POKE SR,E2 : REM ENVELOPE :rem 23
140 IF WF=64 THEN READ W : POKE PW,W : REM PULSE W
    IDTH                                      :rem 196
150 READ F : IF F<0 THEN END                  :rem 121
160 A=TI : IF F>0 THEN POKE FL,F AND 255 : POKE FH
    ,F/256 : POKE CT,WF+1                      :rem 90
170 IF TI-A < T1 GOTO 170                     :rem 196
180 POKE CT,WF : A=TI                         :rem 125
190 IF TI-A < T2 GOTO 190                     :rem 201
200 GOTO 150                                   :rem 97
```

London Data

```
300 DATA 20,3,85,172,64,8 : REM LONDONDERRY AIR
                                               :rem 17
310 DATA 0,4143,4389,4927,5530,0,0,4927,5530,7382,
    6577,5530,4927,4389,3691,0                :rem 231
320 DATA 0,4389,5530,5859,6577,0,0,7382,6577,5530,
    4389,5530,4927,0,0,0                      :rem 177
330 DATA 0,4143,4389,4927,5530,0,0,4927,5530,7382,
    6577,5530,4927,4389,3691,3288             :rem 142
340 DATA 3691,4143,4389,4927,5530,0,0,5859,5530,49
    27,4389,4927,4389,0,3288,0                :rem 248
350 DATA 2195,6577,7382,8286,8779,0,0,8286,8286,73
    82,6577,5530,6577,5530,4389,0             :rem 179
360 DATA 0,6577,7382,8286,8779,0,0,8286,8286,7382,
    6577,5530,4927,0,0,0                      :rem 203
370 DATA 0,6577,7382,6577,11060,0,0,9854,9854,8779
    ,7382,8779,6577,5530,4389,0                :rem 74
380 DATA 0,4143,4389,4927,5530,0,0,5859,5530,4927,
    4389,4927,4389,0,3288,0                    :rem 89
390 DATA 2195,0,0,0,-1                         :rem 37
```

Early Data

```
300 DATA 14,4,148,194,16 : REM EARLY ONE MORNING
                                                   :rem 47
310 DATA 8779,Ø,8779,8779,8779,11Ø6Ø,13153,13153,1
    4764,11718,9854,8779                          :rem 241
320 DATA 8286,9854,6577,Ø,8779,Ø,8779,8779,8779,11
    Ø6Ø,13153,13153                               :rem 236
330 DATA 14764,11718,9854,8286,8779,Ø,Ø,Ø,9854,Ø,1
    1Ø6Ø,11718                                    :rem 193
340 DATA 13153,11Ø6Ø,8779,Ø,9854,Ø,11Ø6Ø,11718,131
    53,11Ø6Ø,8779,Ø                               :rem 161
350 DATA 8779,11Ø6Ø,13153,17557,16572,14764,13153,
    11718                                         :rem 2Ø5
360 DATA 11Ø6Ø,9854,8779,8286,8779,Ø,Ø,Ø,-1
                                                  :rem 1Ø5
```

Bass Data

```
300 DATA 1Ø,2,38,168,32 : REM CLASSIC BASE LINE
                                                  :rem 194
310 DATA 1465,Ø,Ø,Ø,1845,Ø,Ø,1955,2Ø71,2195,Ø,2195
    ,Ø,Ø,Ø,Ø                                        :rem 4
320 DATA 1465,1845,1955,2Ø71,2195,1644,1Ø97,2195,1
    465,1845,1955,2Ø71                             :rem 85
330 DATA 2195,2195,Ø,Ø,2195,Ø,Ø,2195,Ø,2195,2463,2
    195                                            :rem 57
340 DATA 732,Ø,923,Ø,1Ø97,Ø,1232,Ø,1465,Ø,1232,Ø,1
    Ø97,Ø,923,Ø                                   :rem 159
350 DATA 978,Ø,732,Ø,978,Ø,1Ø97,Ø,1232,Ø,1465,Ø,12
    32,Ø,1Ø97,Ø                                   :rem 18Ø
360 DATA Ø,1465,1383,1465,1Ø97,1232,1Ø97,1232,1465
    ,Ø,13Ø5,Ø,1232,Ø,1Ø97,Ø                        :rem 1Ø
370 DATA 1465,1465,1383,1465,1Ø97,1232,1Ø97,1232,1
    465,13Ø5,Ø,Ø,1232,1163,Ø,Ø                    :rem 165
380 DATA 1Ø97,1Ø97,Ø,1Ø97,Ø,1Ø97,Ø,1Ø97,1Ø97,Ø,Ø,1
    644,2195,Ø,Ø,691                              :rem 191
390 DATA 732,Ø,923,Ø,978,Ø,1Ø36,Ø,1Ø97,1232,1383,Ø
    ,1465,Ø,Ø,Ø,-1                                 :rem 56
```

Mozart Data

```
300 DATA 1Ø,2,85,168,64,2                         :rem 196
310 DATA 4927,Ø,Ø,3691,4927,Ø,Ø,3691,4927,3691,492
    7,62Ø7,7382,Ø,Ø,Ø                               :rem 6
320 DATA 6577,Ø,Ø,553Ø,6577,Ø,Ø,553Ø,6577,553Ø,465
    Ø,553Ø,3691,Ø,Ø,Ø                             :rem 244
330 DATA 4927,Ø,4927,Ø,Ø,62Ø7,553Ø,4927,4927,465Ø,
    465Ø,Ø,Ø,553Ø,6577,465Ø                        :rem 59
340 DATA 553Ø,4927,4927,Ø,Ø,62Ø7,553Ø,4927,4927,46
    5Ø,465Ø,Ø,Ø,553Ø,6577,465Ø                    :rem 217
```

```
350 DATA 4927,4927,4927,4650,4927,4927,6207,5530,6
    207,6207,7382,6577,7382,0,0,0         :rem 142
360 DATA 4143,0,0,5530,4927,4650,4143,4650,4927,0,
    6207,0,4927                           :rem 215
370 DATA -1                                :rem 18
```

Yankee Data

```
300 DATA{2 SPACES}8,1,20,166,16 : REM YANKEE DOODL
    E                                     :rem 190
310 DATA 13153,13153,14764,16572,13153,16572,14764
    ,9854                                 :rem 201
320 DATA 13153,13153,14764,16572,13153,0,12415,0
                                           :rem 66
330 DATA 13153,13153,14764,16572,17557,16572,14764
    ,13153                                :rem 250
340 DATA 12415,9854,11060,12415,13153,0,13153,0
                                           :rem 11
350 DATA 11060,12415,11060,9854,11060,12415,13153,
    0                                     :rem 207
360 DATA 9854,11060,9854,8779,8286,0,9854,0
                                          :rem 132
370 DATA 11060,12415,11060,9854,11060,12415,13153,
    11060                                 :rem 153
380 DATA 9854,13153,12415,14764,13153,0,13153,0,-1
                                          :rem 167
```

Susannah Data

```
300 DATA 15,3,20,169,32 : REM OH SUSANNAH :rem 115
310 DATA 0,0,2930,3288,3691,4389,4389,4927,4389,36
    91,2930,3288                           :rem 60
320 DATA 3691,3691,3288,2930,3288,0,2930,3288,3691
    ,4389,4389,4927                       :rem 221
330 DATA 4389,3691,2930,3288,3691,3691,3288,3288,2
    930,0,0,0                             :rem 144
340 DATA 3910,0,3910,0,4927,4927,0,4927,4389,4389,
    3691,2930                             :rem 146
350 DATA 3288,0,2930,3288,3691,4389,4389,4927,4389
    ,3691,2930,3288                       :rem 229
360 DATA 3691,3691,3288,3288,2930,0,0,0,-1 :rem 31
```

Knock Data

```
300 DATA 4,4,0,192,128 : REM THE END        :rem 2
310 DATA 3288,0,0,2463,2195,2463,2765,0,0,2463,0,0
    ,0,0,0,3104,0,0,3288,0,0,-1           :rem 156
```

Data Storage

The tunes have been stored in a special format. The first DATA statement sets various parameters that stay in effect for the whole song. The remaining DATA statements control the individual notes.

The tempo of a piece of music determines how quickly the notes are played. In our simple player program, the tempo is controlled by the duration of each note. The variable T1 tells how many sixtieths of a second should be placed between the attack and release of a note. Variable T2 indicates how many sixtieths of a second come between the release of the note and the attack of the next note. The values for T1 and T2 are the first two numbers in the first DATA statement. Making these values larger or smaller will decrease or increase the tempo.

The next two values in the first DATA statement are read into the variables E1 and E2, which control the envelope. These variables correspond to the locations AD (attack and decay) and SR (sustain and release). Remember that each parameter has a range from 0 to 15, and the first one of each pair must be multiplied by 16. An attack rate of 2 with a decay rate of 3 means that the third number in the DATA statement would be 2*16+3, which equals 35.

The last number is for the waveform; it can be 16, 32, 64, or 128. If it is 64, which designates the pulse wave, an additional number will be read, to specify the pulse width.

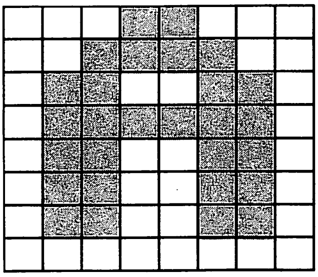
The remaining DATA statements contain note information. There is one frequency number for each note. Notice how the AND operator is used in line 160 to separate the frequency number into low and high bytes. This method is simpler than using the integer function, but because the AND operator works only on numbers up to 32767, the frequency range is not as great.

A frequency number of 0 indicates a rest. A rest has the duration of a normal note, but no tone is produced. Negative numbers flag the end of a tune.

Redefining the Character Set

When you look at the numbers 24, 60, 102, 126, 102, 102, 102, and 0, they do not seem to have any special meaning. Even when you represent them in binary form (00011000, 00111100, 01100110, 01111110, 01100110, 01100110, 01100110, and 00000000), there does not seem to be any significance to them. But if you look at them stacked vertically, you will notice that these numbers are indeed special.

```
00011000
00111100
01100110
01111110
01100110
01100110
01100110
00000000
```



The bit patterns of these numbers form the letter A. Where a bit is set, the character color shows. Where a bit is clear, the background color shows through. The eight rows of bytes and eight columns of bits create an 8 by 8 matrix which is sufficient to define the image for any character. Each character that can be displayed on the screen has its own set of eight numbers. Collectively, these character definitions form what is called a *character set*. By changing the numbers for a character, you can change its shape to create an entirely different image. This section shows you how to redefine the characters in a character set.

To set up a character set in memory so that it can be changed, you must follow a certain procedure. To illustrate that procedure, enter this program, SAVE it to disk or tape, LIST it, and then RUN it. If you RUN the program a second time, it will appear as if nothing is happening. This is because the character set has already been moved during the first run. If you wish to see the characters change a second time, turn off your computer then turn it back on and reLOAD the program.

Moveset

```
10 POKE 56,48 : CLR : REM RESERVE MEMORY   :rem 193
20 POKE 53272,(PEEK(53272)AND240)OR12 : REM SELECT
    CHARACTER SET                          :rem 78
30 POKE 56334,PEEK(56334)AND254 : REM DISABLE KEYB
   OARD                                    :rem 15
```

```
40 POKE 1,PEEK(1)AND251  : REM ACCESS STANDARD CHAR
   ACTERS                                      :rem 3
50 K=0 : REM START AT BEGINNING OF NEW CHARACTER S
   ET                                        :rem 231
60 POKE 12288+K,PEEK(53248+K) : K=K+1 : IF K<512 G
   OTO 60                                     :rem 30
70 POKE 1,PEEK(1)OR4 : POKE 56334,PEEK(56334)OR1 :
   END                                       :rem 103
```

At first, all of the characters on the screen will be changed to very unusual shapes. Then, one by one, each character will be defined, until the first 64 characters have been moved into memory starting at location 12288. Since the cursor is not one of these characters, it is not properly defined, but this will not affect the operation of the computer. Also, the amount of free memory has been greatly reduced, which can be verified by using the free memory function. Advanced methods beyond the scope of this book let you get around this limitation.

To redefine a single character, such as the letter A, you must first locate its set of eight bytes. The location of the defining bytes for a character is determined by the screen POKE value for the character. The first eight bytes define character 0, the next eight bytes define character 1, and so on. Since the letter A appears on the screen when you POKE screen memory with the value 1, the definition for this character must be stored in the second set of eight bytes, from locations 12288+8 to 12288+15. Enter the following two lines to turn the letter A upside down.

POKE 12296,102:POKE 12297, 102:POKE 12298,102:POKE 12299,126
POKE 12300,102:POKE 12301,60:POKE 12302,24

Only the letter A was inverted; none of the other characters was affected. Now, wherever an A appears on the screen, it is upside down. Try POKEing some other values into these locations to see how the eight bytes correspond with the character definition.

With a few modifications to the original program, you can invert all of the characters.

Invertset

```
10 POKE 56,48 : CLR : REM RESERVE MEMORY  :rem 193
20 POKE 53272,(PEEK(53272)AND240)OR12 : REM SELECT
    CHARACTER SET                             :rem 78
```

```
30 POKE 56334,PEEK(56334)AND254 : REM DISABLE KEYB
   OARD                                        :rem 15
40 POKE 1,PEEK(1)AND251 : REM ACCESS STANDARD CHAR
   ACTERS                                       :rem 3
50 K=Ø : REM START AT BEGINNING OF NEW CHARACTER S
   ET                                         :rem 231
6Ø J=Ø                                         :rem 29
7Ø POKE 12288+K+J,PEEK(53248+7+K-J) : J=J+1 : IF J
   <8 GOTO 7Ø                                  :rem 11
8Ø K=K+8 : IF K<512 GOTO 6Ø                    :rem 37
9Ø POKE 1,PEEK(1)OR4 : POKE 56334,PEEK(56334)OR1 :
    END                                       :rem 1Ø5
```

Of course, it is very difficult to read anything that is upside down on the screen. The fastest way to restore the character set is to press RUN/STOP and RESTORE, but you will have to run the set-moving program again before redefining any more characters.

There are many practical applications for redefined characters. The first use is to create stylized lettering. It adds a nice touch to a program when instructions and scores are printed in something other than the normal characters. Here are two examples of different lettering styles. The first program gives the letters a fancy look, and the second gives the appearance of scanner-readable printing (also known as *computer type*), such as is found on the lower-left margin of checks.

Fancyset

```
1Ø POKE 56,48 : CLR : REM RESERVE MEMORY  :rem 193
2Ø POKE 53272,(PEEK(53272)AND24Ø)OR12 : REM SELECT
    CHARACTER SET                              :rem 78
3Ø K=12288                                    :rem 24Ø
4Ø READ P : POKE K,P : K=K+1 : IF K<128ØØ GOTO 4Ø
                                               :rem 81
5Ø END                                         :rem 6Ø
8ØØ DATA Ø,6Ø,1Ø2,11Ø,11Ø,96,62,Ø,Ø,24,6Ø,1Ø2,1Ø2,
    126,1Ø2,Ø,Ø,252,1Ø2,124                   :rem 187
8Ø1 DATA 1Ø2,1Ø2,252,Ø,Ø,62,1Ø2,96,96,1Ø2,6Ø,Ø,Ø,2
    52,1Ø2,1Ø2,1Ø2,1Ø2,252,Ø,Ø                 :rem 82
8Ø2 DATA 254,98,12Ø,96,98,254,Ø,Ø,254,98,12Ø,96,96
    ,24Ø,Ø,Ø,124,196,192,222,2Ø4               :rem 16
8Ø3 DATA 124,Ø,Ø,1Ø2,1Ø2,126,1Ø2,1Ø2,1Ø2,Ø,Ø,126,2
    4,24,24,24,255,Ø,Ø,3Ø,12,12               :rem 118
8Ø4 DATA 12,2Ø4,12Ø,Ø,Ø,247,1Ø8,12Ø,12Ø,1Ø8,246,3,
    Ø,24Ø,96,96,96,98,254,Ø,Ø,198              :rem 35
```

```
805 DATA 238,254,214,198,198,Ø,Ø,231,118,126,11Ø,1
    Ø2,231,Ø,Ø,6Ø,1Ø2,1Ø2,1Ø2,1Ø2              :rem 4
806 DATA 6Ø,Ø,Ø,252,1Ø2,1Ø2,124,96,24Ø,Ø,Ø,6Ø,1Ø2,
    1Ø2,1Ø2,1Ø8,54,1,Ø,252,1Ø2                :rem 87
807 DATA 1Ø2,124,1Ø8,246,3,Ø,62,96,6Ø,6,6,124,Ø,Ø,
    255,153,24,24,24,6Ø,Ø,Ø,1Ø2               :rem 163
808 DATA 1Ø2,1Ø2,1Ø2,1Ø2,6Ø,Ø,Ø,1Ø2,1Ø2,1Ø2,6Ø,6Ø,
    24,Ø,Ø,198,198,214,254,238                :rem 11Ø
809 DATA 198,Ø,Ø,231,1Ø2,6Ø,6Ø,1Ø2,231,Ø,Ø,231,98,
    52,24,24,6Ø,Ø,Ø,254,14Ø,24                :rem 1Ø9
81Ø DATA 48,98,254,Ø,Ø,3Ø,24,24,24,24,3Ø,Ø,Ø,64,96
    ,48,24,12,6,Ø,Ø,12Ø,24,24,24              :rem 21Ø
811 DATA 24,12Ø,Ø,Ø,16,56,1Ø8,198,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,
    Ø,255,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,24                :rem 124
812 DATA 24,24,24,Ø,24,Ø,Ø,1Ø2,1Ø2,1Ø2,Ø,Ø,Ø,Ø,Ø,1
    Ø2,255,1Ø2,1Ø2,255,1Ø2,Ø,24               :rem 1Ø4
813 DATA 62,96,6Ø,6,124,24,Ø,Ø,1Ø2,1Ø8,24,48,1Ø2,7
    Ø,Ø,28,54,28,56,111,1Ø2,59                :rem 134
814 DATA Ø,56,56,24,48,Ø,Ø,Ø,Ø,12,24,24,24,24,24,2
    4,12,48,24,24,24,24,24,24,48              :rem 213
815 DATA Ø,1Ø2,6Ø,255,6Ø,1Ø2,Ø,Ø,Ø,24,24,126,24,24
    ,Ø,Ø,Ø,Ø,Ø,Ø,Ø,24,24,48,Ø,Ø               :rem 1Ø7
816 DATA Ø,126,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,24,24,Ø,Ø,6,12,24
    ,48,96,64,Ø,Ø,6Ø,1Ø2,11Ø,118              :rem 159
817 DATA 1Ø2,6Ø,Ø,Ø,24,56,24,24,24,6Ø,Ø,Ø,6Ø,1Ø2,1
    2,48,98,126,Ø,Ø,126,76,24,12              :rem 2Ø9
818 DATA 1Ø2,6Ø,Ø,Ø,12,28,6Ø,1Ø8,126,12,Ø,Ø,126,98
    ,124,6,1Ø2,6Ø,Ø,Ø,6Ø,96,124               :rem 16Ø
819 DATA 1Ø2,1Ø2,6Ø,Ø,Ø,126,7Ø,12,24,48,48,Ø,Ø,6Ø,
    1Ø2,6Ø,1Ø2,1Ø2,6Ø,Ø,Ø,6Ø,1Ø2              :rem 181
82Ø DATA 62,6,12,56,Ø,Ø,Ø,24,24,Ø,24,24,Ø,Ø,Ø,24,2
    4,Ø,24,24,48,6,12,24,48,24,12             :rem 232
821 DATA 6,Ø,Ø,Ø,126,Ø,Ø,126,Ø,Ø,96,48,24,12,24,48
    ,96,Ø,Ø,6Ø,1Ø2,12,24,Ø,24,Ø               :rem 134
```

Computerset

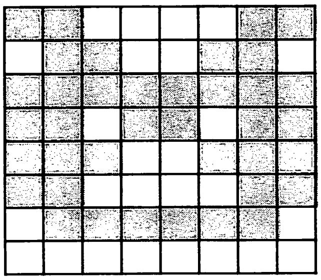
```
1Ø POKE 56,48 : CLR : REM RESERVE MEMORY  :rem 193
2Ø POKE 53272,(PEEK(53272)AND24Ø)OR12 : REM SELECT
   CHARACTER SET                           :rem 78
3Ø K=12288                                 :rem 24Ø
4Ø READ P : POKE K,P : K=K+1 : IF K<128ØØ GOTO 4Ø
                                           :rem 81
5Ø END                                     :rem 6Ø
8ØØ DATA 127,99,111,111,111,96,127,Ø,63,51,51,127,
    115,115,115,Ø,126,1Ø2,1Ø2                 :rem 75
8Ø1 DATA 127,1Ø3,1Ø3,127,Ø,127,1Ø3,1Ø3,96,99,99,12
    7,Ø,126,1Ø2,1Ø2,119,119,119               :rem 198
```

```
802 DATA 127,Ø,127,96,96,127,112,112,127,Ø,127,96,
    96,127,112,112,112,Ø,127,99               :rem 208
803 DATA 96,111,1Ø3,1Ø3,127,Ø,115,115,115,127,115,
    115,115,Ø,127,28,28,28,28,28              :rem 237
8Ø4 DATA 127,Ø,12,12,12,14,14,11Ø,126,Ø,1Ø2,1Ø2,1Ø
    8,127,1Ø3,1Ø3,1Ø3,Ø,48,48,48              :rem 199
8Ø5 DATA 112,112,112,126,Ø,1Ø3,127,127,119,1Ø3,1Ø3
    ,1Ø3,Ø,1Ø3,119,127,111,1Ø3                :rem 1Ø3
8Ø6 DATA 1Ø3,1Ø3,Ø,127,99,99,1Ø3,1Ø3,1Ø3,127,Ø,127
    ,99,99,127,112,112,112,Ø,127              :rem 246
8Ø7 DATA 99,99,1Ø3,1Ø3,1Ø3,127,7,126,1Ø2,1Ø2,127,1
    19,119,119,Ø,127,96,127,3,115              :rem 53
8Ø8 DATA 115,127,Ø,127,28,28,28,28,28,28,Ø,1Ø3,1Ø3
    ,1Ø3,1Ø3,1Ø3,1Ø3,127,Ø,1Ø3                :rem 123
8Ø9 DATA 1Ø3,1Ø3,1Ø3,111,62,28,Ø,1Ø3,1Ø3,1Ø3,111,1
    27,127,1Ø3,Ø,115,115,115,62               :rem 152
81Ø DATA 1Ø3,1Ø3,1Ø3,Ø,1Ø3,1Ø3,1Ø3,127,28,28,28,Ø,
    127,1Ø2,1Ø8,24,55,1Ø3,127,Ø               :rem 155
811 DATA 3Ø,24,24,24,24,24,3Ø,Ø,64,96,48,24,12,6,3
    ,Ø,12Ø,24,24,24,24,24,12Ø,Ø               :rem 151
812 DATA Ø,8,28,54,99,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,255,Ø,Ø,Ø,
    Ø,Ø,Ø,Ø,Ø,Ø,56,56,24,24,Ø,24              :rem 146
813 DATA 24,Ø,238,238,68,68,Ø,Ø,Ø,Ø,1Ø2,255,1Ø2,1Ø
    2,255,1Ø2,Ø,Ø,24,62,96,6Ø,6               :rem 167
814 DATA 124,24,Ø,Ø,1Ø2,1Ø8,24,48,1Ø2,7Ø,Ø,28,54,2
    8,56,111,1Ø2,59,Ø,24,24,24                :rem 118
815 DATA Ø,Ø,Ø,Ø,Ø,3Ø,24,24,56,56,56,62,Ø,12Ø,24,2
    4,28,28,28,124,Ø,Ø,1Ø2,6Ø,255             :rem 249
816 DATA 6Ø,1Ø2,Ø,Ø,Ø,24,24,126,24,24,Ø,Ø,Ø,Ø,Ø,Ø,
    Ø,24,24,48,Ø,Ø,Ø,126,Ø,Ø,Ø                 :rem 44
817 DATA Ø,Ø,Ø,Ø,Ø,Ø,24,24,Ø,3,6,12,24,48,96,64,Ø,
    127,99,99,99,99,99,127,Ø,56               :rem 21Ø
818 DATA 24,24,24,62,62,62,Ø,127,3,3,127,96,96,127
    ,Ø,126,6,6,127,7,7,127,Ø,112              :rem 246
819 DATA 112,112,112,119,127,7,Ø,127,96,96,127,3,3
    ,127,Ø,124,1Ø8,96,127,99,99               :rem 227
82Ø DATA 127,Ø,127,3,3,31,24,24,24,Ø,62,54,54,127,
    119,119,127,Ø,127,99,99,127               :rem 2Ø4
821 DATA 7,7,7,Ø,6Ø,6Ø,6Ø,Ø,6Ø,6Ø,6Ø,Ø,6Ø,6Ø,6Ø,Ø,
    6Ø,6Ø,24,48,6,12,24,48,24,12              :rem 2Ø6
822 DATA 6,Ø,Ø,126,Ø,Ø,126,Ø,Ø,Ø,96,48,24,12,24,48
    ,96,Ø,127,99,3,31,28,Ø,28,Ø               :rem 166
```

Again, these programs redefine only the first 64 characters, so characters such as the graphics symbols and reverse characters will not be defined.

You may want to go beyond the variations of characters and create new ones. This would be done to accommodate some alphabets which use letters and symbols not in the normal character set. You might redefine some of the less commonly used punctuation marks or graphics symbols to get the additional characters.

You may also choose to completely forget about letters and other symbols, and define some entirely new shapes. Consider the Hurkle game, in which the ball character was used for the falling Hurkle. A character could be easily redefined to have arms or legs. Just draw the shape on an 8 by 8 grid, add up the bit values, and use the results in POKE statements. See the section on binary numbers for more detailed information. Also, there is a binary-to-decimal conversion program on page 78 of the *Commodore 64 User's Guide*. The example given here shows how to redefine the letter A so that it is a funny face. Remember to run the program above that moves the character set, before entering the POKE statements.



```
11000011 128+64+2+1=195
01100110 64+32+4+2=102
11111111 =255
11011011 128+64+16+8+2+1=219
11100111 128+64+32+4+2+1=231
11000011 128+64+2+1=195
01111110 64+32+16+8+4+2=126
00000000 =0
```

```
POKE 12296,195:POKE 12297,102:POKE 12298,255:POKE
  12299,219
POKE 12300,231:POKE 12301,195:POKE 12302,126:POKE
  12303,0
```

After you've entered these two lines, the letter A will have been changed to a little face character. You may have to change the screen colors to get the best effect.

There is one precaution you should take when creating the image for a new character. In the definition for the funny face, the bits in each row are organized so that no single bit is by itself. The bits occur in groups, with at least two bits in each group. This is to avoid a phenomenon called *chroma noise*, which happens when one bit is displayed without any horizontally adjacent bits. For an example of chroma noise, consider the bit pattern 01010101, which is decimal 85.

```
POKE 12288,85:POKE 12289,85:POKE 12290,85:POKE
  12291,85
POKE 12292,85:POKE 12293,85:POKE 12294,85:POKE
  12295,85
```

The result is a square character which alternates between two colors, depending on the column in which it is printed. It is best to avoid this effect by always using bit groups at least two bits wide in each row. The problem does not occur vertically.

Sometimes an 8 by 8 matrix is not large enough to represent a desired shape, so two or more characters must be used. The following lines redefine the letters A and B to look like a ship.

```
POKE 12296,12:POKE 12297,12:POKE 12298,12:POKE
  12299,12
POKE 12300,255:POKE 12301,127:POKE 12302,63:POKE
  12303,15
POKE 12304,192:POKE 12305,192:POKE 12306,192:POKE
  12307,220
POKE 12308,255:POKE 12309,254:POKE 12310,253:POKE
  12311,240
```

Multiple redefined characters are often used to create background scenes, such as different types of terrains.

Multicolor Mode

Now that you know how to redefine a character set, there is one last feature of character graphics to be introduced. Normally, a character image can be only one color, with the background color showing through gaps in the image.

The extended background color mode gives you three additional background colors, but you are still limited to only one color for every character image. It is possible to get more than one color, though, by playing with the chroma noise effect. RUN the character set redefining program (Moveset without line 5) from the last section before entering the following lines.

```
POKE 12296,85:POKE 12297,85:POKE 12298,85:POKE
  12299,85
```

```
POKE 12300,170:POKE 12301,170:POKE 12302,170:POKE
  12303,170
```

The numbers 85 and 170 have bit patterns 01010101 and 10101010. The top half of what used to be the letter A should now be one color, and the bottom half should be another color. Unfortunately, this method is not very useful, because the colors are very weak and change with the position of the character on the screen. Therefore, the graphics chip supports another special mode, called *multicolor* mode, which lets each character contain up to three image colors, not just one. Most importantly, you have full control over each color.

Clear the screen, make sure the cursor is the normal light blue color (press Commodore-7 if not), fill about half of the top row with the redefined letter A, and then move the cursor down one row. Now turn on the multicolor mode with the following statement:

```
POKE 53270,PEEK(53270)OR16
```

Set the cursor color to white and enter these two lines (be careful; you will be entering these lines without being able to see the characters on the screen):

```
POKE 53282,13
POKE 53283,7
```

Now you can change the individual colors within the same character. Locations 53282 and 53283 are for background colors 1 and 2, respectively. The full range of colors, 0 to 15, can be POKEd into these locations.

As in the normal mode, the bit patterns determine which parts of the character display which colors, but since there are now multiple colors, the bit patterns must be interpreted a little differently. The key is to separate the bits into pairs. A byte can be divided into four pairs of bits. Each bit-pair indicates which color is displayed for that point.

Bit Pattern	Color
00	background color 0 (location 53281)
01	background color 1 (location 53282)
10	background color 2 (location 53283)
11	color memory (locations 55296 to 56295)

Enter the following lines to make the character for letter A display all four colors.

```
POKE 12296,0:POKE 12297,0
POKE 12298,85:POKE 12299,85
POKE 12300,170:POKE 12301,170
POKE 12302,255:POKE 12303,255
```

Because bit-pairs are used, each byte can hold information for only four points instead of the usual eight. To compensate for the reduced number of points, each point is twice the normal width, so multicolor mode characters appear to be the same size as normal characters.

As an added feature, you can select which characters are to be displayed in multicolor mode. For each character, the bits will be interpreted in the multicolor mode only if the color memory location for that character contains a value greater than seven. This is why we suggested that you change the cursor to the color white. White corresponds to color 2, so all white characters are displayed normally. This does create one restriction, though. When multicolor mode is enabled, colors specified by color memory are limited to the range from 0 to 7. Colors 8 to 15 select the multicolor mode for that character, but the actual colors will be the same as those from 0 to 7. A 10 (8+2) in memory location 55296 means that the top left character should be displayed in multicolor mode, and that the color for bit pattern 11 should be 2. So, for every color memory location, use colors from 0 to 7, and add 8 if you want the character to be displayed in multicolor mode, using the additional colors from background color locations 2 and 3.

To turn off the multicolor mode, use the following statement:

```
POKE 53270,PEEK(53270)AND239
```

Chapter 10

Input

Input

The Concept of Input

Input occurs when information that is outside the computer is brought inside. Examples of input that we have observed so far include typing programs into memory from the keyboard, loading programs from disk or tape, and using a joystick to tell a program which way to move a screen character. The converse, *output,* is performed when information inside the computer is sent outside, as when information is printed on the screen or stored to disk or tape.

Now let's look in more detail at some ways input and output allow you to interact with a program while it's running, to provide it with information and get information in return. A program communicates with the user via the screen, which is an output device. You already have communicated with a program via the joystick, but the joystick is an input device that is very limited in its range. Very little information can be given to the computer using the joystick. The keyboard, on the other hand, can supply a lot of information. It can generate a lot of characters, but more importantly, the characters can be combined to form numbers and words. The keyboard is an excellent input device.

In order to better understand the keyboard's value, we must ask some new questions. How can a program input information from the keyboard? And when the program gets the information, how does it handle it? How is the information stored?

First, consider the characteristics of the incoming information. Let's say that the user is supposed to specify some numeric value. The program has no way of knowing what that number might be. And every time the user specifies a number, it could be different from the previous one. Changing values of undetermined range is a job for variables. Variables are used to control many aspects of a program, from setting things like color and pitch to counting loops. If there is a way for a user to change the values of some variables while a program is running, that could have a profound effect on the program. The program will be more flexible, useful, and interactive. This chapter introduces two statements which allow the user to set the values of variables while a program is running, without the user having to know anything about BASIC.

The GET Statement

The GET statement provides a means for directly transferring a keystroke to a variable. If you press any digit from 0 to 9 and then use GET, that value, from 0 to 9, will be assigned to a designated variable. Give it a try.

```
10 GET A                                    :rem 130
20 PRINT A; "{2 SPACES}";                   :rem 234
30 GOTO 10                                  :rem 253
```

As you pressed the number keys, the corresponding numbers were printed on the screen by line 20. But most of the screen was filled with the value 0. This is because a keypress can be fetched by the GET statement only once. Because the computer operates so quickly, it is impossible to type fast enough to press a key every time GET is executed, so most of the time the variable is assigned the default value of 0.

Try pressing a nonnumeric key, such as the letter A, while this program is running. The program will stop with a SYNTAX ERROR. You cannot specify a variable or nonnumeric key using this form of GET; only a numeral is acceptable. For example, if you know that the program contains the variable B, and that B has a value of 6, typing B on the keyboard will not cause the variable indicated by GET to be assigned the value 6. Remember, the GET statement is supposed to be used without the user having to know anything about the program.

The syntax for GET is the keyword GET followed by a variable list. Unfortunately, the computer works so quickly that trying to get a single keypress hardly works, let alone trying to get several. With the way that GET is set up, the user can specify only a value from 0 to 9, which is a rather small range.

The Commodore 64 keyboard has a temporary storage area, called a *buffer,* which can partially solve the problem of getting several keys. You can type on the keyboard at any time, even when the computer does not need keyboard input. The computer will remember the keystrokes, although they will not show on the screen. The keystroke information is temporarily put in the buffer. When the GET statement is later executed, it will fetch the keys stored in the buffer before using new keys typed at the keyboard. The following demonstration program waits ten seconds while you type digit keys.

```
10 A=TI                                      :rem 124
20 IF TI-A < 600 GOTO 20                     :rem 105
30 GET KEY                                    :rem 44
40 PRINT KEY                                 :rem 218
50 K=K+1 : IF K<12 GOTO 30                   :rem 227
```

The computer can remember up to ten keystrokes, and the order in which they were pressed. Keys pressed after the buffer is full are forgotten.

The keyboard buffer makes it easier to get multiple keys. Unfortunately, it still does not solve the problem of being able to get values other than the numbers 0 through 9.

Another problem with the above program is that if somebody else were to run it, there would be no indication that something was supposed to be typed on the keyboard. Rather than staying in an infinite loop, the program should inform the user that input is needed from the keyboard. Add this line to the program:

```
15 PRINT "PLEASE TYPE SOME DIGITS"            :rem 43
```

From a user's standpoint, being instructed what to do is at least better than typing RUN and watching the computer "play dead." A good program always keeps the user informed about what is happening, and prints some sort of prompt when keyboard input is needed.

Unlike the other statements that have been introduced thus far, GET can only be used in a program. If you try to use the GET statement in the immediate mode, the ILLEGAL DIRECT error will be printed.

Summary

- The syntax for the GET statement is the keyword GET and a variable list.
- When GET is executed, the computer checks whether a key has been pressed. If so, the value of that key is assigned to the indicated variable. This is repeated for each variable in the list.
- If no key has been pressed, the variable is assigned the value 0.

- In the above examples, pressing a key other than one of the number keys causes a SYNTAX ERROR. GET cannot be used to assign the value of one variable to another; only constant numbers will work.
- The keyboard has a *buffer* which can store up to ten keystrokes in the order they were entered. Keys are ignored if pressed after the buffer is full.
- It is good practice to have a program print a prompting message when keyboard input is needed.
- The GET statement can only be used in the program mode; an ILLEGAL DIRECT error will be generated if it is used in the immediate mode.

The INPUT Statement

The INPUT statement is simpler and safer to use than GET. The main problem with the GET statement is that it can input only one key at a time; INPUT lets the user enter a complete number.

```
10 REM COLOR CHANGE PROGRAM                    :rem 130
20 INPUT A                                      :rem 51
30 PRINT "BORDER COLOR ="; A                    :rem 42
40 POKE 53280,A                                  :rem 2
```

When you run this program, the first thing you notice is that a question mark is printed. The INPUT statement automatically prints a question mark to let you know that the computer is waiting for your input.

Every time you type a key, it appears on the screen. If you make a mistake, you can press the DEL key to correct it. When you are through typing digits, press the RETURN key. Up until this point, everything has been done by the INPUT statement. Only when RETURN is pressed does the program continue to line 30. INPUT is a very powerful statement.

The INPUT statement also has built-in error handling. If you try to enter letters instead of digits in the above program, the message REDO FROM START will be printed as soon as you press RETURN. This message means "something was wrong—start all over again." The question mark prompt will appear, and you will have to try again, this time typing digits instead of letters.

If you type nothing and just press RETURN in response to an INPUT statement, the indicated variable will not be assigned, but will retain its previous value. To see this, add the following line to the program and run it a few times, sometimes entering a number, sometimes just pressing RETURN.

```
10 A=9                                          :rem 24
```

The INPUT statement supports a variable list. If you use an INPUT to assign two variables, two numbers will have to be entered when INPUT is executed. Our example program is modified to input two variables.

```
10 A=9 : B=13                                   :rem 53
20 INPUT A,B                                   :rem 161
30 PRINT "BORDER COLOR ="; A                    :rem 42
40 POKE 53280,A                                  :rem 2
50 PRINT "BACKGROUND COLOR ="; B                :rem 79
60 POKE 53281,B                                  :rem 6
```

After you see the question mark, type two numbers, separated by a comma, and then press RETURN. Both of the variables will be assigned new values.

Care should be taken when the comma is used in responses for INPUT. In typing single numbers, no comma is necessary. If you type a number followed by a comma and nothing else, the computer will interpret the comma as meaning that a second number has been typed, and upon finding no second number, will assume the value 0. Try this with the first example. If you type nothing but a comma and RETURN for an INPUT, the computer will again think that two numbers were entered. But when it finds no numbers, it will assume the value 0 for both numbers. Try this with the second example.

If you don't type any numbers and just press RETURN, no new values will be assigned. But what if you type only one number? Perhaps the person using the program doesn't know that two values need to be entered when the question mark prompt appears. If only one number is typed and RETURN is pressed, the computer will print another prompt, except that this time it is two question marks. The two question marks mean "more input is needed." The second number is entered, RETURN is pressed, and the value is assigned to the second variable. If there are even more variables in the list after INPUT, the double question mark will again be printed and the whole process repeated. If at any time a bad value is entered, and the REDO FROM START error message appears, all of the values will have to be reentered, starting with the first one.

There is one difference in the handling of input after a prompt of ? and ??. The first time, when just one question mark is printed, pressing the RETURN key will cause all the variables to retain their current values. But if RETURN is pressed after a prompt of two question marks, the variable to be assigned will be assigned the value of 0, and INPUT will go on to the next variable in the list.

If more numbers are typed than are needed to assign all of the variables in the variable list, the extra numbers are ignored and a warning message of EXTRA IGNORED is printed. Program execution will continue to the next statement; unlike the REDO FROM START error, this error does not cause the statement to restart. The message is simply a warning to let the user know that not all of the input was needed or used.

One of the nice things about INPUT is that it automatically prints the question mark as a prompt. However, a question mark does not tell the user a whole lot, and it is always a good idea to print a little more information before getting input. The prompt should indicate two things: how many numbers are to be entered, and what they are needed for. This could be accomplished by adding one statement to the program:

```
20 PRINT "BORDER AND BACKGROUND COLORS"; : INPUT A
   ,B                                        :rem 42
```

Notice the semicolon at the end of the character string. Using the semicolon here means that the question mark will be printed at the end of the message. Try it out.

Since prompt messages are used so often with INPUT, the statement syntax has been designed to accommodate a character string which will be printed before the question mark prompt. Other than that, the INPUT statement operates just as seen earlier. Try this new line 20.

```
20 INPUT "BORDER AND BACKGROUND COLORS"; A,B
                                             :rem 99
```

The program operates identically to the previous version. The character string is just a convenience, and it is up to you when you want to use it. When a character string is used, it must be followed by a semicolon. If you would like to experiment a little further with INPUT, you can save yourself from having to type RUN all the time by adding one more line to the program.

```
70 GOTO 20                                           :rem 2
```

The RUN/STOP key cannot be used to stop a program while it is in the middle of an INPUT statement. In such a case, press RUN/STOP and RESTORE.

As with the GET statement, the ILLEGAL DIRECT error will occur if you try to use INPUT in the immediate mode.

We now see that the syntax for INPUT is the keyword INPUT, an optional character string enclosed in quote marks and ended with a semicolon, and then a variable list. If the character string is used, the semicolon has to be present, or a SYNTAX ERROR will occur.

Summary

- The syntax of the INPUT statement is the keyword INPUT, an optional character string terminated with a semicolon, and a variable list.
- If the character string is used, it must be in quotes, and it must be followed by a semicolon, or a SYNTAX ERROR will occur.
- When INPUT is executed, the character string is printed first, if one was provided.
- The INPUT statement then prints a question mark, and waits for the user to type one or more numbers and press RETURN. During this time the program is executing just the INPUT statement.
- The numbers entered are assigned to the variables in the variable list.
- If not enough numbers are entered, a second prompt, consisting of two question marks, is printed, and the computer waits for more numbers to be entered.
- If too many numbers are entered, a warning message that says EXTRA IGNORED is printed, and only the first few values are assigned to variables. The extra numbers are not used.

- Including illegal characters in a line causes the error message REDO FROM START to be printed. All of the numbers have to be typed again, no matter how many variables had already been assigned.
- Merely pressing RETURN in response to the ? prompt cancels the assignment of any variables. All variables retain their value.
- Pressing RETURN after a ?? prompt causes the current variable to be assigned the value 0.
- Using two commas together can make the computer assume a default value of 0 for one of the variables, so caution is advised.
- The only way to abort a program while an INPUT statement is being executed is to press RUN/STOP and RESTORE.
- The INPUT statement can be used only in the program mode, not in the immediate mode.

Sound Experiment

Here is a demonstration program that serves two purposes. The program shows you how to use GET and INPUT, and it makes it easy for you to explore the many envelope and waveform combinations that can be produced by the Commodore 64.

Sound Demonstration

```
100 PRINT "SOUND DEMONSTRATION" : PRINT   :rem 153
110 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:C
    T=54276:PW=54275                      :rem 249
120 POKE MV,15 : PRINT "SELECT A WAVEFORM":rem 106
130 PRINT "1 TRIANGLE" : PRINT "2 SAWTOOTH" : PRIN
    T "3 PULSE" : PRINT "4 NOISE"          :rem 38
140 GET K : IF K<1 OR K>4 GOTO 140        :rem 109
150 PRINT : WF=2↑K*8 : IF WF<>64 GOTO 190 :rem 145
160 PRINT "SELECT PULSE WIDTH (KEY 1-8)"   :rem 1
170 GET K : IF K<1 OR K>8 GOTO 170        :rem 119
180 PRINT : POKE PW,K                     :rem 173
190 PRINT "ENTER ATTACK RATE (" A ")"; : INPUT A :
     IF A<0 OR A>15 GOTO 190              :rem 184
200 PRINT "ENTER DECAY RATE (" D ")"; : INPUT D :
    {SPACE}IF D<0 OR D>15 GOTO 200         :rem 98
210 PRINT "ENTER SUSTAIN LEVEL (" S ")"; : INPUT S
     : IF S<0 OR S>15 GOTO 210            :rem 173
220 PRINT "ENTER RELEASE RATE (" R ")"; : INPUT R
    {SPACE}: IF R<0 OR R>15 GOTO 220       :rem 57
```

```
230 POKE AD,A*16+D : POKE SR,S*16+R : RESTORE
                                             :rem 175
240 READ F : IF F<0 THEN PRINT : GOTO 120   :rem 53
250 POKE FL,F-256*INT(F/256) : POKE FH,F/256 : POK
    E CT,WF+1 : T=TI                        :rem 206
260 IF TI-T<30 GOTO 260                     :rem 181
270 POKE CT,WF : T=TI                       :rem 144
280 IF TI-T<30 GOTO 280                     :rem 185
290 GOTO 240                                :rem 106
300 DATA 268,337,401,536,675,803,1072,1351,1607,21
    45,2703,3215,4291                       :rem 255
310 DATA 5407,6430,8583,10814,12860,17167,21629,25
    721,34334,-1                             :rem 24
```

When you run this program, you will see a numbered list of the four waveforms. After you have chosen which waveform you want, just press the appropriate number key. The GET statement is being used here, so no RETURN is needed. If you select the pulse waveform, you will also be asked to press one of the keys from 1 through 8 to set the pulse width. Again, all you have to do is press the number key.

After the waveform is all set, you need to enter values for the envelope. The valid numbers range from 0 to 15, and you must press RETURN after each number. The INPUT statement is being used to enter these values.

At this point, all of the parameters have been specified, and the computer will start playing notes that span eight octaves. When it is done, you can go through the waveform and envelope selection process all over again. However, for each of the four envelope parameters, the last value will appear in parentheses, and you can choose that by pressing RETURN without typing any numbers. Remember, INPUT does not assign the variable if you only press RETURN.

This is not a major program, but it is much better than entering a lot of POKE statements in the immediate mode. Often a simple little program like this one can help you learn more about the special features of your Commodore 64.

Chapter 11

String Variables and Functions

String Variables and Functions

String Variables

The INPUT statement made keyboard input a lot easier, but the way we have used it so far restricts input to numbers. You will often want to have a user enter alphabetic information while a program is running. You have seen that an attempt to type letters in response to INPUT generates the REDO FROM START error. This happened because the INPUT statement tried to assign the character string to the indicated variable and found a *type mismatch,* as in a statement such as LET A="CHRIS".

What we need is a whole new kind of variable which can have a character string as its value, so we can assign values like CHRIS and COMMODORE instead of 963 or 46.735.

This new kind of variable is called a *string variable.* A string variable can be assigned by the LET, READ, and INPUT statements. String variables can be printed just like character strings, and can be used almost anywhere we have already used a character string between quotes. There are also some functions for use just with strings.

There are many differences between numeric variables and string variables, starting with their names. To distinguish a string variable from a numeric variable, the names of string variables end with a $. Just as with a numeric variable, the variable name can be any length, but only the first two characters are significant. This means that variable names like CHRIS$ and CHARLIE$ both refer to the same string variable, CH$. But numeric and string variables are completely separate, so it is possible to use the variables A and A$ at the same time, without conflict. The example shows string variables in use.

```
10 C$="COMMODORE 64                                :rem 54
20 PRINT C$                                        :rem 86
30 Q$="A" : R$="FRIENDLY" : S$="COMPUTER          :rem 167
40 PRINT Q$;R$,S$                                 :rem 186
50 C=999                                          :rem 144
60 PRINT C;C$                                     :rem 216
70 M$="PUNCTUATION!!! )('&%$#                     :rem 254
80 PRINT X$;M$                                     :rem 29
```

Remember that with character strings, the closing quote mark is optional if it is the last thing on the line. This is demonstrated in the assignment of C$ in line 10.

Numeric variables have a default value of 0 until they are assigned a new value. With string variables, the default value is not a character; it is the absence of any characters. This is called a *null string*, and is written "". A null string contains no characters between the quote marks. If you try to print a null string, nothing will be printed, which is why nothing was printed for the unassigned X$ in line 80 of the example. All string variables in a program have a null value until they are assigned a new value. Also, any string variable can be assigned a null value, as in LET A$="". The CLR statement assigns the null string to all string variables when it sets all numeric variables to 0.

A string variable can be assigned character strings that look like numbers. For example, you can LET C$="64". When you PRINT C$, it will appear as if a number had been printed. But remember, this is a string variable. Its value is of the type *character string*. The value of a numeric variable is of the type *number*. Two very different types are involved here, and although they can sometimes be used together, as in PRINT, there are restrictions. The assignment C="64" or C=C$ causes the TYPE MISMATCH error to be printed, because the values to the left and right of the equal sign must be of the same type.

Now that string variables have been introduced, we can be more specific about the definition of a variable. In general, variables have three characteristics: a name, a value, and a type.

The GET statement is much more useful now that we have string variables. If GET is used to input a string variable, any key can be pressed without causing a SYNTAX ERROR. If no key has been pressed, the null string is assigned to the indicated variable. GET with strings is often used to get the answer to a simple question.

```
10 PRINT "DO YOU WANT TO CHANGE THE BORDER COLOR?"
                                              :rem 162
20 GET A$ : IF A$="" GOTO 20                  :rem 241
30 IF A$="N" THEN PRINT "NO" :END             :rem 212
40 IF A$<>"Y" GOTO 20                          :rem 10
50 PRINT "YES"                                 :rem 39
```

```
60 INPUT "WHAT IS THE COLOR NUMBER";C      :rem 177
70 POKE 53280,C                              :rem 7
```

The program stays in an infinite loop at line 20 if no key is pressed. In lines 30 and 40, the string variable A$ is compared to character strings using the relational operators = and <>. The program is designed to print full responses of YES or NO only when the keys Y or N are pressed. Nothing will be printed if a different key is pressed.

The GET statement will assign only one character to a string variable. To input strings longer than one character, use the INPUT statement.

```
10 PRINT "PLEASE ENTER YOUR NAME"          :rem 218
20 INPUT N$                                :rem 100
30 PRINT N$;" ";                            :rem 28
40 GOTO 30                                   :rem 0
```

String variables and the INPUT statement form a very powerful combination in programming. Now it is possible to use the full range of the keyboard for getting information from a user.

Just as with numeric variables, if the user presses RETURN in response to the ? prompt without typing anything else, the indicated string variable is not assigned a new value. Pressing RETURN for the ?? prompt causes the string variable to be assigned the null string.

You do not have to type quote marks when responding to the input of a string variable. Quote marks are needed only when the string being entered contains a comma or colon, and since INPUT ignores leading spaces, quote marks are also needed to input a string of spaces.

String variables can be used with READ and DATA. Just put character strings in the DATA statements, separated by commas, and include string variables in the variable list for READ.

```
10 READ C$                                 :rem 228
20 PRINT C$                                 :rem 86
30 IF C$<>"BLACK" GOTO 10                   :rem 14
40 DATA RED,YELLOW,BLUE,GREEN,BLACK        :rem 219
```

READ can only work with constant numbers and character strings; you cannot put variable names in a DATA statement. The quote marks are normally used to distinguish between a variable name and a character string. Since variable names should never appear in a DATA statement, the quote marks around character strings in DATA statements are optional. The only thing to watch out for when not using the quote marks is that spaces between the character string and the separating comma are included as part of the character string assigned to the variable being read. Leading spaces, however, are ignored. Quote marks also have to be used when a character string to be read contains a comma or colon.

```
40 DATA RED,YELLOW,"BLUE,GREEN",BLACK        :rem 31
```

If the computer executes a READ statement to assign a string variable, and the DATA consists of two commas used together, the variable will be assigned the null string.

Character strings are often mixed with numbers in DATA statements. The example program has been rewritten to show one application.

```
10 READ C$,C                                  :rem 83
20 PRINT C$ : POKE 53280,C : A=TI            :rem 133
30 IF TI-A < 60 GOTO 30                       :rem 59
40 IF C$<>"BLACK" GOTO 10                     :rem 15
50 DATA RED,2,YELLOW,7,BLUE,6,GREEN,5,BLACK,0
                                             :rem 188
```

If you try to read a string into a numeric variable, a SYNTAX ERROR will appear with the error line of the DATA statement, not the READ statement.

String variables can be used almost everywhere a character string can be used. You can use strings for filenames, as in F$="DEMO" : LOAD F$. The only exception to this is that a string variable cannot replace the prompt string in an INPUT statement. The following example will not work as intended.

```
10 P$="WHAT IS YOUR FAVORITE COLOR"           :rem 84
20 INPUT P$;C$                                 :rem 8
```

The computer thinks that the variable P$ is to be assigned. Then, because only a comma can be used to separate variable names in a variable list, the semicolon will cause a SYNTAX error. Two correct methods are as follows.

```
10 P$="WHAT IS YOUR FAVORITE COLOR"           :rem 84
20 PRINT P$; : INPUT C$                      :rem 207
```

```
10 INPUT "WHAT IS YOUR FAVORITE COLOR"; C$:rem 213
```

Finally, there is one reserved string variable. The variable TIME$, or just plain TI$, is always six characters long. The first two characters are the hour, in 24-hour time (also called military notation), the next two are the minute, and the last two tell the second.

```
10 PRINT TI$ : GOTO 10                          :rem 131
```

The only way to set the time is to assign a character string of six characters, all of them digits, to the variable TI$. The statement below sets the time to 3 hours, 58 minutes, and 9 seconds.

TI$="035809

Summary

- Commodore 64 BASIC supports a second kind of variable besides the numeric variable used exclusively in previous chapters. The new kind of variable is called a string variable.
- The two kinds of variables are used to contain different types of values. A numeric variable has a value of the type *number*. A string variable has a value of the type *character string*.
- To distinguish between the two types of variables, string variable names end with a $ symbol. Otherwise, all the other rules about variable names apply, including the rule about two significant characters.
- Variable names used for numeric variables have no effect on names used for string variables. The numeric variable C and the string variable C$ are entirely different things.
- A *null string* is a string which consists of no characters.
- The default value for all string variables is the null string.
- The CLR statement sets all string variables to the null string.
- Both types of variables can be used in the same variable list for PRINT, GET, INPUT, and READ. However, the TYPE MISMATCH error will be printed if an attempt is made to incorrectly mix the two types. In any variable assignment, the value being assigned must be of the same type as the variable name to which it is assigned.
- In general, the three distinguishing characteristics of a variable are its name, value, and type.
- The availability of string variables makes the GET statement much more useful. GET will assign the indicated string variable the null value if no key is pressed. Otherwise, a single character will be assigned, according to the key pressed.

- The INPUT statement is used to input strings longer than one character.
- Pressing RETURN only in response to an input prompt of ? does not change the value of the variable being assigned. But entering nothing for ?? causes all unassigned string variables in the variable list to be assigned the null string.
- String variables can also be included in the variable list for READ.
- The quote marks that delimit a character string are optional when used in DATA statements. The only time that they have to be used is when the string includes a comma or colon as one of its characters.
- The occurrence of two consecutive commas in a DATA statement will be read as a null string.
- String variables can be used in any place where character strings are used, except for the prompt string of an INPUT statement.
- The only reserved string variable is TIME$. The only way to change the time is to assign a string of digits to TI$.

Concatenation and Comparisons

Because the value of a string variable is a character string rather than a number, you would not expect some operators to work with strings. For instance, how could you divide one character string by another? But we have seen that some operators can be used, such as = and <>. This section shows which operators can be used with string variables, and what they do.

The five arithmetic operators—addition, subtraction, multiplication, division, and exponentiation—obviously work only with numbers, as implied by the name *arithmetic.* There is a string operation, however, which behaves like addition and is represented by the plus sign. This operation, called *concatenation,* is used to combine character strings.

```
10 A$="HAPPY" : B$="BIRTHDAY"                  :rem 65
20 C$=A$+B$                                   :rem 252
30 PRINT C$                                    :rem 87
```

The variable C$ is 13 characters long, and the whole character string HAPPYBIRTHDAY is printed to the screen. The first five characters are the string HAPPY, and the remaining eight are BIRTHDAY. When A$ and B$ were concatenated, the characters in B$ were added right on the end of A$. No spaces were added where the strings were joined. Adding one string to the end of another is called *appending*. Concatenation is used to append one string to another.

Note that A$ and B$ are not affected by the operation and still have the same values.

Concatenation always combines strings in left-to-right order. Make the following change to the example and run it again:

```
20 C$=B$+A$                                    :rem 252
```

This time the string BIRTHDAY is in front of HAPPY, because of the order of the variables in line 20.

The example could also have been written without using a third string variable. Change these two lines and RUN the program one more time:

```
20 A$=B$+A$                                    :rem 250
30 PRINT A$                                     :rem 85
```

You can use concatenation several times, if necessary.

```
10 M1$="A " : M2$="FRIENDLY " : M3$="COMPUTER"
                                                :rem 78
20 PRINT M1$+M2$+M3$                            :rem 46
```

Remember, concatenation is a different operation from addition, even though the two operations use the same sign. Values to be concatenated are placed together end to end, not added to produce a sum. When the computer sees the plus sign, it examines the values to the left and right to determine whether addition or concatenation is called for. The values on each side of + must always be of the same type, or the TYPE MISMATCH error will occur.

There is a limit to how long a string can be. A string variable cannot contain more than 255 characters, or the STRING TOO LONG error will be printed. To cause this error, try the next program.

```
10 Q$="!"                                      :rem 120
20 PRINT Q$                                    :rem 100
30 Q$=Q$+Q$                                     :rem 42
40 GOTO 20                                     :rem 255
```

In most cases, you will not need to use strings that even begin to approach a length of 255 characters, so this limit should be of little concern.

We have already seen that relational operators can be used with strings, in checking for equality or inequality. The question is, how does the concept of *less than* or *greater than* apply to strings? Maybe the length has something to do with it.

```
10 X$="AARDVARK" : Y$="ZOO"                    :rem 218
20 IF X$<Y$ THEN PRINT "LESS THAN"             :rem 136
30 IF X$=Y$ THEN PRINT "EQUAL"                 :rem 160
40 IF X$>Y$ THEN PRINT "GREATER THAN"           :rem 95
```

When you run this program, you may be surprised to find that X$ is less than Y$, even though X$ has more characters. The variable X$ is less than Y$ because the letter A comes before the letter Z in the alphabet. Relational operators can be used to alphabetize character strings, following the same rules used in dictionaries. Checking always starts with the first character of each string. If the first characters are equal, checking continues with the second character of each string. This may continue for as many characters as needed, until two are found which are not equal. Thus, the string ALICE would be greater than the string ALEXANDER because I is greater than E.

String length becomes a factor only when one string starts with the same characters as another string. When comparing AL and ALEX, it is impossible to compare the third characters because AL has no third character. In such a case, the longer string is greater. And the null string is always less than a string of any length.

The combinations of relational operators can also be used, so <= and >= are valid. All six relational operators can be used with strings, just so long as the types are not mixed.

The next example shows how relational operators can be used for range checking on characters, and how concatenation can build a long string variable. The program asks you to type anything on the keyboard, but it will let you enter only letters of the alphabet or the space.

```
10 PRINT "WHAT IS YOUR NAME?"                  :rem 177
20 GET A$:IF A$="" GOTO 20                     :rem 241
30 IF A$>="A" AND A$<="Z" OR A$=" " THEN N$=N$+A$
   {SPACE}: PRINT N$                            :rem 70
40 GOTO 20                                     :rem 255
```

When AND, OR, and NOT are used as logical operators, they operate on values of *true* or *false* instead of numbers. Likewise, they cannot operate on character strings, although they can be used in expressions which contain character strings, like line 30 of the example. Using AND and OR to clear and set bits applies only to numbers, not to character strings.

Summary

- The arithmetic operators cannot be used with character strings, but the plus sign, which indicates *addition* when used with numbers, means *concatenate* when used with strings.
- When two strings are concatenated, the second string is appended to the end of the first one, in left-to-right order, forming a longer string.
- The maximum length for a string is 255 characters. Attempting to make a character string longer than this limit causes the STRING TOO LONG error to be printed.
- All of the relational operators can be used with strings, including combinations such as <=.
- The conditions *less than* and *greater than* depend on the alphabetical ordering of the characters in the strings. A comparison is made on the first character of each string. If the first characters are equal, checking will continue to subsequent characters until two are found which are not equal.
- If one string runs out of characters before unequal characters are found, the longer string is greater.
- The null string is always less than any other string.
- None of the logical operators can be used directly with character strings.
- With any operator, the two operands must be of the same type, or the TYPE MISMATCH error will occur.

String Functions

String variables open up many new possibilities in programming, but they also present some problems. For instance, it would be handy to be able to count the number of characters in a string. To handle situations like this, there are several functions that are specifically for use with strings. Like the arithmetic functions introduced earlier, these functions have a three-letter name and require one argument, but sometimes the argument may be a string instead of a number. And sometimes the value returned by these functions is a string. This section examines three of these functions.

The first string function is called LEN. The argument must be a string, and the value returned is always a number. The LEN function returns the length of the string used as the argument.

```
10 PRINT "PLEASE CHOOSE A NEW PASSWORD"     :rem 75
20 INPUT P$ : IF LEN(P$)<4 THEN PRINT "PASSWORD IS
    TOO SHORT" : GOTO 20                    :rem 169
30 PRINT "GOOD -THE PASSWORD IS" LEN(P$) "LETTERS
   {SPACE}LONG"                              :rem 181
```

The length of a string is simply the number of characters in the string. The null string contains no characters, so the LEN of a null string would be 0. The maximum string length is 255, so LEN can never return a value greater than 255. Using a number instead of a string for the argument causes the TYPE MISMATCH error.

This function is often used together with other string functions. Examples of using LEN to control loops are given in the later sections of this chapter.

As you know, computers work with numbers. Everything, including characters, must be translated into a number before it can be processed by the computer. For that reason, every character has an ASCII code. ASCII stands for American Standard Code for Information Interchange. It is an agreement among the manufacturers of computer equipment concerning which numbers correspond to which characters. For example, the ASCII code for the letter A is 65. Whenever the computer has to do something with the letter A, it is internally dealing with the number 65.

ASCII provides some compatibility between computers, making possible things like telecommunications between different kinds of computers. Even if you are not working with telecommunications, ASCII can make it easier to work with characters.

The ASC function returns the ASCII value of the first character in a string. That means that like LEN, the argument has to be a string, and the value returned is always a number.

PRINT ASC("ALPHABET")

The above example printed the number 65. The ASC function looks at only the first character of the string. If the string contains other characters, they are ignored. The range of values that can be returned by ASC is from 0 to 255.

One precaution is necessary when using this function. The null string contains no characters, so there is no way to determine an ASCII value. Using a null string as the argument for the ASC function produces the ILLEGAL QUANTITY error.

```
10 GET A$ : IF A$="" GOTO 10                        :rem 239
20 PRINT ASC(A$) : GOTO 10                           :rem 80
```

Try typing the various keys on the keyboard. You will see that each key which prints a character has an ASCII value. Next, try pressing the special function keys. These keys have ASCII values, too. Now you know how a program can determine if the special function keys have been pressed. This also applies to other special keys like the screen clear and cursor movement keys.

The last function is a little different from the first two. Unlike LEN and ASC, which require strings as arguments, CHR$ needs a number, specifically an ASCII number. Also, CHR$ returns a string, not a number. This is indicated by the $ at the end of the function name. The CHR$ function takes an ASCII value and returns the corresponding character. Thus, CHR$(65) can be used just like the character string A. It could even be concatenated to another string. One use for the CHR$ function is to generate the double quote character, which is otherwise impossible to put in a character string. The ASCII code for the double quote is 34.

PRINT "CHRIS SAID, ";CHR$(34);"WOW!";CHR$(34)

The CHR$ function can be used to print any character.

```
10 K=32                                              :rem 78
20 PRINT CHR$(K);" ";                                :rem 70
30 K=K+1 : IF K<147 GOTO 20                          :rem 25
```

CHR$ is often an easier way to print special characters, which are difficult to put between quote marks. Notice that the READY prompt and cursor are in black. This was done by character 144, one of the color-changing control codes. The program stopped just short of character 147, which is the code to clear the screen. By using all of the color-changing character codes, a colorful demonstration program can be written.

```
10 REM RAINBOW                                     :rem 87
20 PRINT CHR$(147) : REM CLEAR SCREEN              :rem 34
30 READ C : IF C=-999 THEN END                    :rem 233
40 PRINT CHR$(C),"{4 SPACES}COMMODORE 64"           :rem 5
50 GOTO 30                                          :rem 1
60 DATA 5,28,30,31,129,144,149,150,151             :rem 70
70 DATA 152,153,154,155,156,158,159,-999          :rem 204
```

One line appears to be missing because it is in the same color as the background.

Summary

- Several functions are available for use with character strings. The three functions introduced in this section are LEN, ASC, and CHR$.
- The LEN and ASC functions require a string for an argument and return a number. This is reversed for CHR$, which needs a number for an argument and returns a string. The presence of the $ at the end of the function name indicates that the function returns a character string.
- The LEN function is used to count the number of characters in a string. The range of this function is 0 to 255, with 0 being the length of the null string.
- The ASC function returns the ASCII code for the first character in a string. The string must not be the null string, or the ILLEGAL QUANTITY error will be printed. ASC is often used to check input characters. The special function keys have ASCII codes which can be detected by this function.
- The CHR$ function takes an ASCII code and returns a string one character long, the character of the given ASCII code. A common use for this function is to print the double quote character. It also simplifies the printing of special control codes, such as those for screen clearing and color changing.

Type Conversion Functions

Thus far, every attempt to mix numbers and character strings in the same operation has been greeted by the TYPE MISMATCH error. Nevertheless, there will be occasions when a string consisting of digit characters must be used like a number, or when a number must be treated as if it were a string. To do this, you must convert the value in question from one type to the other. There are two directions for conversion, so there are two functions that perform the conversions. The VAL function is used to convert a character string to the type *number,* and the function STR$ is used to convert a number to the type *character string.*

The VAL function returns the numeric value of a character string. Given a string, VAL first skips past any leading spaces. Next, VAL looks at the remaining characters in the string, stopping only when it comes to the end of the string, or when a nonnumeric character is found. The numeric value of the string number is then returned by VAL.

```
10 N$="{3 SPACES}37 DEGREES CELSIUS"      :rem 213
20 PRINT "THE TEMPERATURE IS"; 100-VAL(N$);
                                          :rem 216
30 PRINT "DEGREES BELOW THE BOILING POINT" :rem 27
```

The computer printed a 63 on the screen. The characters which have to do with numbers are the plus and minus signs, the period (decimal point), and of course the ten digits.

VAL returns 0 for the null string or strings which do not start with any characters used with numbers.

PRINT VAL("CHRIS")

The STR$ function takes any number and returns the character string representation of that number. Unlike CHR$, which returns only a single character, STR$ may return several characters.

```
10 INPUT "WHAT IS YOUR FAVORITE NUMBER";N   :rem 6
20 PRINT "THE NUMBER" N "HAS" LEN(STR$(N))-1 "DIGI
   T(S)"                                    :rem 238
```

The character string may contain any of the characters related to numbers, which were listed above. The characters in a string produced by STR$ are always identical to the characters that would appear on the screen if the number were printed. The first character is reserved for the sign of the number, which is why the number 1 was subtracted from the length in the example. The sign character will be a minus sign if the number is negative; otherwise, it will be a space.

Summary

- The VAL and STR$ functions are used to convert a value from one type to the other, to avoid the TYPE MISMATCH error.
- The VAL function takes a string, interprets the characters which are related to numbers, and returns a value.
- VAL ignores any leading spaces.
- The interpretation uses all characters in the string, starting at the beginning, until the end of the string is reached, or a character is found which is not related to numbers.
- The only characters which are related to numbers are the ten digits, the plus and minus signs, and the period.
- The null string and strings which start with a nonnumeric character cause VAL to return a value of 0.
- The STR$ function takes a number and produces its character representation. This representation includes all the characters that would be used if the number were printed. If the number is positive, the string will start with a space.

Substring Functions

Concatenation can be used to put strings together, but there is no operation to take strings apart. Sometimes, however, it is handy to look at selected characters in a string—just a part of the string, not the whole string. So Commodore 64 BASIC has three functions designed to work with parts of character strings. A sequence of characters contained in a string is called a *substring,* which explains the name *substring functions.*

The substring functions, LEFT$, RIGHT$, and MID$, are inconsistent with the other functions we have used so far. First, not all of the names of these functions are exactly three characters long. Also, these functions have more than one argument. Further, the arguments used in these functions are not of the same type. Obviously, the substring functions are going to require careful examination.

The LEFT$ function returns the leftmost characters of a string. To do this it needs two arguments; the first is the string, and the second is the number which tells how many characters, starting with the first character in the string, are to be returned. Run this program:

```
10 K=K+1 : PRINT LEFT$("COMMODORE 64",K) : IF K<12
    GOTO 10                                    :rem 14
```

The first string printed is C, the next is CO, and so on, ending with the full COMMODORE 64. The RIGHT$ works in a similar way, but using the rightmost characters. Add this line to the program:

```
20 K=K-1 : PRINT RIGHT$("COMMODORE 64",K) : IF K>1
   GOTO 20                                   :rem 53
```

A 0 for the second argument causes LEFT$ and RIGHT$ to return the null string. You can also use a number greater than the length of the first argument, but no additional characters beyond the length of the string will be printed. The only way that the length number can cause an error is if it is greater than 255, an ILLEGAL QUANTITY.

Using LEFT$ and RIGHT$, we can now split one string into two parts.

```
10 C$="HAPPYBIRTHDAY"                        :rem 34
20 A$=LEFT$(C$,5)                           :rem 108
30 B$=RIGHT$(C$,8)                          :rem 196
40 PRINT A$,B$                              :rem 232
```

These functions let a program examine one or more characters in a string, starting at either end. But they still do not let you look at any individual character in the middle of the string. One clever way of accomplishing this objective is to use a combination of LEFT$ and RIGHT$. To print only the sixth character in C$, try this line:

```
50 PRINT LEFT$(RIGHT$(C$,8),1)              :rem 173
```

The letter B was printed. This method works, but it is cumbersome. It would be nice if there was an easier way to work with a character in the middle of a string. That's where the MID$ function comes in. The first argument identifies a string. The second indicates which character of the string is at the beginning of the substring. And the third—yes, third—argument tells how many characters are in the substring.

PRINT MID$("COMMODORE 64",3,2)

The substring MM was printed, because the two characters starting at three characters into the string are MM. Actually, the third argument of MID$ is optional, and if it is omitted, the substring will continue to the end of the string.

PRINT MID$("COMMODORE 64",3)

This time, the substring is MMODORE 64. The MID$ function is so versatile it can be used to do everything done by LEFT$ and RIGHT$, but the use of the arguments is slightly different, so LEFT$ and RIGHT$ are a little more convenient in some cases.

Here is a demonstration which will print the ASCII code for every character in a string you enter.

```
10 INPUT "WHAT IS THE STRING"; S$           :rem 111
20 IF S$="" GOTO 10                         :rem 131
30 K=K+1 : PRINT MID$(S$,K,1);ASC(MID$(S$,K))
                                            :rem 147
40 IF K<LEN(S$) GOTO 30                     :rem 189
```

Because the ASC function uses only the first character in a string or substring, the third argument was not needed in the second use of MID$.

Remember that the MID$ function can only return a character string; it cannot be used to directly change a character string. Line 20 of the next example is illegal and will generate a SYNTAX ERROR.

```
10 C$="HAPPYBIRTHDAY"                        :rem 34
20 MID$(C$,6,1)="C" : REM WRONG!            :rem 103
30 PRINT C$                                  :rem 87
```

The intention was to change the letter B in C$ to the letter C. The only way to change a character in a string is to take the string apart and put it back together again.

```
20 C$=LEFT$(C$,5)+"C"+RIGHT$(C$,7)            :rem 8
```

The substring functions come in handy surprisingly often. By letting you work with any part of a string, they make string handling more flexible and let you do things that otherwise couldn't be done. Just be careful not to make string expressions too complicated. If you nest too many string functions, you will get the FORMULA TOO COMPLEX error.

Summary

- A part of a string, consisting of contiguous characters in the string, is called a *substring*.
- The substring functions differ from other functions in that their names are not always three characters long, and they take multiple arguments of different types.
- The LEFT$ and RIGHT$ functions need a string for the first

argument, and a number for the second argument. The substring returned is the leftmost or rightmost characters in the string. The length of the substring is determined by the second argument.

- LEFT$ and RIGHT$ can be used to split a character string into two parts, thereby reversing the process of concatenation.
- Changing a character in a string requires using LEFT$ and RIGHT$ with concatenation.
- The MID$ function needs at least two arguments. The second argument tells how far into the string the substring starts. The optional third argument specifies the length of the substring. If the third argument is not provided, the substring continues to the end of the string.
- These functions may return the null string as the substring. They will not return a string longer than the original string.
- Nesting of too many string functions results in the FORMULA TOO COMPLEX error.

Chapter 12

Subroutines

Subroutines

The Statements GOSUB and RETURN

As you begin to write longer programs, you will often come across instances of duplicated lines, where the same sequence of statements occurs at more than one place in the program.

```
10 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:CT
   =54276:PW=54275                              :rem 200
20 POKE MV,15:POKE AD,85:POKE SR,168:POKE FL,0:POK
   E PW,8                                       :rem 241
30 POKE FH,RND(0)*80+2 : POKE PW,RND(0)*8+1
                                                :rem 255
40 POKE CT,65 : K=50                            :rem 232
50 K=K-1 : IF K>0 GOTO 50 : REM DELAY            :rem 67
60 POKE CT,64 : K=50                            :rem 233
70 K=K-1 : IF K>0 GOTO 70                       :rem 186
80 GOTO 30                                        :rem 4
```

Notice that lines 50 and 70 are almost identical, and that both start with the same value for K. Since this example is a short program, typing the same statements twice does not seem redundant. However, in another program, the duplicated section could be longer, or it might be duplicated more than once. In such a case, it seems wasteful to repeat the same lines several times. For situations just like this, use a *subroutine.* A subroutine is a sequence of statements that occurs once in a program, but which can be executed at different places. Here is a listing of the example program, revised to use a subroutine:

```
10 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:CT
   =54276:PW=54275                              :rem 200
20 POKE MV,15:POKE AD,85:POKE SR,168:POKE FL,0:POK
   E PW,8                                       :rem 241
30 POKE FH,RND(0)*80+2 : POKE PW,RND(0)*8+1
                                                :rem 255
40 POKE CT,65 : GOSUB 70                        :rem 226
50 POKE CT,64 : GOSUB 70                        :rem 226
60 GOTO 30                                        :rem 2
70 K=50                                          :rem 84
80 K=K-1:IF K>0 GOTO 80                         :rem 188
90 RETURN                                        :rem 73
```

The subroutine consists of lines 70, 80, and 90. The main program is from lines 10 to 60. When the computer executes the GOSUB statement on line 40, execution jumps to line 70. Line 70 is called the *entry point* of the subroutine. After the computer is done with the delay loop, the RETURN statement sends execution back to the main program. Execution continues with the first statement after the GOSUB, which in this case is POKE CT,64. Lines 70–90 get executed a second time when the GOSUB on line 50 is executed.

The GOSUB statement means *go to subroutine*. The act of temporarily diverting execution to a subroutine is referred to as *calling* the subroutine. A subroutine can be called any number of times in the course of a program. So that the computer knows where the subroutine begins, the keyword GOSUB must be followed by a line number. The line number must be a constant; no expressions or variables are allowed. Also, if a program tries to GOSUB to a line which does not exist, the UNDEF'D STATEMENT error message will be printed.

A subroutine is like a miniature program, except that its last statement is RETURN, not END. When executed, the RETURN statement causes execution to pick up where it left off in the main program. Execution jumps to the first statement after the most recent GOSUB. RETURN is like a GOTO statement, but there is no line number. The place to jump back to has already been determined by the previous GOSUB. Another difference is that RETURN can jump into the middle of a line, whereas GOTO must always jump to the beginning of a line. This means that you can put statements on the same line after a GOSUB. The statements in lines 40–60 could have been written on one line.

```
40 POKE CT,65 : GOSUB 70 : POKE CT,64 : GOSUB 70 :
   GOTO 30                                  :rem 111
50                                          :rem 101
60                                          :rem 102
```

Even though a subroutine may be called several times in a program, it does not have to do the same thing every time it is called. A common technique is to have the subroutine use a variable. The variable is assigned just before the subroutine is called.

```
40 POKE CT,65 : K=100 : GOSUB 80           :rem 54
50 POKE CT,64 : K=50 : GOSUB 80            :rem 10
60 GOTO 30                                  :rem 2
```

One other technique is to have multiple entry points for a subroutine. Let's say that the example program calls the subroutine often, and that it usually needs K to start at 50. The assignment of K is in the first line of the subroutine, and the subroutine is entered at that point. But for the few occasions when the subroutine must be executed with a different value for K, the entry point is one line later.

```
50 POKE CT,64 : GOSUB 70                          :rem 226
```

Here is a more elaborate example. This is the main program.

```
100 PRINT "PLEASE NAME A BORDER COLOR"     :rem 187
110 A=53280 : GOSUB 150                     :rem 98
120 PRINT "PLEASE NAME A BACKGROUND COLOR":rem 223
130 A=53281 : GOSUB 150                    :rem 101
```

This is the subroutine.

```
150 INPUT C$ : IF C$="" GOTO 150            :rem 13
160 IF C$="RED" THEN POKE A,2 : GOTO 200  :rem 235
170 IF C$="YELLOW" THEN POKE A,7 : GOTO 200
                                           :rem 242
180 IF C$="GREEN" THEN POKE A,5 : GOTO 200:rem 134
190 PRINT "I DO NOT KNOW THE COLOR " C$:GOTO 150
                                            :rem 71
200 RETURN                                 :rem 114
```

When you run this program, the computer will ask you to name a color. Red, yellow, and green are the only colors that will be recognized. When you enter the color name, the border will change to that color. The procedure will be repeated for the background. Then, another prompt will appear. This time, when you enter a color name, you will get a RETURN WITHOUT GOSUB error. A quick look at the program shows why. The execution of the main program has accidentally fallen into the subroutine. The problem can be remedied by placing an END statement at line 140.

```
140 END                                    :rem 108
```

This is an example of when the END statement is necessary, but is not the last statement in a program. Like DATA statements, subroutines are usually put at the end of a program, out of the way of the main program. Forgetting to use END to prevent main program execution from running into a subroutine is a common programming mistake.

The RETURN statement depends on the fact that a GOSUB has been executed before it. The RETURN WITHOUT GOSUB error indicates that a RETURN has been encountered without a matching GOSUB. It is possible, though, to have more RETURN statements in a program than GOSUB statements. Just make sure that for every GOSUB executed, exactly one RETURN is executed. The example program can be simplified a little bit by replacing every GOTO 200 with a RETURN. Line 200 is no longer needed.

```
160 IF C$="RED" THEN POKE A,2 : RETURN        :rem 0
170 IF C$="YELLOW" THEN POKE A,7 : RETURN     :rem 7
180 IF C$="GREEN" THEN POKE A,5 : RETURN    :rem 155
200                                         :rem 146
```

Subroutines are a second form of repetition, with loops being the first. Like a loop, a subroutine is a sequence of statements which occurs once in a program, but will have been executed many times by the time the program is done. The difference is that every time a subroutine is entered, the sequence of statements is executed only once. The placement of GOSUB statements within a program is what controls how many times the statements in the sequence are repeated.

Summary

- A subroutine is a sequence of statements that occurs once in a program, but is executed from several places in the program.
- The purpose of using a subroutine is to avoid duplicating statement sequences in a program.
- The statements used to implement subroutines are GOSUB (go to subroutine) and RETURN (return to main program).
- The syntax for the GOSUB statement is the keyword GOSUB followed by a line number. The RETURN statement consists only of the keyword RETURN.
- The GOSUB statement makes execution jump to the beginning of the specified line. If the specified line does not exist, an UNDEF'D STATEMENT error will occur.
- The main difference between GOSUB and GOTO is that with GOSUB, the computer remembers where execution stopped in the main program.
- Because execution eventually resumes where it left off in the main program, statements can be placed after a GOSUB on the same line.
- The RETURN statement causes execution to jump to the first statement after the most recent GOSUB. This may mean

jumping into the middle of a line. RETURN is comparable to the END statement of a main program.

- RETURN should be used only when a GOSUB has been previously executed. If the computer encounters a RETURN without having previously executed a matching GOSUB, the place to jump back to is undefined, so a RETURN WITHOUT GOSUB error is printed.
- A subroutine can contain several RETURN statements, provided that for every GOSUB executed, exactly one RETURN is executed.
- Subroutines are usually put after the main program. An END statement is needed to separate the main program from the subroutine. Having a main program accidentally fall into a subroutine is a sure way of getting the RETURN WITHOUT GOSUB error.
- When execution is transferred to a subroutine, it is said that the subroutine has been *called.*
- To make a subroutine more flexible, it can use a variable that is assigned just before the subroutine is called.
- The *entry point* of a subroutine is the line to which execution jumps when the subroutine is called. Subroutines can have more than one entry point.

Multiple Subroutines

A program can have more than one subroutine.

```
100 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:C
    T=54276                                 :rem 211
110 POKE FL,0 : POKE MV,15                   :rem 77
120 PRINT "FREQUENCY (HIGH)"; : L=255 : GOSUB 210
    {SPACE}: POKE FH,N : N=0                :rem 212
130 PRINT "ATTACK RATE"; : GOSUB 200 : E=N:rem 218
140 PRINT "DECAY RATE"; : GOSUB 200 : POKE AD,16*E
    +N                                      :rem 232
150 PRINT "SUSTAIN LEVEL"; : GOSUB 200 : E=N
                                            :rem 151
160 PRINT "RELEASE RATE"; : GOSUB 200: POKE SR,16*
    E+N                                     :rem 165
170 POKE CT,129 : GOSUB 230 : PRINT "*"     :rem 170
180 POKE CT,128 : GOSUB 230 : PRINT          :rem 60
190 GOTO 120                                :rem 102
200 L=15                                    :rem 129
210 INPUT N : IF N<0 OR N>L OR N<>INT(N) THEN PRIN
    T "OUT OF RANGE" : GOTO 210              :rem 96
220 RETURN                                  :rem 116
```

```
23Ø K=3ØØ                                          :rem 176
24Ø K=K-1 : IF K>Ø GOTO 24Ø                         :rem 24
25Ø RETURN                                         :rem 119
```

When you run this program, you must specify a frequency value from 0 to 255, and envelope values from 0 to 15. The program will start the sound, using the noise waveform. An asterisk is printed when the sound is released. After that, the program repeats for a different sound.

This program uses two subroutines. The one starting at line 230 is a standard delay loop. The other subroutine is used to input and check the range of a number. It has two entry points: line 200 for when the range is from 0 to 15, and line 210 for when the range is from 0 to some other number. Because the two subroutines are called separately and occupy different lines in the program, they coexist without any problems. You should not get the RETURN WITHOUT GOSUB error.

The example shows that a program can have several subroutines, each one independent of the others. How about having two or more subroutines in effect at the same time? Can one subroutine call another subroutine? The answer is yes, provided that for every GOSUB executed, exactly one RETURN is executed. When one subroutine is called from within another, the subroutines are said to be *nested.*

```
1Ø REM DEMO OF NESTED SUBROUTINES
2Ø PRINT "MAY
3Ø GOSUB 5Ø
4Ø PRINT "YOU!":END

                         5Ø PRINT "THE
                         6Ø GOSUB 8Ø
                         7Ø PRINT "WITH":RETURN

8Ø PRINT "FORCE"
9Ø PRINT "BE":RETURN
```

A well-known saying will be printed when you run this program. There are two subroutines in this example. The first subroutine starts at line 50, and it calls the other one which starts at line 80. Trace through the program to see the order in which the statements are executed. First there is the GOSUB in line 30, then the GOSUB of line 60, followed by the RETURN on line 90, and the RETURN at line 70. Notice that at no time have more RETURN statements been executed than GOSUB statements. If the order of execution were GOSUB, RETURN, and RETURN, the second RETURN would generate the RETURN WITHOUT GOSUB error, because no corresponding GOSUB had previously been executed. Also, once the program is finished, there have been just as many RETURN statements executed as GOSUB statements.

Of course, the example is not a practical application of nested subroutines. It is intended only to show you how nesting works.

Nested subroutines can often be found in larger programs. Commodore 64 BASIC has a maximum nesting limit of 23 levels, and in certain circumstances this limit is even less. If a program ever exceeds the nesting capacity of BASIC, the error message OUT OF MEMORY will be printed, even if there are plenty of free bytes still available. With simple programs, this should never happen.

There is one last possibility to be considered. What if a subroutine calls itself? This technique, called *recursion,* is the third form of repetition, after loops and subroutines. Many interesting things can be done using recursion, but it is difficult to program properly, and few of our present programming needs require that we use it. Recursion is an advanced topic that is beyond the scope of this book. For now, just be careful to avoid program lines like the next one, which is not only pointless, but is also a sure way of producing the OUT OF MEMORY error.

```
100 GOSUB 100                                  :rem 162
```

Summary

- Several subroutines can be used in one program.
- When one subroutine calls another subroutine, the subroutines are *nested.*
- In order for nested subroutines to work without creating a RETURN WITHOUT GOSUB error, exactly one RETURN must be executed for every GOSUB executed.
- The Commodore 64 cannot nest deeper than 23 levels. Nesting beyond this limit causes the OUT OF MEMORY error.
- The third form of repetition, recursion, happens when a subroutine calls itself. For our purposes, recursion is best avoided.

Bitmapped Graphics

The only kind of graphics which we have dealt with so far has been character graphics. In character graphics, each character has its own definition, and that one definition may appear at several places on the screen. You could even put the same character at all 1000 screen positions. But character graphics has some limitations. When you want to draw a really big object, you have to redefine a lot of characters. For drawing long lines at different angles, there may not even be enough characters available. Only 256 characters can be defined at any time. With 256 characters you could create a detailed picture. Assuming, however, that each character might need a different definition, you could run out of characters after filling only the top quarter of the screen.

To get around the restrictions of character graphics, your Commodore 64 also supports *bitmapped* graphics. In bitmapped graphics, each character position on the screen is allotted eight bytes for its own definition. Every point that is part of an 8 by 8 character matrix can be individually referenced, and this is true for all 1000 positions on the screen. Thus, instead of having 40 by 25 screen positions, there are now 40*8=320 by 25*8=200 points which can be independently set or cleared. A total of 64,000 points should certainly be sufficient for drawing most pictures.

Since there are 1000 screen positions, and the definition for each position takes 8 bytes, a bitmapped graphics screen requires 8000 bytes. The normal screen consists of only 1000 bytes, which hardly comes close. As is the case with a redefined character set, a separate section of memory will have to be used. In fact, you could think of this 8000-byte area as one gigantic character set. The first eight bytes define the character in the upper-left corner of the screen. The next eight bytes define the character immediately to the right, and so on. Also, as with character sets, it is necessary to follow a certain procedure before using bitmapped graphics. A program is provided to take care of that, but first let's consider the problem of dealing with 64000 points.

With so many individual points, we are going to need an easy way of labeling each one so we can distinguish one from another. We will number the points, using a standard coordinate system in which each point has its own horizontal and vertical coordinates. No two points have the same combination of coordinates. The horizontal (X) coordinate can range from 0 to 319, running from left to right. The vertical (Y) coordinate goes from 0 to 199, from top to bottom. To specify a particular point, the X coordinate is used to denote a column, and the Y coordinate is used to select a row. The intersection of the column and row is the desired point. The point which has coordinates 0 and 0 is in the upper-left corner of the screen. The point with the coordinates 160 and 73 would be 160 columns over and 73 rows down. Note that points with higher values for Y appear lower on the screen. This is backward from normal mathematics, where Y increases upward.

When you set one of the 64,000 bits that make up the bitmapped screen, you are *plotting* a point. You can designate a point using the X and Y coordinates, but to actually plot the point in memory, you must find the correct byte and set the appropriate bit. To simplify this as much as possible, a plotting subroutine is provided as line 900 of the demonstration program. All you have to do to plot a point is put the coordinate values in the variables X and Y and call the subroutine. The point will be automatically plotted, and you don't have to deal with any complex calculations.

One other matter to be discussed is how to set the color for each point. Since the normal 1000 bytes for screen memory are not needed in bitmapped mode, they are used to hold color information. For each character position, the byte at the corresponding location in normal screen memory specifies the image and background color. This means that within each 8 by 8 matrix, all points have to be the same color, but the color can change from character to character. Also, each character position can have a different background color. The color information is calculated by taking the number for the image color, multiplying it by 16, and adding the number for the background color. If location 1024 contains the value 52, which is 3*16+4, the points in that character will be in color 3, and the background will be color 4. Color memory is not used in the standard bitmapped graphics mode.

Here is the demonstration program. It plots the seven points that form the Big Dipper.

Plotdemo

```
100 POKE 53272,PEEK(53272)OR8:BASE=8192     :rem 167
110 POKE 53265,PEEK(53265)OR32              :rem 115
120 K=BASE                                   :rem 54
130 POKE K,0:K=K+1:IF K<16192 GOTO 130      :rem 243
140 K=1024                                  :rem 228
150 POKE K,16:K=K+1:IF K<2024 GOTO 150      :rem 243
160 X=10:Y=10:GOSUB 900                      :rem 17
170 X=50:Y=30:GOSUB 900                      :rem 24
180 X=60:Y=50:GOSUB 900                      :rem 28
190 X=90:Y=80:GOSUB 900                      :rem 35
200 X=80:Y=120:GOSUB 900                     :rem 69
210 X=130:Y=140:GOSUB 900                   :rem 116
220 X=160:Y=110:GOSUB 900                   :rem 117
230 END                                     :rem 108
900 BY=BA+40*(YAND248)+(YAND7)+(XAND504):POKE BY,P
    EEK(BY)OR2↑(NOTXAND7):RETURN             :rem 23
```

The first thing the program does is inform the graphics chip that the 8000-byte section of memory will start at location 8192. This location is called the *base* of the bitmapped screen.

POKE 53272,PEEK(53272) OR 8

The next POKE puts the graphics chip into the bitmapped mode. The following POKE does this.

POKE 53265,PEEK(53265) OR 32

To turn off the bitmapped mode, use this statement:

POKE 53265,PEEK(53265) AND 223:POKE
  53272,PEEK(53272) AND 247

The variable BASE is also assigned the value 8192 for future reference.

The program then enters a loop to clear the 8000 bytes. You can see each byte being cleared. This takes about two minutes.

Next, the program uses a loop to set the color for the image and background of each character position. The points will be white on a black background.

Finally, the program is ready to start plotting points. Seven points are plotted, after which the program ends.

Admittedly, mere plotting of points is not very exciting. It does not give you a good example of what is possible using bitmapped graphics. Here is a modified version of the demonstration program that includes a line drawing subroutine. Set the variables X and Y to the coordinates of the first end point, set X0 and Y0 to the coordinates of the second end point, and call the subroutine with the statement GOSUB 910. The subroutine will plot all the points between the two end points to draw a straight line.

You may notice that we put a REM statement in front of line 130. Clearing the 8000 bytes takes a long time, and for purposes of experimenting we don't have to do it every time.

Drawdemo

```
100 POKE 53272,PEEK(53272)OR8:BASE=8192    :rem 167
110 POKE 53265,PEEK(53265)OR32             :rem 115
120 K=BASE                                  :rem 54
130 REM POKE K,0:K=K+1:IF K<16192 GOTO 130:rem 215
140 K=1024                                 :rem 228
150 POKE K,16:K=K+1:IF K<2024 GOTO 150     :rem 243
160 X=10:Y=10:GOSUB 900                     :rem 17
170 X0=50:Y0=30:GOSUB 910                  :rem 121
180 X0=60:Y0=50:GOSUB 910                  :rem 125
190 X0=90:Y0=80:GOSUB 910                  :rem 132
200 X0=80:Y0=120:GOSUB 910                 :rem 166
210 X0=130:Y0=140:GOSUB 910                :rem 213
220 X0=160:Y0=110:GOSUB 910                :rem 214
230 X0=90:Y0=80:GOSUB 910                  :rem 127
240 END                                    :rem 109
900 BY=BA+40*(YAND248)+(YAND7)+(XAND504):POKE BY,P
    EEK(BY)OR2↑(NOTXAND7):RETURN            :rem 23
910 DX=ABS(X0-X):DY=ABS(Y0-Y):SX=SGN(X0-X):SY=SGN(
    Y0-Y):K=1:IF DX<DY GOTO 960            :rem 205
920 E=DY-DX/2                              :rem 228
930 IF E<0 THEN E=E+DY:GOTO 950            :rem 171
940 Y=Y+SY:E=E+DY-DX                       :rem 245
950 X=X+SX:GOSUB 900:K=K+1:IF K<=DX GOTO 930
                                            :rem 27
955 RETURN                                 :rem 131
960 E=DX-DY/2                              :rem 232
970 IF E<0 THEN E=E+DX:GOTO 990            :rem 178
980 X=X+SX:E=E+DX-DY                       :rem 246
990 Y=Y+SY:GOSUB 900:K=K+1:IF K<=DY GOTO 970
                                            :rem 39
995 RETURN                                 :rem 135
```

Now a few words about the plotting subroutines. The variable BASE must be assigned the value 8192 in order for the subroutine to work. The only time this value may change is when you're using advanced graphics methods that let you place the 8000-byte section in other places. The variables X and Y should never be allowed to contain values outside the ranges 0–319 and 0–199, respectively, or some wrong locations may be inadvertently POKEd, causing the computer to act in unusual ways. And if you want to erase a point, replace line 900 with the following:

UNPLOT

```
900 BY=BA+40*(YAND248)+(YAND7)+(XAND504):POKE BY,P
    EEK(BY)ANDNOT2↑(NOTXAND7)                :rem 32
905 RETURN                                   :rem 126
```

The last feature of bitmapped graphics is that it supports a multicolor mode, similar to the one for character graphics. In this mode, each point can be one of three colors, or contain the background color. The three possible image colors can be different for each character position. With all these color possibilities, the multicolor bitmapped mode is extremely versatile.

To achieve the multicolor effect, bit-pairs must again be used, so the total number of points displayed horizontally is reduced to 160, and each point is double the normal width. The combination of the two bits determines the color used for each point.

Bit Pattern	Color
00	background color 0 (location 53281)
01	screen memory (color number times 16)
10	screen memory
11	color memory

The following demonstration program draws three lines, one in each color. The first line is drawn by plotting every other point, starting at the second point from the left. The next line is drawn in the same way, except that the plotting starts in the first column. The third line is drawn so that every point is plotted. Keep in mind the fact that the set of three colors can be changed at each character position.

McDEMO

```
100 POKE 53272,PEEK(53272)OR8:BASE=8192    :rem 167
110 POKE 53265,PEEK(53265)OR32 : REM ENABLE BIT MA
    PPED MODE                                :rem 243
115 POKE 53270,PEEK(53270)OR16 : REM ENABLE MULTIC
    OLOR MODE                                :rem 102
```

```
120 K=BASE                                         :rem 54
130 POKE K,0:K=K+1:IF K<9152 GOTO 130             :rem 193
140 K=0                                            :rem 77
150 POKE 1024+K,208:POKE 55296+K,1:K=K+1:IF K<200
    {SPACE}GOTO 150                                :rem 41
160 K=0:POKE 53280,0                               :rem 22
170 X=K : Y=10 : GOSUB 900                        :rem 252
180 X=K+1 : Y=12 : GOSUB 900                       :rem 91
190 X=K:Y=14:GOSUB 900:X=K+1:GOSUB 900            :rem 203
200 K=K+2 : IF K<319 GOTO 170                     :rem 128
210 END                                           :rem 106
900 BY=BA+40*(YAND248)+(YAND7)+(XAND504):POKE BY,P
    EEK(BY)OR2↑(NOTXAND7):RETURN                   :rem 23
```

The advantage to bitmapped graphics is that it gives you full control over every point on the screen. This makes large, detailed pictures feasible. With the color available in multicolor mode, your Commodore 64 can display some dazzling pictures on the screen.

Chapter 13

Simplified Loops

Simplified Loops

The FOR and NEXT Statements

There are two kinds of loops, depending on whether it is known how many times the loop will be repeated when it is first entered. Loops can either repeat indefinitely until a certain condition is true (the GET statement gets something other than the null string), or repeat a specified number of times (READ is used to change a color eight times). The IF-THEN statement is great for exiting a loop of the first type, but loops which repeat a certain number of times require a counting variable, starting and ending values, an assignment statement to increment the counting variable, conditional logic to check if the loop is done, and a way to make execution jump back to the top of the loop. These loops are really quite simple; it's just the statements which set them up that get complicated. Since such loops are used very often, two special statements are available, to simplify loops which repeat a specified number of times.

```
10 K=1                                   :rem 26
20 PRINT "COMMODORE 64"                  :rem 66
30 K=K+1 : IF K<5 GOTO 20               :rem 178

10 FOR K=1 TO 5                         :rem 217
20 PRINT "COMMODORE 64"                  :rem 66
30 NEXT K                               :rem 237
```

All we want to do is print the name of our favorite computer five times, and both programs do so. The first program shows how we have learned to set up loops of this type. The second program uses the new statements, FOR and NEXT, to accomplish the same thing. The FOR statement assigns the value 1 to the variable K (comparable to line 10 of the first program). Execution continues to the following statement, which is a PRINT. There could be several statements here, not just one. When the computer reaches the NEXT statement, it increments K by 1 (this is like K=K+1), and checks if the new value for K is less than or equal to the number that was after the keyword TO in the FOR statement (IF K<=5). If so, execution jumps back to the first statement after FOR (the GOTO 20). This condition will be true every time through the loop except the last time, when K equals 5. Adding 1 to K makes it 6, which is outside the range specified by the FOR statement. The loop is done, and execution continues with the first statement after NEXT.

Why is it preferable to use the FOR and NEXT statements rather than the other method? Mainly because FOR and NEXT are easier to use—there are only two statements instead of three, and less typing is needed. Also, it is easier to read a loop formed with FOR and NEXT. With FOR and NEXT, the boundaries of the loop can be quickly spotted. You must carefully read line 30 of the first example to determine that it is the end of a loop. If other statements inside the loop included IF-THEN statements, they could easily be mistaken for the loop's end. More statements after line 30 could make line 30 hard to spot. Another advantage is that the second version could be written on one line, whereas the older version could not. In fact, the older version can't even be done in the immediate mode, but the newer one can.

FOR K=1 TO 5 : PRINT "COMMODORE 64" : NEXT K

The NEXT statement, like RETURN, can jump into the middle of a line.

Just by looking at the FOR statement, you can tell right away how many times the loop will be executed. The number before the keyword TO tells the starting value of the counting variable, and the number after TO tells the ending value. And it's easy to change how many times the loop is executed.

```
10 FOR K=1 TO 15                                :rem 10
```

The PRINT statement will now be executed 15 times. You can also change the starting value of the variable. By starting K at 14, the loop is executed only twice.

```
10 FOR K=14 TO 15                               :rem 62
```

The counting variable does not have to be used just to control how many times the loop is executed; it can also be used inside the loop. The next example puts all of the characters on the screen, in different colors.

```
10 FOR K=0 TO 255                               :rem 63
20 POKE 1024+K,K : REM CHARACTER                :rem 240
30 POKE 55296+K,K : REM COLOR                   :rem 39
40 NEXT K                                       :rem 238
```

The FOR and NEXT statements greatly simplify the handling of loops. By being able to change the starting and ending values, FOR and NEXT can do anything that can be done with the old method, with one exception. How can the loop be made to go backward? Consider the following program.

```
10 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:CT
   =54276                                    :rem 163
20 POKE MV,15:POKE AD,Ø:POKE SR,24Ø:POKE FL,Ø:POKE
    CT,17                                    :rem 2Ø3
3Ø FOR K=Ø TO 255                             :rem 65
4Ø POKE FH,K : PRINT K                       :rem 17Ø
5Ø NEXT K                                    :rem 239
6Ø POKE CT,16                                :rem 191
```

The program produces a tone which starts at a low pitch and increases. Perhaps if the values before and after the keyword TO were switched, the loop would go backward. Try it.

```
3Ø FOR K=255 TO Ø                             :rem 65
```

The loop is executed only once. To find out why, we have to trace the steps followed by the computer. The variable K starts at 255, is used in line 40, and gets incremented to 256 by the NEXT statement. This new value of K is certainly not less than or equal to 0, the number after TO, so the loop is done. The problem is that the NEXT statement still adds 1 to the counting variable, when it needs to subtract 1. Fortunately, there is a way to make NEXT add numbers other than 1 to the counting variable. The FOR statement includes an optional keyword STEP. This keyword goes at the end of the FOR statement, and it must be followed by a value. This is the value that is added to the counting variable every time NEXT is executed. A value of −1 will cause K to be decremented, and the example will now work correctly.

```
3Ø FOR K=255 TO Ø STEP -1                    :rem 219
```

The optional STEP also lets you control the size of the increment or decrement. Think of climbing a tall staircase. To get to the top faster, you might take two steps at a time, skipping every other step. Using a STEP of 2 in the example means that only every other frequency number will be POKEd.

```
3Ø FOR K=Ø TO 255 STEP 2                     :rem 175
```

Notice that the variable does not have to match exactly the value after the keyword TO in order for the loop to terminate. That is because NEXT checks if the value is less than or equal to the limit, so this loop ends after being executed with K equal to 254. The checking is reversed when a negative value is used for STEP, however, and the test is for greater than or equal to the limit.

Regardless of what value is used for STEP, a loop using FOR and NEXT will always be executed at least once, because the range check on the counting variable is not done until the NEXT statement. All of the statements in the loop will have been executed before the first time NEXT is reached. On the other hand, it is possible to create an infinite loop if a STEP value of 0 is used. Of course, just a plain GOTO would accomplish the same thing.

All of the examples shown so far have used the variable K as a counting variable, but any numeric variable can be used. The demonstration could be modified using J to control the loop.

```
30 FOR J=0 TO 255                                  :rem 64
40 POKE FH,J : PRINT J                            :rem 168
50 NEXT J                                         :rem 238
```

FOR-NEXT loops can be used in different places in a program without confusing the computer. Add this line to the program.

```
55 FOR J=255 TO 0 STEP -1 : POKE FH,J : PRINT J :
   {SPACE}NEXT J                                   :rem 34
```

It is even possible for two loops in a program to use different variables.

```
55 FOR K=255 TO 0 STEP -1 : POKE FH,K : PRINT K :
   {SPACE}NEXT K                                   :rem 38
```

The NEXT statement will always jump to the first statement after the most recent FOR. However, when NEXT is executed, if the variable after the keyword does not match the one in the most recent FOR statement, a NEXT WITHOUT FOR error will occur. This means that a NEXT with the proper variable name is missing.

Now the inevitable question: If several FOR-NEXT loops can be used in the same program, can they be nested? Again, the answer is yes, but only if certain restrictions are observed. With subroutines, the GOSUB and RETURN statements had to match, or the RETURN WITHOUT GOSUB error would occur.

The same thing is true with FOR and NEXT, but it is a little more complicated, because the matching must be done in order. The example has been rewritten to show two loops nested correctly.

```
10 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:CT
   =54276                                   :rem 163
20 POKE MV,15:POKE AD,0:POKE SR,240:POKE FL,0:POKE
    CT,17                                   :rem 203
30 FOR K=0 TO 255                            :rem 65
40 POKE FH,K : PRINT K                      :rem 170
45 FOR J=0 TO 255 : POKE FL,J : NEXT J      :rem 122
50 NEXT K                                   :rem 239
60 POKE CT,16                               :rem 191
```

Location FL (frequency low) acts like a fine-tuning control for the frequency. The loop controlled by J repeats 256 times to change FL, but that entire loop is repeated another 256 times, due to the nesting. A slight glitch in the sound is normal as FH is changed. Note that the J loop is entirely inside the K loop. This is the rule about nesting FOR-NEXT loops. The order of variables in FOR statements is K followed by J, so the order for NEXT statements must be reversed—J and then K. If you do not follow this rule, you will get the NEXT WITHOUT FOR error. Here is a program which uses three correctly nested loops.

```
10 FOR K=1 TO 10                              :rem 5
20 FOR J=1 TO 10                              :rem 5
30 FOR I=1 TO 10                              :rem 5
40 PRINT K,J,I                               :rem 39
50 NEXT I                                   :rem 237
60 NEXT J                                   :rem 239
70 NEXT K                                   :rem 241
```

Again, the order of NEXT statements is precisely opposite the order of the FOR statements. The next program breaks the rule, and will generate ?NEXT WITHOUT FOR ERROR IN 70.

```
10 FOR K=1 TO 10                              :rem 5
20 FOR J=1 TO 10                              :rem 5
30 FOR I=1 TO 10                              :rem 5
40 PRINT K,J,I                               :rem 39
50 NEXT I                                   :rem 237
60 NEXT K                                   :rem 240
70 NEXT J                                   :rem 240
```

It is suggested that you avoid using the variable I as a counter variable, because it looks too much like the number one.

An easy way to tell if loops are nested correctly is to draw lines connecting each pair of FOR and NEXT statements. In the correct example above, you could draw a line from the end of line 10 to the end of line 70, and likewise from 20 to 60, and 30 to 50. The lines will not cross in a correctly written program. If the lines cross, you don't even have to run the program to know that you will get the error message.

There is one other caution about nesting. A counting variable used for one loop cannot be used for any other loop that is in effect at the same time. The following program will generate an error, even though the variable names are in the correct order. The problem is that the variable K appears twice.

```
10 FOR K=1 TO 10                                      :rem 5
20 FOR J=1 TO 10                                      :rem 5
30 FOR K= 1 TO 10                                     :rem 7
40 PRINT K,J,K                                       :rem 41
50 NEXT K                                           :rem 239
60 NEXT J                                           :rem 239
70 NEXT K                                           :rem 241
```

This does not mean that the variable K can't be used for two different loops in the same program. The restriction applies only when the loops are nested. At line 30, K is already in use, so it should not be starting another loop. The error, however, will not occur until the NEXT statement on line 60 is executed.

Simplified Loops

FOR-NEXT loops are used with great frequency in BASIC programs, and as you might expect, there is a shortcut. All of those NEXT statements in the earlier examples may seem to be a little redundant, especially if they were put on the same line.

```
10 FOR K=1 TO 10                                      :rem 5
20 FOR J=1 TO 10                                      :rem 5
30 FOR I=1 TO 10                                      :rem 5
40 PRINT K,J,I                                       :rem 39
50 NEXT I : NEXT J : NEXT K                         :rem 116
```

The shortcut is to replace all of the NEXT statements with just one NEXT, and follow it with a variable list. This is a list of all the counting variables, in the reversed order. You could replace three statements with only one.

```
50 NEXT I,J,K                                :rem 218
```

You can do this only when there are two or more NEXT statements, one right after the other, which can only happen when nested loops are being used. If there are any statements between one NEXT and another, do not use the shortcut.

A second shortcut is to not specify a variable after the keyword NEXT. This is sometimes done when only one loop is being used. The following line is legal and will work:

FOR Q=1 TO 10 : PRINT Q : NEXT

Notice that using this second shortcut makes it impossible to use the first one. Omitting the variable name also makes it harder to find mistakes in a program, because if no variable names are used, no variable name mismatch can be detected. The NEXT WITHOUT FOR error will only occur when more NEXT statements are encountered than FOR statements. It is suggested that you use this second shortcut only when a program contains just one loop.

Sometimes you will see a FOR-NEXT loop with no statements inside. Loops like the one below are another way of creating a delay in a program.

FOR Q=1 TO 300 : NEXT

There is a maximum number of loops which can be nested at the same time. This limit changes according to the number of subroutines in effect and the complexity of calculations. The absolute maximum number of levels that can be nested is nine. Only under unusual circumstances would a program ever have to nest deeper than eight levels. Remember, this limit applies only to nesting; a program could have a hundred loops, but only a few of them might be in effect simultaneously.

The FOR and NEXT statements are designed to handle only loops which repeat a designated number of times, this number being known when the loop is first entered. Loops that repeat until a condition is met should not be written with FOR and NEXT, because FOR-NEXT loops must always be allowed to go to completion. Never jump out of a loop before it has finished. You can use GOSUB, but a GOTO which jumps out of the loop should be avoided. Prematurely jumping out of FOR-NEXT loops is not only bad programming style, but it can also really confuse the computer, without the problems being immediately evident. If it is necessary to jump out of a loop, write the loop using the older method, with the assignment statements and IF-GOTO.

It is also not a good practice to change the value of a counting variable while it is being used in a FOR-NEXT loop. The statements inside the loop can use its value, as was done in many of the example programs, but to assign the counting variable by a LET statement or other means is not recommended.

After a loop has finished, the counting variable retains its last value. Remember that the last value is the one that was out of range and caused the loop to end, so in a loop which went from 1 to 10 stepping by 1, the last value would be 11. Stepping by 1.5, the last value would be 11.5

Summary

- There are two kinds of loops. One kind repeats indefinitely, waiting for a condition to be met. The computer does not know, when it first enters the loop, how many times the loop will be executed. The other type of loop repeats a set number of times. In this case, the computer does know when it enters the loop how many times the loop will be executed.
- The kind of loop which repeats a set number of times is very simple, but to implement it in a program requires assignment statements, conditional logic, and a way to make execution jump to the top of a loop.
- The FOR and NEXT statements are used to simplify the handling of loops which repeat a specified number of times. These two statements do all the work normally done by two LET statements and an IF-GOTO statement.
- The syntax of the FOR statement starts with the keyword FOR. Then comes the name of the counting variable. This must be a numeric variable. String variables cannot be used. After the variable name comes an equal sign, just as in the LET statement, followed by a value which is the starting value for the loop. Next comes the keyword TO and another value, this time the ending value for the loop. All of this may optionally be followed by the keyword STEP and one more value.

- The syntax for the NEXT statement is the keyword NEXT and an optional variable list.
- The FOR statement assigns the counting variable its initial value, and makes the computer remember the end and step values for the loop.
- The NEXT statement makes sure that if it includes a variable name, the name matches the variable in the most recent FOR statement. If there is a mismatch, the NEXT WITHOUT FOR error message is printed. Otherwise, it adds the step value to the counting variable, checks if the counting variable is still in the range specified in the FOR statement, and if so, causes execution to jump to the first statement after the corresponding FOR statement. When that loop is finished, the NEXT statement checks for more variables in its variable list. If there is another one, the whole procedure is repeated for that loop. When there are no more variables, execution proceeds to the statement after NEXT.
- There are many advantages to using FOR and NEXT to create loops:
  1. Only two statements are needed.
  2. The FOR statement indicates right away how many times the loop will repeat.
  3. The boundaries of a loop defined using FOR and NEXT are easy to spot, especially if the variable name is put after NEXT.
  4. Because the NEXT statement can jump into the middle of a line, an entire loop can be written on just one line.
  5. It is possible to perform a FOR-NEXT loop in the immediate mode.
- Although the value of the counting variable should never be changed while it is in the middle of a loop, the value can be used by other statements in the loop.
- If the STEP keyword is omitted in the FOR statement, a stepping value of 1 is assumed.
- To make the counting variable count backward, a negative step value has to be used.
- When the NEXT statement checks if the value of the counting variable is still in range, it checks not just for equality,

but also for less than. This means that the counting variable does not have to exactly match the ending value for the loop to end.

- With a positive step value, the NEXT statement will repeat the loop if the counting variable is less than or equal to the loop's ending value. For a negative step value, a check for *greater than or equal to* is used.
- All of the statements in a FOR-NEXT loop will be executed once before the NEXT statement is ever reached. This means that even if the starting and ending values in a FOR statement are wrong, as in FOR K=1 TO −1 (no STEP −1), the loop will be executed at least once.
- There is no sense in using a step value of 0, because doing so creates an infinite loop.
- Any number of independent FOR-NEXT loops can be used in one program.
- FOR-NEXT loops can be nested, provided that the NEXT statements occur in exactly the opposite order of the corresponding FOR statements.
- An easy way to tell if loops are nested correctly is to draw lines connecting the FOR and NEXT statements. If the lines cross, the nesting is wrong, and a NEXT WITHOUT FOR error message is to be expected.
- The maximum number of FOR-NEXT loops which can be nested fluctuates, but the absolute maximum is nine levels. A program won't usually have to nest deeper than that.
- A FOR statement cannot use a counting variable that is currently being used by another FOR-NEXT loop.
- The use of a variable name after NEXT is optional but recommended, because it helps catch errors which otherwise might go unnoticed and cause problems later in the program.
- Jumping out of an unfinished FOR-NEXT loop can create problems that may not be immediately detected. Always allow FOR-NEXT loops to go to completion. If it is necessary to jump out of a loop, do not use FOR and NEXT. Instead, use assignment statements and conditional logic.
- The counting variable retains its value after a FOR-NEXT loop has ended.

Introduction to Sprite Graphics

With character and bitmapped graphics modes, both of which can use up to 16 colors, your Commodore 64 is capable of producing some very impressive graphics displays. You can even do animation on your Commodore 64, with sprites, by changing sprite shapes and positions.

First, let's look at animation using character graphics. You change the shape of a character by redefining the character set, but the 8 by 8 matrix is often too small to create an object. Several characters have to be used, and it's not always easy to move that many characters. Even with a single character, the positioning is restricted to the range of 40 by 25 characters. Character movement appears jerky, as in the "Hurkle" game, because the character jumps by 8 points. Character graphics has some drawbacks when it comes to animation.

Bitmapped graphics can be used to create big shapes. You can draw a shape of any size, starting at any of 320 by 200 positions. The only problem is that the drawing is too slow. Moving an object, which requires erasing every point and replotting it at a new position, is even slower. Animation is virtually impossible using bitmapped graphics.

In answer to these problems, the Commodore 64 supports sprite graphics, which combines the best features of character and bitmapping modes. Sprite graphics makes it easy for an object to change shape or position. Sprites are, unquestionably, the best graphics feature of the Commodore 64.

A sprite is like a *super character* because it is three characters wide and almost three characters high. This solves the size problem associated with normal character graphics. A sprite can also be positioned to start at any of the 320 by 200 points on the screen, and only three POKE statements are needed to change the position. This takes care of the movement problem. Already you can see that sprites are well-suited for animation.

One of the most exciting features of sprite graphics, though, is that sprites can be used at the same time as character or bitmapped graphics. A sprite is completely independent of the normal screen display. The sprite image appears on top of the normal characters and plotted points, as if it were superimposed on the screen. Because the sprite is separate, no erasing or redrawing needs to be done when the sprite is moved. The display behind the sprite is undisturbed. For a demonstration of what a sprite can do, enter and run the following program.

```
100 POKE 56,62 : CLR : REM RESERVE MEMORY :rem 237
110 SP=2040:SE=53269:SC=53287:SX=53248:SY=53249:SZ
    =53264                                  :rem 224
120 BN=248 : BL=BN*64 : REM SET BLOCK NUMBER AND S
    TARTING LOCATION                        :rem 253
130 POKE SP,BN : REM POINT TO BLOCK         :rem 216
140 FOR K=BL TO BL+20                        :rem 80
150 READ P : POKE K,P : NEXT K : REM READ DEFINITI
    ON INTO BLOCK                           :rem 190
160 FOR K=BL+21 TO BL+63                    :rem 231
170 POKE K,0 : NEXT K : REM CLEAR REMAINDER OF BLO
    CK                                       :rem 78
180 POKE SC,8 : REM SET SPRITE COLOR TO ORANGE
                                            :rem 129
190 POKE SE,1 : REM ENABLE SPRITE            :rem 90
200 FOR X=0 TO 290 : Y=X/2                    :rem 5
210 POKE SX,24+XAND255:POKE SZ,-(X>231):POKE SY,50
    +Y : REM SET POSITION                   :rem 134
220 NEXT X                                   :rem 43
700 DATA 255,255,255,240,0,15,240,0,15,240,0,15,24
    0,0,15,240,0,15,255,255,255             :rem 165
```

Let's take a look at exactly what happened in this program.

Line 100 sets aside some free memory for use with sprites. The only drawback is that there is now less free memory available to BASIC, which you will discover when you use the free memory function.

Line 110 sets assignments for hardware locations that are used for controlling sprites. You will need to use these locations in every program that uses sprites.

A sprite definition resides in a block of 64 bytes. These blocks are numbered from 0 to 255. In line 120, variable BN is set to the block number that is used by the program. To calculate the location of the block in memory, the value in BN must be multiplied by 64. The result gives you the location where the first byte of the sprite definition should be POKEd.

Location SP in line 130 is the *sprite pointer,* and tells the graphics chip which block number the sprite definition is stored in.

Lines 140 and 150 read the sprite definition from DATA statements and POKE it into the block.

The sprite definition block consists of 64 bytes. Since only the first 21 bytes are being used, the remaining bytes are set to 0 by the loop in lines 160 and 170.

Line 180 sets the color of the sprite. Any value from 0 to 15 can be POKEd into location SC. The color numbers are the same as those used for the border and background locations.

In line 190, POKE is used to activate, or *enable*, the sprite. If this POKE is not executed, the sprite image will not appear.

Line 200 begins the loop to move the sprite across the screen. A sprite is positioned using X and Y coordinates, just as in the bitmapped graphics mode. The point specified by the coordinates is the upper-left corner of the sprite. For purposes of demonstration, the Y position is calculated using X so that the sprite will move diagonally across the screen.

The three POKE statements that position the sprite on the screen are in line 210. The exact details of these statements are not important; all that matters is that they work.

Line 220 is the end of the movement loop, and the end of the program. Line 700 contains the data for the sprite definition.

With just a couple of changes, you can make the sprite move according to the joystick. Here are the two lines that need to be changed.

```
200 JS=NOTPEEK(56320):X=X+SGN((JSAND8)-(JSAND4)):Y
    =Y+SGN((JSAND2)-(JSAND1))                    :rem 24
220 GOTO 200                                      :rem 95
```

Notice that the sprite can be moved off the main screen area. The sprite will disappear behind the border. The program has not been written, however, to let you move the sprite completely off the screen. It may stop due to an ILLEGAL QUANTITY error or start acting in an unusual manner. You could add some lines to the program that would use IF-THEN statements to check the range of X and Y to make sure that the sprite stays in bounds.

The important things shown by the program are that a sprite can be easily positioned at any point on the screen, and that it is independent of the rest of the screen, so no erasing and replotting is necessary.

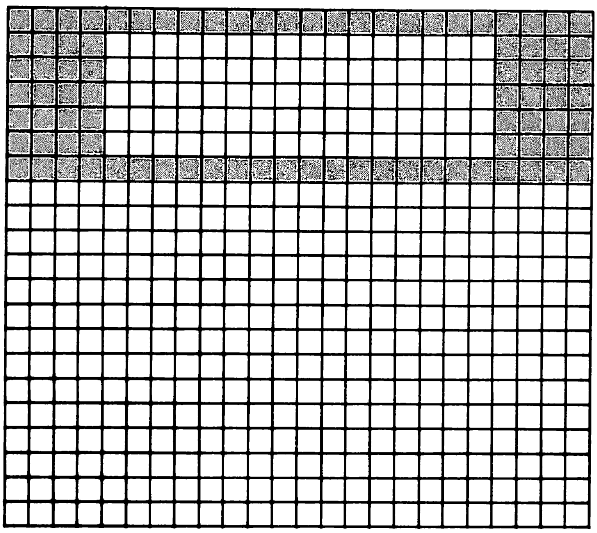
Defining a Sprite Shape

The shape used in the demonstration program was the outline of a box, but any other shape could have been used instead.

Simplified Loops

Now that you have seen what a sprite can do, let's see how the sprite shape is actually defined.

Defining a sprite is almost as easy as redefining a character. Just remember that a sprite is like a super character, because it is three characters wide and about two and a half characters high. Instead of working with an 8 by 8 matrix, you now have a 24 by 21 matrix. The illustration shows how the box image was created.

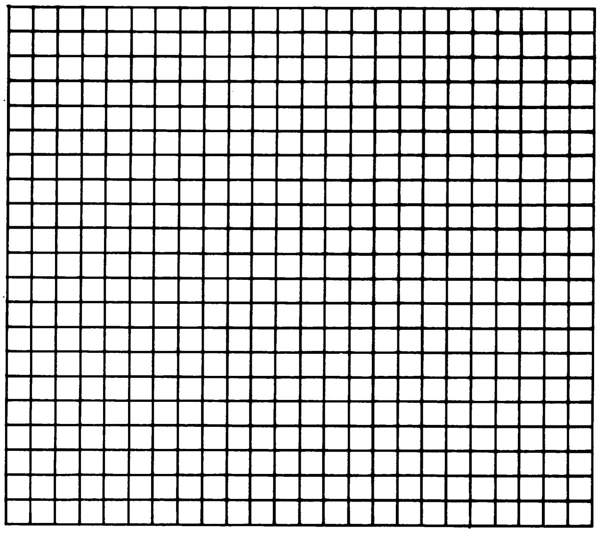


255	255	255
240	0	15
240	0	15
240	0	15
240	0	15
240	0	15
255	255	255
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

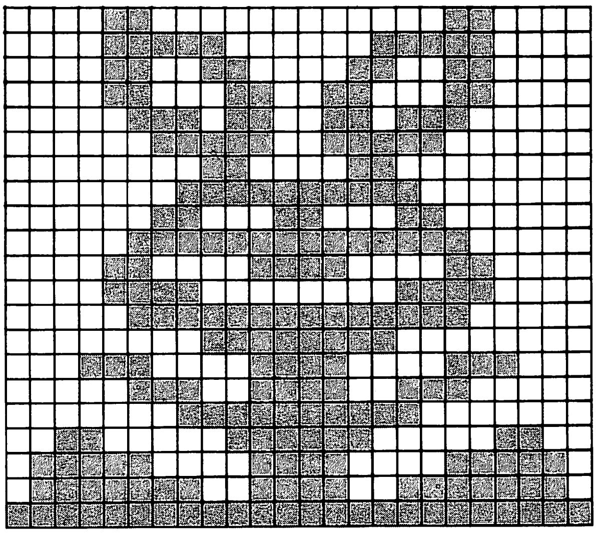
The sprite color is displayed at every point where a bit is set to 1. Where a bit is 0, the image is transparent, and the main screen shows through. Notice how screen characters appear in the window of the box.

The decimal numbers were calculated in the same way a character is redefined. These numbers were then copied into DATA statements to be read into the definition block by a FOR-NEXT loop. In the example, only the first 21 bytes were used, so the remaining bytes were set to 0. Since a total of 63 bytes are needed to define a sprite, byte 64 in the definition block is not used.

If you would like to define your own sprite shape, here is a step-by-step description of how it is done. Start with an empty 24 by 21 grid. Above it, write the numbers 128, 64, 32, 16, 8, 4, 2, and 1 in the appropriate columns. You will have to repeat this two more times, because the sprite is three bytes wide.



The next step is to fill in all the squares that form the sprite image.



Once you are satisfied with the picture you have drawn, you are ready to convert the bit patterns to decimal numbers. For each byte, look at only those bits which are set to one. Then look at the corresponding numbers at the top of the columns, and add those numbers together. The sum is the decimal equivalent for that byte. For example, the binary number 00001100 is 8+4=12 when converted to decimal. It is an easy process, but it will take a little while because you have to do it for all 63 bytes. Here are the decimal numbers for the example picture.

12	0	48
15	129	240
12	195	48
12	102	48
7	102	224
3	231	192
0	195	0
1	255	128
3	24	192
7	255	224
12	60	48
15	0	240
7	255	224
0	255	0
28	60	56
7	60	224
1	255	128
48	126	12
124	60	62
127	60	254
255	255	255

There should be a total of 63 numbers. Now you have to put the numbers into DATA statements. This is done on a row-by-row basis. The first three numbers in the first DATA statement come from the top row. In the example, the first three numbers would be 12, 0, and 48. Then you would move to the next row. When you run out of room in one DATA statement, use another one. The first DATA statement for the example would look like this.

```
700 DATA 12,0,48,15,129,240,12,195,48,12,102,48,7,102,224,
    3,231,192,0,195,0
```

All that is left is to write a program that reads the sprite data and POKEs it into a definition block. If the loop POKEs all 64 bytes in the block, however, remember to add the number 0 to the last DATA statement to avoid an OUT OF DATA error.

The demonstration program from before has been modified to display the sprite image we just created. All of the lines below line 130 have been changed.

```
100 POKE 56,62 : CLR : REM RESERVE MEMORY :rem 237
110 SP=2040:SE=53269:SC=53287:SX=53248:SY=53249:SZ
    =53264                                :rem 224
120 BN=248 : BL=BN*64 : REM SET BLOCK NUMBER AND S
    TARTING LOCATION                      :rem 253
130 POKE SP,BN : REM POINT TO BLOCK       :rem 216
140 FOR K=BL TO BL+63                      :rem 87
150 READ P : POKE K,P : NEXT K : REM READ DEFINITI
    ON INTO BLOCK                         :rem 190
180 POKE SC,3 : REM SET SPRITE COLOR TO CYAN
                                          :rem 235
190 POKE SE,1 : REM ENABLE SPRITE          :rem 90
200 JS=NOTPEEK(56320):X=X+2*SGN((JSAND8)-(JSAND4))
    :Y=Y+SGN((JSAND2)-(JSAND1))           :rem 116
210 POKE SX,24+XAND255:POKE SZ,-(X>231):POKE SY,50
    +Y : REM SET POSITION                 :rem 134
220 GOTO 200                               :rem 95
700 DATA 12,0,48,15,129,240,12,195,48,12,102,48,7,
    102,224,3,231,192,0,195,0              :rem 78
710 DATA 1,255,128,3,24,192,7,255,224,12,60,48,15,
    0,240,7,255,192,0,255,0               :rem 246
720 DATA 28,60,56,7,60,224,1,255,128,48,126,12,124
    ,60,62,127,60,254,255,255,255          :rem 55
730 DATA 0                                :rem 228
```

The significant differences between this and the earlier program are the change in shape, the new color, and the faster movement. The color was set to cyan by changing the POKE value in line 180 to 3. For the faster horizontal movement, the X offset was multiplied by 2 in line 200.

The new shape is larger than the original box, but it still doesn't cover much of the screen area. Another special feature is that a sprite can be expanded in the horizontal and vertical directions. The expansion is controlled by two hardware locations, so we need to assign two more variables. Change line 110 of the program to contain the two variable assignments.

SP=2040:SE=53269:SC=53287:SX=53248:SY=53249:
SZ=53264:SH=53277:SV=53271

Run the program again, and stop it so that the sprite image is on the screen.

Location SH controls the horizontal expansion of a sprite. When you enter the following line, the sprite will expand to twice its normal width.

```
POKE SH,1
```

The sprite is still 24 points wide, but each point has been doubled in width. This means that some of the detail is lost, and rounded parts do not appear quite so round anymore. Also notice that the expansion occurs to the right; the left edge is at the same place as before so that the X coordinate does not have to change. To return the sprite to normal width, enter this line.

```
POKE SH,0
```

The only horizontal expansion available is either normal or double width. To double the height of the sprite, try this line.

```
POKE SV,1
```

Every point is now twice as tall as before. The expansion occurs downward, so that the Y coordinate is still valid. The sprite can also be expanded horizontally at the same time, for an image that is four times the normal size. To set the height back to normal, use this statement:

```
POKE SV,0
```

Remember that the expansion feature affects only the way the sprite is displayed. The numbers in the definition block stay the same, and can only be changed by POKE statements.

There is one last feature related to sprite definitions. Just as character and bitmapped graphics support multicolor modes, so does sprite graphics. One POKE statement is all it takes to put a sprite into multicolor mode. However, since a sprite is separate from character and bitmapped graphics, there is a special location just for sprite graphics. The following statement turns on the multicolor mode for a sprite:

```
POKE 53276,1
```

To turn off the multicolor mode, use this statement.

```
POKE 53276,0
```

Simply turning on the multicolor mode is not enough to set the extra colors. The sprite must be defined in a certain way.

When a sprite is displayed in the multicolor mode, the bits are interpreted in pairs. Two bits correspond to each point. This means that a sprite is now only 12 points wide, so to compensate for the size, each point is wider. The sprites can still be positioned horizontally to start at any of the 320 points, though.

Defining a multicolor sprite takes a little more effort because of the way the bits are interpreted. Here is an example of a definition specifically designed for multicolor mode.

```
00000000 00000000 00000000
00000000 00000000 00000000
00000001 01010101 00000000
00000001 11111101 00000000
00000100 11111100 01000000
00000000 00110000 00000000
00000010 11111110 00000000
00001010 10101010 10000000
00001010 10101010 10000000
00101010 10101010 10100000
00100010 10101010 00100000
10000001 01010101 00001000
00000001 01010101 00000000
00000001 01010101 00000000
00000001 01000101 00000000
00000010 10001010 00000000
00000010 10001010 00000000
00000010 10001010 00000000
00000010 10001010 00000000
00000011 11001111 00000000
00000000 00000000 00000000
```

In multicolor mode, a sprite keeps its main color, and gets two new ones. Those points defined with a 10-bit combination will be displayed in the main color, set by location SC. Bit combinations 01 and 11 are set by locations 53285 and 53286. These are the sprite multicolor registers. Points defined by 00 are transparent and allow the normal screen to show through.

00	transparent
01	sprite multicolor 0 (location 53285)
10	sprite main color
11	sprite multicolor 1 (location 53286)

The example shape has been incorporated into the following demonstration.

```
100 POKE 56,62 : CLR : REM RESERVE MEMORY :rem 237
110 SP=2040:SE=53269:SC=53287:SX=53248:SY=53249:SZ
    =53264:SH=53277:SV=53271                :rem 28
120 BN=248 : BL=BN*64 : REM SET BLOCK NUMBER AND S
    TARTING LOCATION                        :rem 253
130 POKE SP,BN : REM POINT TO BLOCK         :rem 216
140 FOR K=BL TO BL+63                       :rem 87
150 READ P : POKE K,P : NEXT K : REM READ DEFINITI
    ON INTO BLOCK                           :rem 190
160 POKE 53285,7 : REM SET SPRITE MULTICOLOR ZERO
    {SPACE}TO YELLOW                        :rem 218
165 POKE 53286,10 : REM SET SPRITE MULTICOLOR ONE
    {SPACE}TO LIGHT RED                     :rem 35
170 POKE SC,13 : REM SET SPRITE COLOR TO LIGHT GRE
    EN                                      :rem 217
180 POKE SH,1 : REM SET TO DOUBLE WIDTH     :rem 168
185 POKE SV,1 : REM SET TO DOUBLE HEIGHT    :rem 244
190 POKE 53276,1 : REM TURN ON MULTICOLOR MODE
                                            :rem 96
195 POKE SE,1 : REM ENABLE SPRITE           :rem 95
200 X=200:Y=120                             :rem 28
210 POKE SX,24+XAND255:POKE SZ,-(X>231):POKE SY,50
    +Y : REM SET POSITION                   :rem 134
220 END                                     :rem 107
700 DATA 0,0,0,0,0,0,1,85,0,1,253,0,4,252,64,0,48,
    0,2,254,0,10,170,128                    :rem 34
710 DATA 10,170,128,42,170,160,34,170,32,129,85,8,
    1,85,0,1,85,0,1,69,0                    :rem 92
720 DATA 2,138,0,2,138,0,2,138,0,2,138,0,3,207,0,0
    ,0,0,0                                  :rem 127
```

The one sprite contains three colors that can be independently changed. Try some POKE statements in the immediate mode to create various color combinations.

The process of defining a sprite shape does require some effort, but the results can be well worth the time invested. You can create shapes and move them about the screen a whole lot easier using sprites than if you used character or bitmapped graphics.

Changing Sprite Shapes

One way to create the effect of animation is to have an object move across the screen. However, an object can be animated even if it is not moving, by changing its shape. So far, some of the example sprites have moved, but none of them has changed shape. This section shows you how easy it is to change the shape of a sprite.

The multicolor shape definition from the previous section has been modified to show a girl who is skipping rope. The definition is drawn so that the rope is passing over the girl's head. Here is the definition.

```
00000001 01010101 00000000
00000100 00000000 01000000
00010001 01010101 00010000
01000001 11111101 00000100
01000100 11111100 01000100
01000000 00110000 00000100
01000010 11111110 00000100
01001010 10101010 10000100
01001010 10101010 10000100
01101010 10101010 10100100
01100010 10101010 00100100
10000001 01010101 00001000
00000001 01010101 00000000
00000001 01010101 00000000
00000001 01000101 00000000
00000010 10001010 00000000
00000010 10001010 00000000
00000010 10001010 00000000
00000010 10001010 00000000
00000011 11001111 00000000
00000000 00000000 00000000
```

Here is a second definition, which has the girl jumping up and the rope passing under her feet.

```
00000000 00000000 00000000
00000000 01010100 00000000
00000001 11111101 00000000
00000001 11111101 00000000
00000001 00110001 00000000
00000010 11111110 00000000
00001010 10101010 10000000
00001010 10101010 10000000
00101010 10101010 10100000
00100010 10101010 00100000
10000001 01010101 00001000
```

```
01000001 01010101 00000100
01000001 01010101 00000100
01000001 01000101 00000100
01000010 10001010 00000100
00010010 10001010 00010000
00010010 10001010 00010000
00010010 10001010 00010000
00010011 11001111 00010000
00000100 00000000 01000000
00000001 01010101 00000000
```

The idea is that by altering between the two shapes, the girl will appear to be skipping rope. The only problem is in changing the sprite shape fast enough so that the picture changes instantly. Unfortunately, reading from DATA statements and POKEing into the definition block is not quite fast enough. The ideal solution would be to have two definition blocks, one for each definition, and then have the sprite alternate between the two blocks. Another great feature of the Commodore 64 is that it lets you use multiple definition blocks.

In the example programs, memory is reserved for not just one or two blocks, but eight. Instead of always using block number 248, you can use blocks from 248 to 255. The block number that contains the sprite definition is POKEd into location SP, which points to the definition block. This means that to make the sprite switch from one block to another, only one POKE statement is needed.

The next demonstration program puts the two rope-skipping definitions into blocks 248 and 249. The variable BN holds the current block number while the shape data is POKEd into the blocks. To alternate between the two images, the program alternately POKEs location SP with the numbers 248 and 249. When you run the program, it will actually appear as though the girl is skipping rope.

```
100 POKE 56,62 : CLR : REM RESERVE MEMORY :rem 237
110 SP=2040:SE=53269:SC=53287:SX=53248:SY=53249:SZ
    =53264:SH=53277:SV=53271                :rem 28
120 FOR BN=248 TO 249 : BL=BN*64 : REM SET BLOCK N
    UMBER AND STARTING LOCATION             :rem 38
130 FOR K=BL TO BL+63                       :rem 86
140 READ P : POKE K,P : NEXT K,BN : REM READ DEFIN
    ITION INTO BLOCK                       :rem 121
```

```
150 POKE 53285,7 : REM SET SPRITE MULTICOLOR ZERO
    {SPACE}TO YELLOW                             :rem 217
155 POKE 53286,10 : REM SET SPRITE MULTICOLOR ONE
    {SPACE}TO LIGHT RED                           :rem 34
160 POKE SC,13 : REM SET SPRITE COLOR TO LIGHT GRE
    EN                                           :rem 216
170 POKE SH,1 : REM SET TO DOUBLE WIDTH          :rem 167
180 POKE SV,1 : REM SET TO DOUBLE HEIGHT         :rem 239
190 POKE 53276,1 : REM TURN ON MULTICOLOR MODE
                                                  :rem 96
200 X=200:Y=120                                   :rem 28
210 POKE SX,24+XAND255:POKE SZ,-(X>231):POKE SY,50
    +Y : REM SET POSITION                        :rem 134
220 POKE SE,1 : REM ENABLE SPRITE                 :rem 84
230 POKE SP,248 : FOR K=1 TO 300 : NEXT          :rem 186
240 POKE SP,249 : FOR K=1 TO 300 : NEXT : GOTO 230
                                                 :rem 196
700 DATA 1,85,0,4,0,64,17,85,16,65,253,4,68,252,68
    ,64,48,4,66,254,4                            :rem 236
710 DATA 74,170,132,74,170,132,106,170,164,98,170,
    36,129,85,8,1,85,0                            :rem 27
720 DATA 1,85,0,1,69,0,2,138,0,2,138,0,2,138,0,2,1
    38,0,3,207,0,0,0,0,0                          :rem 37
730 DATA 0,0,0,0,84,0,1,253,0,1,253,0,1,49,0,2,254
    ,0,10,170,128,10,170,128                     :rem 237
740 DATA 42,170,160,34,170,32,129,85,8,65,85,4,65,
    85,4,65,69,4,66,138,4                        :rem 193
750 DATA 18,138,16,18,138,16,18,138,16,19,207,16,4
    ,0,64,1,85,0,0                                :rem 72
```

Using advanced techniques, it is possible to reserve more than eight definition blocks. For now, use only the blocks from 248 through 255. Also, remember that the variable BL depends on the value of BN, and must be recalculated every time BN changes before POKEing a definition into a new block.

Here is one more demonstration, this time of a stick figure doing jumping jacks. The program uses four blocks to create a very smooth animation effect.

```
100 POKE 56,62:CLR:PRINT CHR$(147)               :rem 143
110 SP=2040:SE=53269:SC=53287:SX=53248:SY=53249:SZ
    =53264:SH=53277:SV=53271                      :rem 28
120 POKE SC,8:POKE 53285,7:POKE 53286,0:POKE 53276
    ,1:POKE SH,1:POKE SV,1                       :rem 249
130 FOR BN=248 TO 251:BL=BN*64:FOR K=BL TO BL+62:R
    EAD P:POKE K,P:NEXT K,BN                     :rem 155
140 POKE SX,159:POKE SY,130:POKE SP,248:POKE SE,1
                                                  :rem 67
```

```
150 FOR BN=248 TO 251:POKE SP,BN:FOR S=1 TO 90:NEX
    T:NEXT                                  :rem 203
160 FOR BN=250 TO 249 STEP -1:POKE SP,BN:FOR S=1 T
    O 90:NEXT:NEXT                          :rem 102
170 GOTO 150                                :rem 103
500 DATA 0,0,0,0,32,0,0,168,0,0,16,0,0,220,0,0,220
    ,0,3,19,0,3,19,0,3,19,0                 :rem 145
510 DATA 3,19,0,0,16,0,0,220,0,0,204,0,0,204,0,0,2
    04,0,0,204,0,0,204,0,0,204,0            :rem 116
520 DATA 0,204,0,0,204,0,0,204,0,0,32,0,0,168,0,0,
    16,0,0,220,0,3,19,0,12,16,192           :rem 189
530 DATA 48,16,48,192,16,12,0,16,0,0,16,0,0,220,0,
    0,204,0,0,204,0,0,204,0,3,3,0           :rem 201
540 DATA 3,3,0,3,3,0,12,0,192,12,0,192,12,0,192,0,
    0,0,0,32,0,0,168,0,0,16,0               :rem 255
550 DATA 255,223,252,0,16,0,0,16,0,0,16,0,0,16,0,0
    ,16,0,0,16,0,0,220,0                     :rem 18
560 DATA 0,204,0,0,204,0,3,3,0,3,3,0,12,0,192,12,0
    ,192,12,0,192,48,0,48,48,0,48           :rem 226
570 DATA 0,0,0,192,0,12,48,32,48,12,168,192,3,19,0
    ,0,220,0,0,16,0,0,16,0                  :rem 147
575 DATA0,16,0,0,16,0                        :rem 37
580 DATA 0,16,0,0,16,0,0,220,0,0,204,0,3,3,0,3,3,0
    ,12,0,192,12,0,192,48,0,48               :rem 57
590 DATA 48,0,48,192,0,12,192,0,12          :rem 126
```

Chapter 14

Computed Execution

Computed Execution

The ON Statement

In Commodore 64 BASIC, variables and expressions can be used just about any place a number is needed, with one exception: line numbers. Sometimes a program has to do one of several things, depending on the value of a variable. Say, for example, that the variable A has possible values of 1, 2, and 3. It would be very handy if the statement GOTO 100+A would work. Lines 101, 102, and 103 would contain the statements to take the appropriate action according to the value of A. But the computer will ignore the +A in the statement GOTO 100+A, and we have to use a lot of IF-THEN statements. The next demonstration program shows how to set up the special function keys f1, f3, f5, and f7 so that each key changes the border to a different color.

```
10 PRINT "PRESS A FUNCTION KEY" : BD=53280 :rem 78
20 GET A$ : IF A$="" GOTO 20              :rem 241
30 IF ASC(A$)<133 OR ASC(A$)>136 GOTO 10  :rem 242
41 IF ASC(A$)=133 THEN POKE BD,2 : PRINT "RED" : G
   OTO 20                                  :rem 82
42 IF ASC(A$)=134 THEN POKE BD,4 : PRINT "PURPLE"
   {SPACE}: GOTO 20                        :rem 83
43 IF ASC(A$)=135 THEN POKE BD,8 : PRINT "ORANGE"
   {SPACE}: GOTO 20                        :rem 61
44 IF ASC(A$)=136 THEN POKE BD,5 : PRINT "GREEN" :
    GOTO 20                               :rem 241
```

The ASCII codes for the function keys are consecutive. Now we expand the program to include the keys f2, f4, f6, and f8, which will change the background color. Besides entering these new lines, add the statement BK=53281 to line 10, and change the 136 in line 30 to a 140.

```
45 IF ASC(A$)=137 THEN POKE BK,3 : PRINT "CYAN" :
   {SPACE}GOTO 20                         :rem 178
46 IF ASC(A$)=138 THEN POKE BK,9 : PRINT "BROWN" :
    GOTO 20                                :rem 23
47 IF ASC(A$)=139 THEN POKE BK,7 : PRINT "YELLOW"
   {SPACE}: GOTO 20                       :rem 107
48 POKE BK,13 : PRINT "LIGHT GREEN" : GOTO 20
                                          :rem 129
```

Computed Execution

All those IF-THEN statements seem a little redundant. Each one checks the same variable, and the value being checked always changes by 1. This program would do the same thing in less space if we could use the statement GOTO 40+ASC(A$)−132, but that doesn't work. So we do the next best thing. If you can't add a variable to a line number, how about having a variable choose one of several line numbers? In circumstances like these, where you have possible values starting at one and increasing by one, use the ON statement.

The syntax for this statement is the keyword ON, a numeric value (it can be an expression including variables), the keyword GOTO, and a line number list. The order of the line numbers is very important. If the numeric value is one, the computer will perform a GOTO using the first line number. A value of 2 causes execution to jump to the line indicated by the second line number, and so on. Here's the statement that should be added to the example program.

```
40 ON ASC(A$)-132 GOTO 41,42,43,44,45,46,47,48
                                           :rem 2
```

From lines 41 to 48, every IF-THEN statement could be removed. Here is what line 41 should look like.

```
41 POKE BD,2 : PRINT "RED" : GOTO 20       :rem 51
```

Because the ASCII codes for the special function keys start at 133, we subtract 132 to start the values at 1. Using ON-GOTO in this program replaces eight IF-THEN statements, takes less typing, and makes it easier to understand what the program does.

The line numbers in the list do not have to increment by a constant number. They could be various odd values, and do not even have to be in increasing order. A statement like ON A GOTO 110,130,5,77,999,3 is perfectly acceptable. Also, the same line number may appear several times in the same list. Perhaps values of 1 and 3 both require the same processing, so the same line number would appear in the first and third positions. Do be aware that consecutive commas in the line number list will be interpreted as having a 0 between them. So you cannot try to omit certain values from the range. Using a line number list like 110,130,,170 does not mean that execution will skip over the ON statement if a value of 3 is used. Rather, it will jump to line 0.

Because of the direct correspondence between the value after ON and the list of line numbers, you may wonder what happens if the value is so large that there are not enough line numbers in the list. Take the statement ON A GOTO 110,130,170 for example. When this is executed, the first thing the computer does is look at the integer portion of A. If A is 1, 2, or 3, execution will jump to the respective lines. But if A is 4 or more, the entire ON-GOTO statement will be ignored, and execution will continue with the next statement. This is the only time when you might put a statement after a GOTO on the same line. The same thing will happen if A is 0—execution will skip right over the ON-GOTO. But if the value is negative, an ILLEGAL QUANTITY error is the result.

Sometimes, if there are a lot of line numbers in the line number list, the ON statement will not fit on one line. The method to get around this problem is to use two ON statements. The first one handles the first line numbers, and the second one takes care of all the line numbers left over. Let's say that the possible values for A are from 1 to 8. The next couple of lines show how ON statements can be used together.

```
120 ON A GOTO 300,312,320,100,110                  :rem 59
125 ON A-5 GOTO 340,350,366                        :rem 55
```

The only way that execution will ever get to line 125 is if the value is greater than 5. Since five line numbers were in the first list, that number is subtracted from the value after ON in the second statement.

Finally, the ON statement can also be used to call subroutines. In place of the keyword GOTO, use GOSUB. Of course, RETURN statements will have to be used to end the subroutine call. Whichever is used, the UNDEF'D STATEMENT error will occur if a line number is used and there is no corresponding line in the program.

Summary

- The ON statement is used to get around the problem that line numbers after GOTO or GOSUB cannot be expressions or variables.
- The syntax for the ON statement consists of the keyword ON, a numeric value, the keyword GOTO or the keyword GOSUB, and a list of line numbers.

- When executed, the ON statement takes the integer portion of the numeric value, counts that far into the line number list, and then performs a GOTO or GOSUB to that line.
- If the value exceeds the number of line numbers in the list, or if the value is 0, the statement will be ignored. Execution will continue with the next statement, which may be on the same line.
- A negative value produces the ILLEGAL QUANTITY error.
- There are no restrictions on the line numbers in the list, other than that no variables or expressions are allowed. The line numbers do not have to be in increasing order or increment by the same number, although programmers have a tendency to use the numbers that way.
- The same line number can be used more than once in the same list.
- Consecutive commas are counted as meaning that a line number of 0 is intended. It is not possible to make ON work with only selected values in a range.
- One ON statement can often replace several IF-THEN statements. For this reason the ON statement is considered another form of conditional logic.
- Sometimes it is necessary to use a couple of ON statements together, if the line number list is too long. In such a case, the number of line numbers in the first line must be subtracted from the value in the second ON statement.
- Because the ON statement can perform a GOTO or GOSUB, it will cause the UNDEF'D STATEMENT error to be generated when a requested line is not found.

Multiple Sprites

What could be better than having eight different definition blocks for one sprite? Having eight sprites, of course. The Commodore 64 can support up to eight sprites, each with its own shape, color, width, height, and position.

The sprites are numbered from 0 to 7. The previous examples have used only sprite 0, and the POKE statements given earlier work only when using sprite 0 alone. Here are modified versions of the statements for use with all eight sprites. The variable SN stands for sprite number and ranges in value from 0 to 7.

To set a sprite's definition block pointer:
POKE SP+SN, block number

To set a sprite's main color:
POKE SC+SN, color number

To expand a sprite to double width:
POKE SH,PEEK(SH)OR2↑SN

To reduce a sprite to normal width:
POKE SH,PEEK(SH)ANDNOT2↑SN

To expand a sprite to double height:
POKE SV,PEEK(SV)OR2↑SN

To reduce a sprite to normal height:
POKE SV,PEEK(SV)ANDNOT2↑SN

To enable (turn on) a sprite:
POKE SE,PEEK(SE)OR2↑SN

To disable (turn off) a sprite:
POKE SE,PEEK(SE)AND NOT2↑SN

To position a sprite:
POKE SX+SN*2,24+XAND255
POKE SZ,PEEK(SZ)AND NOT2↑SNOR-(X>231)*2↑SN
POKE SY+SN*2,50+Y

To see eight sprites in action, here is a demonstration program:

```
100 POKE 56,62 : CLR : REM RESERVE MEMORY :rem 237
110 SP=2040:SE=53269:SC=53287:SX=53248:SY=53249:SZ
    =53264:SH=53277:SV=53271                  :rem 28
120 BN=248 : BL=BN*64 : REM SET BLOCK NUMBER AND S
    TARTING LOCATION                          :rem 253
130 FOR K=BL TO BL+63                          :rem 86
140 READ P : POKE K,P : NEXT K : REM READ DEFINITI
    ON INTO BLOCK                             :rem 189
150 FOR SN=0 TO 7                             :rem 101
160 POKE SP+SN,BN : REM POINT TO BLOCK        :rem 167
170 READ C : POKE SC+SN,C : REM SET SPRITE COLOR
                                              :rem 145
180 POKE SH,PEEK(SH)ANDNOT2↑SN : REM SET SPRITE TO
     NORMAL WIDTH                              :rem 98
190 POKE SV,PEEK(SV)ANDNOT2↑SN : REM SET SPRITE TO
     NORMAL HEIGHT                            :rem 184
200 POKE SE,PEEK(SE)OR2↑SN : REM ENABLE SPRITE
                                                :rem 1
210 NEXT SN                                   :rem 115
```

```
220 SN=INT(RND(Ø)*8)                             :rem 117
230 X=1Ø+SN*4Ø : Y=1Ø+RND(Ø)*16Ø                 :rem 1Ø3
240 POKE SX+SN*2,24+XAND255:POKE SZ,PEEK(SZ)ANDNOT
    2↑SNOR-(X>231)*2↑SN                          :rem 28
25Ø POKE SY+SN*2,5Ø+Y                            :rem 175
26Ø FOR K=1 TO 3ØØ : NEXT K : GOTO 22Ø           :rem 57
7ØØ DATA 12,Ø,48,15,129,24Ø,12,195,48,12,1Ø2,48,7,
    1Ø2,224,3,231,192,Ø,195,Ø                    :rem 78
71Ø DATA 1,255,128,3,24,192,7,255,224,12,6Ø,48,15,
    Ø,24Ø,7,255,192,Ø,255,Ø                      :rem 246
72Ø DATA 28,6Ø,56,7,6Ø,224,1,255,128,48,126,12,124
    ,6Ø,62,127,6Ø,254,255,255,255                :rem 55
73Ø DATA Ø                                       :rem 228
74Ø DATA 1Ø,7,4,13,5,3,8,15                      :rem 31
```

The demonstration used the same definition for all eight sprites, but each sprite can be displayed with a different shape if you add seven more definitions. The statement POKE SP+SN,248+SN would then be placed inside the loop based on the variable SN.

Sprites are independent of each other, except when two or more sprites are displayed in the multicolor mode.

To turn on the multicolor mode for a sprite:
POKE 53276,PEEK(53276)OR2↑SN

To turn off the multicolor mode for a sprite:
POKE 53276,PEEK(53276)AND NOT2↑SN

Each multicolor sprite has its own main color, but the two additional colors are shared among all of the sprites. In multicolor sprites, all points defined by the bit combination 01 get their color from location 53285, and the color specified in location 53286 will be displayed at every point defined by the bit combination 11.

00	transparent
01	sprite multicolor 0 (location 53285)
10	sprite main color (location SC+SN)
11	sprite multicolor 1 (location 53286)

This means that up to eight girls skipping rope might be displayed, each with a different dress color, but the hair and skin colors would all be the same.

The ability to display several sprites poses a new question. What happens when two or more sprites are in the same position at the same time? An overlapping of sprites is called a *collision*. When two sprites collide, the lower number sprite has *priority,* meaning that it will appear in front of sprites with higher numbers. Sprite number 0 has priority over all other sprites, and all other sprites have priority over sprite number 7. For a demonstration of priority, make the following changes to the example program:

```
180 POKE SH,PEEK(SH)OR2↑SN : REM SET SPRITE TO DOU
    BLE WIDTH                                :rem 49
190 POKE SV,PEEK(SV)OR2↑SN : REM SET SPRITE TO DOU
    BLE HEIGHT                              :rem 135
220 FOR SN=0 TO 7                            :rem 99
230 X=30+SN*30 : Y=100                       :rem 71
260 NEXT SN                                 :rem 120
```

Priority is not confined to just sprites. Usually, all of the sprites have priority over anything else displayed on the screen, such as characters or plotted points. It's as if there are two screens, the one with the sprites appearing in front of the one with the normal screen information. This priority can be changed, though.

To make a sprite appear in front of other screen data:
POKE 53275,PEEK(53275)AND2↑SN

To make a sprite appear behind other screen data:
POKE 53275,PEEK(53275)OR2↑SN

The priority feature lets you create a three-dimensional effect by giving the impression of depth to the screen.

Collisions between sprites often occur in games. Once a collision has occurred, the program usually has to take some special action, such as starting an explosion or changing an object's direction of movement. Because collisions are frequent and important events with sprites, the graphics chip keeps track of all sprites involved in collisions. Determining whether a sprite has hit another sprite is as simple as PEEKing a location.

To detect a sprite-to-sprite collision:
IF (PEEK(53278)AND2↑SN)<>0 THEN PRINT "SPRITE HAS COLLIDED"

Alternate form with reversed logic:
IF (PEEK(53278)AND2↑SN)=0 THEN PRINT "SPRITE HAS NOT COLLIDED"

The image parts of the sprites must overlap for the collision to be detected. Overlapping transparent sections do not count as collisions. It does not matter if the overlapping sprite images are the same color. Once location 53278 has been PEEKed, another collision will not be detected until the two sprites overlap again. If the sprites continue to overlap, a collision will continuously be detected. Collisions can also be detected when sprites are positioned off the main screen. It is good idea to PEEK the location at the beginnning of the program, and clear any previous collisions.

By replacing the PRINT statement with other code, a program can easily detect a collision between sprites, and quickly take the necessary action. Here is a demonstration program:

```
100 POKE 56,62 : CLR : REM RESERVE MEMORY :rem 237
110 SP=2040:SE=53269:SC=53287:SX=53248:SY=53249:SZ
    =53264                                   :rem 224
120 BN=248 : BL=BN*64 : REM SET BLOCK NUMBER AND S
    TARTING LOCATION                         :rem 253
130 POKE SP,BN : POKE SP+1,BN : REM POINT SPRITES
    {SPACE}TO BLOCK                           :rem 38
140 FOR K=BL TO BL+62                         :rem 86
150 POKE K,255 : NEXT K : REM SET EVERY BIT IN DEF
    INITION                                  :rem 144
180 POKE SC,3 : POKE SC+1,8 : REM SET SPRITE COLOR
    S                                         :rem 47
190 POKE SE,3 : REM ENABLE SPRITES ZERO AND ONE
                                             :rem 164
200 SN=0 : X=60 : Y=60 : GOSUB 300 : REM POSITION
    {SPACE}TARGET SPRITE                     :rem 137
210 SN=1 : X=30 : Y=30 : GOSUB 300 : REM POSITION
    {SPACE}MOVING SPRITE                     :rem 142
220 P=PEEK(53278) : REM CLEAR ANY PREVIOUS COLLISI
    ONS                                      :rem 137
230 JS=NOTPEEK(56320):X=X+2*SGN((JSAND8)-(JSAND4))
    :Y=Y+SGN((JSAND2)-(JSAND1))              :rem 119
240 GOSUB 300                                :rem 169
250 IF (PEEK(53278)AND2↑SN)=0 GOTO 230        :rem 53
260 PRINT"SPRITE HAS COLLIDED"                :rem 92
270 POKE SC+1,RND(0)*16 : GOTO{2 SPACES}270
                                             :rem 232
300 POKE SX+SN*2,24+XAND255:POKE SY+SN*2,50+Y
                                             :rem 107
310 POKE SZ,PEEK(SZ)ANDNOT2↑SNOR-(X>231)*2↑SN:RETU
    RN                                       :rem 116
```

Collisions can also occur between sprites and the normal screen graphics. Here are the statements to check if a sprite has collided with a character or plotted point.

To detect a sprite-to-background data collision:
IF (PEEK(53279)AND2↑SN)<>0 THEN PRINT "SPRITE HAS HIT BACKGROUND"

Alternate form:
IF (PEEK(53279)AND2↑SN)=0 THEN PRINT "SPRITE HAS NOT HIT BACKGROUND"

There is one exception to this last kind of collision. A collision between a sprite and a multicolor bit combination of 01 is not detected. This makes it possible to at least display something on the screen without interfering with collision checking.

Chapter 15

Arrays

Arrays

Using Arrays

The idea of consecutive numbers, which was the basis for the ON statement, can be taken one step further. Often in computing you will come across several different but related values. For example, a program may need to keep track of the horizontal size of each sprite. Normally this would take eight variables. You might name them H1, H2, and H3, and so on. But that's a lot of variables just to keep track of sprite widths, and there is no easy way to process them. Each one must be accessed by its own name, so they can't be accessed by a loop. It would be most convenient if you could say FOR K=1 TO 8 : PRINT HK : NEXT and have the computer print out the values of H1, H2, etc., but that won't work. All you would get is the value of variable HK printed eight times. This is a case when we want to use one name to refer to a bunch of different values. A normal variable can have only one value at a time, so we need something more powerful. What we need is an array.

An array is a group of values which have a common name but can still be accessed individually. To specify a particular value in an array, a number in parentheses must be placed after the array name. Such a number is called a *subscript.* Enter the following lines in the immediate mode to see how arrays and subscripts are used.

```
A(1)=35
PRINT A(1)
A(2)=28 : PRINT A(2)
A(2)=A(1)-20 : PRINT A(2)
B(7)=73 : B(10)=98
PRINT B(7),B(10)
```

As with variables, only the first two characters in an array name are significant. The characters must be followed by parentheses containing a subscript value, which can be a number, an expression, or a variable.

Because arrays and variables are different things, the names used for arrays are totally separate from those used for variables. You can have a variable named A and an array named A without there being any conflict. The similarity between variables and arrays is not complete, though. If you have two arrays, named A and B, the assignment statement

A=B will not copy every value from array B to array A as it does for variables. To assign the values of one array to another, each value will have to be copied individually.

The advantage of an array is that it provides an easy way of dealing with a lot of values, as long as the values are related and occur in some order. For example, we might keep track of all sprite widths in an array, storing the values for sprites 0–7 in order.

Since the subscript can be a variable, arrays are perfect for using with loops. For example, let's say that we want to copy the six values of array B to array A.

FOR K=1 TO 6 : A(K)=B(K) : NEXT

Six assignments were made in only half as many statements. To print every value in array A is even easier.

FOR K=1 TO 6 : PRINT A(K) : NEXT

Indeed, the combination of loops and arrays is powerful and is used often.

There are several limitations placed on subscripts. First, a subscript must be an integer number. If a noninteger value is given, only the integer portion is used.

Unless you specify otherwise, the computer will set up a default array to contain 11 values, with subscripts ranging from 0 to 10. Attempts to use subscripts beyond the size of the array, such as 17, cause the error message BAD SUBSCRIPT to be printed. Negative values result in an ILLEGAL QUANTITY error. Here are some examples of subscripts which would cause these errors.

X(11) YZ(−3) Q(1001)

When you first set up a numeric array, the default value for every item is 0. A common way to put initial values into an array is to include the values in DATA statements and place them into the array using a loop with a READ statement in it. The next program stores the frequencies for one octave of notes in the arrays FL and FH. An array used in such a manner is often called a *table*, and the subscript is an *index*. In the program, the musical note is used as an index into the table to get a certain frequency.

```
10 MV=54296:AD=54277:SR=54278:FL=54272:FH=54273:CT
   =54276:PW=54275                          :rem 200
20 POKE MV,15:POKE AD,85:POKE SR,168:POKE PW,8
                                            :rem 154
```

```
30 FOR K=Ø TO 7 : READ FL(K),FH(K) : NEXT : PRINT
   {SPACE}"PLEASE TYPE SOME NOTES           :rem 209
40 GET N$ : IF N$="" GOTO 40                 :rem 15
50 F=ASC(N$)-65 : IF F<Ø OR F>7 GOTO 40     :rem 142
60 PRINT N$ : POKE FL,FL(F) : POKE FH,FH(F)
                                            :rem 253
70 POKE CT,65 : FOR K=1 TO 15Ø : NEXT        :rem 8Ø
80 POKE CT,64 : FOR K=1 TO 5Ø : NEXT : GOTO 4Ø
                                            :rem 246
90 DATA 12,7,233,7,225,8,1Ø4,9,143,1Ø,218,11,78,13
   ,24,14                                   :rem 179
```

Notice how the keyboard buffer remembers keys when you press them too quickly. To let you play one full octave, the program supports a note H even though there is no such note in normal music.

Arrays have one more important similarity to variables: They can be used to store character strings.

```
1Ø INPUT "HOW MANY PLAYERS"; NP              :rem 81
2Ø IF NP<1 OR NP>1Ø OR NP<>INT(NP) GOTO 1Ø    :rem 7
3Ø FOR K=1 TO NP                             :rem 68
4Ø PRINT "WHAT IS THE NAME OF PLAYER"; K   :rem 239
5Ø INPUT NAME$(K) : IF NAME$(K)="" GOTO 4Ø:rem 158
6Ø NEXT                                     :rem 165
7Ø PRINT : PRINT "THESE PEOPLE ARE PLAYING:"
                                             :rem 99
8Ø FOR K=1 TO NP                             :rem 73
9Ø PRINT NAME$(K) : NEXT                     :rem 8Ø
```

The strings in an array can be treated just like normal character strings.

Summary

- An array is a group of numbers or character strings which have a common name but can be individually accessed.
- The same restrictions that apply to a variable name also apply to array names.
- Array names are separate from variable names.
- To assign one array to another, each value in the array must be copied separately.
- To specify one particular value in an array, a number in parentheses is placed after the array name. This number is called a *subscript*.

- Subscripts must be positive integer numbers. Subscript numbering starts with 0. A negative subscript produces an ILLEGAL QUANTITY error. A subscript that is too large produces the BAD SUBSCRIPT error. The default maximum subscript is 10.
- The values in an array are usually related to each other in some way, and are ordered according to the subscripts. The default values are the number 0 and the null string.
- The advantage of using an array is that it makes it easier to manage a lot of values. Because subscripts can be variables, arrays are ideal for use in loops.

The DIM Statement

If you want to set up an array to contain 11 values or less, you don't have to do a thing; the computer will create the array automatically. But if you need an array to handle more than 11 values, you must tell the computer the exact size of your array. This is done with the DIM statement.

The maximum subscript number for an array is called the *dimension* of the array. The default dimension for an array is 10. The DIM statement lets you dimension an array to handle any number of values, provided that there is enough memory for the computer to store them. After the keyword DIM comes the array name followed by parentheses. Inside the parentheses is the dimension value of the array. This can be a variable or expression, just as long as the final value is positive.

When you first run a program, no arrays are dimensioned. If you use an array before the DIM statement, that array will be dimensioned to the default value of 10. Then, if you later try to dimension the array using DIM, you will get the REDIM'D ARRAY error even though you never used DIM before. For this reason, DIM statements are usually placed at the beginning of a program, before values in the arrays are ever accessed. Also, once an array has been dimensioned in a program, it cannot be dimensioned a second time while the program is running. To do so causes the REDIM'D ARRAY error.

Now, let's take another look at the doodle program at the beginning of the book. The program uses an array dimensioned for 15, the maximum number that can be returned by the joystick. Using the joystick number as an index into the array, a number is obtained and added to the current screen address to get a new position on the screen. The result is a fast and colorful doodling program that doesn't use slow IF-THEN statements to process the joystick.

A program can have several arrays dimensioned at the same time, using different array names. In fact, programs need to do this so often that there is a shortcut to make this easier. The DIM statement can be followed by a list of array dimensions, and they can be of mixed types. The following line shows a typical DIM statement:

```
100 DIM N(100),NAME$(30),CNT(K),BUFF$(K)   :rem 124
```

The next program will take any number of words and alphabetize them. It does so by sorting the words alphabetically, using an advanced technique called a *bubble sort,* which is beyond the scope of this book, but is included here so that you can get an idea of what a program can do when arrays are used.

```
100 INPUT "HOW MANY WORDS"; WC                  :rem 236
110 IF WC<2 GOTO 100                            :rem 243
120 DIM W$(WC)                                  :rem 211
130 FOR K=1 TO WC                               :rem 113
140 PRINT "WHAT IS WORD NUMBER"; K              :rem 193
150 INPUT W$(K) : IF W$(K)="" GOTO 150          :rem 109
160 NEXT : PRINT "PLEASE WAIT..."                :rem 90
200 F=0 : FOR K=1 TO WC-1                       :rem 186
210 IF W$(K) <= W$(K+1) GOTO 230                :rem 243
220 W$=W$(K) : W$(K)=W$(K+1) : W$(K+1)=W$ : F=1
                                                :rem 183
230 NEXT                                        :rem 212
240 IF F=1 GOTO 200                             :rem 164
250 PRINT : PRINT "IN ALPHABETICAL ORDER:":rem 214
260 FOR K=1 TO WC                               :rem 117
270 PRINT W$(K)                                  :rem 61
280 NEXT                                        :rem 217
```

Finally, there is one more feature about arrays that has not yet been mentioned. All of the arrays used so far have been of one dimension: only one number for a subscript. Arrays can have many dimensions. A two-dimensional array would be dimensioned with a DIM statement like the one shown below.

```
100 DIM A(100,8)                                :rem 242
```

A particular value in the array would be accessed with a subscript that consisted of two numbers.

```
110 A(1,1)=64                                  :rem 89
```

You can picture such an array as being like the television screen. There are 25 rows of 40 characters each. Any point on the screen can be designated by a row and column number. Likewise, any value in the array A can be specified by two numbers. To print the entire contents of a two-dimensional array, nested loops could be used.

```
500 FOR R=1 TO 100                            :rem 112
510 FOR C=1 TO 8                                :rem 9
520 PRINT A(R,C)                              :rem 119
530 NEXT C,R                                  :rem 152
```

If you put only one number in the subscript for a two-dimensional array, you will get the BAD SUBSCRIPT error.

Two-dimensional arrays are useful for representing matrices in higher mathematics; three-dimensional arrays are rarely used.

Summary

- The *dimension* of an array is the maximum subscript number that can be used for the array.
- When a program first starts running, or the CLR statement is executed, all arrays are undimensioned. When an undimensioned array is accessed for the first time, the array is dimensioned to a default of 10.
- For those occasions when an array needs to be dimensioned larger than 10, the DIM statement must be used.
- To save memory, an array can also be DIMensioned smaller than 11.
- The syntax for the DIM statement is the keyword DIM followed by an array name list. Each array name in the list must be followed by parentheses containing the dimension numbers.
- If an array is one-dimensional, only one number will be inside the DIM statement parentheses, and only one number will be needed to form a subscript.
- A two-dimensional array requires two numbers in the DIM statement and subscript. These arrays are often thought of in terms of rows and columns.
- Arrays of more than two dimensions are possible.

- If a subscript contains too few or too many numbers for the dimension of an array, or if a subscript number exceeds the maximum number specified in the DIM statement, the BAD SUBSCRIPT error will occur.
- The only restrictions in dimensioning an array are that the computer must have enough unused memory to hold the array, and the array must not be dimensioned already.
- Trying to DIM an array when there is insufficient memory causes the OUT OF MEMORY error.
- Attempting to DIM an array that is already dimensioned causes the REDIM'D ARRAY error.

Chapter 16

Program Development

Program Development

Learning How to Program

The preceding chapters have given specific information about the various statements in BASIC. This last chapter shows you how to put the statements together to form a program. Since everybody has preferred methods of developing a program, the information presented here is mainly common sense and friendly advice.

The first step toward gaining proficiency in programming is to learn from existing programs. Take a look at the listing of a BASIC program with which you are familiar, and see how it works. One of the first things to do is to identify the variables used and figure out their purposes. Once you understand how the variables are used, it will be relatively easy to figure out what each line does. You will notice that the lines tend to occur in groups, with each group handling some part of the program.

The next thing to do is to modify the program, making a few simple changes to make it easier to use. Change colors and positions of objects, or printed messages. Change some inputs so that prompts are easier to read, and so that default values can be selected by pressing RETURN. A shoot-em-up game could be enhanced by putting in bigger, fancier explosions. Also, look for areas in the program that could be simplified. If you see some statements duplicated often, you might try putting them in a subroutine. Check whether any parts of the program are not used.

Once you have a pretty good understanding of the whole program, you are ready to start adding to it. If it's a game, perhaps a high-score or two-player option would be nice. If it is a data management program, you might think of some useful operation not currently supported.

After having made several modifications, you may find that the program has become so patched up and disorganized that it ought to be rewritten. This is a good way to ease yourself into programming, since you will be writing a new program, but you already know how it should work, what variables are needed, and so on.

There will come a time when you feel ready to take the plunge and write your own major program from scratch. This may seem to be a formidable undertaking, but it will be easier if you break the process down into three major steps. The first step is to design the program, to consider all the things that the program must do, how it will react to every possible input from the user, what variables will be needed, and so on. You may think of things like loops, input, and conditional logic, but keep it at a general level and do not consider any details.

The next step is to write the actual BASIC statements. If you have a good design for the program, this step will be rather easy. In fact, once you've acquired some experience, this step will seem almost trivial.

The third step is to detect and fix all of the errors that have crept into the program. Usually a program does not work perfectly the first time, because you mistyped a line or made a mistake in logic. These errors are called *bugs,* and the process of tracking them down and eliminating them is called *debugging* the program. The result will be a program that you can be proud of and will want to share with other people.

Since the three main steps of developing a program are so important, they are discussed in greater detail in the following three sections. The last section is an example program showing how a game was put together.

Specification and Design

This is the most important step in writing a program. With a short program, you can get away without doing any design work. You just sit down in front of the computer and start typing. But with larger programs, not having a design makes the entire task much harder. What you write will not be organized, and it probably won't work the way you expect. You will have a feeling of being lost, and will not be in control of the program. No amount of debugging can make up for a bad design, so this step should not be overlooked if you want your efforts to be successful. The bulk of the work in developing a program should be done in the design phase.

From a theoretical standpoint, a computer is a communication device. It moves information from place to place. This is normally referred to as *input* and *output.* As it moves the information, the computer performs some sort of process, as directed by the program. In a game, pushing a joystick makes things happen on the screen. A word processor takes text that was typed on the keyboard and formats it for printing on paper. If the program manages a mailing list, it might sort a large number of addresses according to zip code. A music-composing program processes note data and controls the appropriate hardware locations to play music. These are just a few examples of how a program processes information. Quite often, this process is repetitive, and the steps that form the process can be placed in a loop. Conditional logic is necessary to have some control over the loop. Input/output, repetition, and conditional logic are the three essential aspects of any major program.

From a practical standpoint, a computer is a general-purpose tool with a wide variety of applications. A particular task for the computer represents a problem, and the program is the solution. Before you can start writing the program, however, you must fully understand the problem. For every possible input by the user, you should know all of the possible responses by the program. You will want to have a firm idea of the things that the program will be expected to do, and the order in which it will do them. This part of the design phase is called the *specification* step, because you have to define or specify everything about the program that is of concern to the user.

The next step is to start considering some of the internal workings of the program. You will also want to begin writing things down. Now that you know what things the program is expected to do, you can begin to think about how it will do them. Here you establish the foundation of the program and begin putting things in order. Make a note of the questions that will be asked by the program, and the messages that it will print. This is the input/output aspect of the program.

There will be places where the program has to make some major decisions. Here is the conditional logic aspect of the program. As you put these things into a logical sequence, the program will start dividing itself into main sections. Often these main sections are defined by the major decisions. All of the sections will probably fit into some sort of main loop, which is where the repetitive aspect comes in. Information obtained at the beginning of the loop is used to distribute execution to one of several sections at the end of the loop.

Another very important step is to start declaring what pieces of information will have to be stored in variables. Generally speaking, only the major points of the program will be considered during the design phase. The key things are to establish the basic organization of the program, including the use of input/output, repetition, and conditional logic, and to identify the main variables that will be used. When you are done with the design phase, you will have a good idea in your mind of how the program will work, and a good guideline on paper of how it will be written.

Implementation

This is the step where you write all of the BASIC statements which form a program. If you have properly designed the program, the coding should be a snap after a little experience.

Let's start by examining the structure of a typical program. You know from the design phase that your program consists of several sections and probably has a main loop. We will get to the main part of the program in a moment, but the first thing to look at is the program initialization. Many programs contain lines at the beginning which are executed only once per program run. They assign initial values to variables, clear the screen, set up screen colors, redefine a character set, print a title message, and so on. This would be the logical place to put DIM statements, since you want to avoid the REDIM'D ARRAY errors. Remember that DIM supports an array name list, so only one DIM statement should be needed. This is also where you would put the assignments that must be made before using sounds or sprites.

After the initialization is finished, the program is ready to enter the main loop. The beginning of the loop gives the information that will be used by the rest of the loop. INPUT and READ statements would go here. Remember to print a prompt message before every input. Once the data has been fetched, IF-THEN statements can be used to perform some calculation, change a value, print a response, or other operation, according to the incoming data. If there is not enough room on one line, the IF-THEN statement may have to send execution to a later part of the program. Eventually, execution will reach the end of the loop and will jump back to the top.

There may be lines after the main loop, perhaps to take care of some cleanup work when the program ends. The main program terminates with an END statement.

The next thing in the overall structure is to reserve room for subroutines. If one subroutine calls another, you might want to put them next to each other.

The last part of the whole program is the DATA statements, which are stuck at the end so that they are out of the way of the main program. The order of these statements is important. Numbers which may be reread should be put first, so that they can be read right after a RESTORE statement.

This is the basic structure for the average program, and is bound to vary for individual applications. One helpful technique is to structure the program parts according to line numbers. For example, initialization could start at line 100, the main loop at 300, subroutines at 600, and DATA statements at 800. This makes it easy for you to find your way around a program. It is also a good practice to put REM statements at the head of each section of code.

As for writing the statements that form the sections of the program, this is simply a skill that must be acquired through practice. You can use the example programs in this book as a guide. In most cases, there is an obvious way of combining the statements in an order that makes sense.

One of the great secrets to programming is simplicity. Always strive to keep your programs as straightforward as possible. If you try to impress others with mysterious, confusing logic that only you can figure out, you may someday get caught in your own trap. There is a certain elegance to a program that is written in a simple, direct manner. Such programs are usually smaller, run faster, are easier to understand and modify, and are less prone to errors.

Here is an example of funny logic that can clutter a program:

```
480 GOTO 550
......
550 GOTO 320
```

Little things like going to a GOTO statement can make program logic hard to follow. It is a good idea to use the GOTO statement sparingly, because an excessive use of GOTO can ruin the organization of a program. Here is another demonstration of lines that are less than straightforward:

```
320 IF A=B GOTO 340
330 PRINT "MESSAGE"
340 ......
```

Don't forget about the relational operators. An alternate method follows:

```
320 IF A<>B THEN PRINT "MESSAGE"
340 ......
```

A simple change lets you eliminate one GOTO statement, and in fact shortens the whole program by one line. Remember that IF-THEN is used to control the execution of one line, and GOTO is used to control execution of several lines. The only time you would need to use the first form in the example is when line 330 contains something too long to be merged with line 320. While you are first writing a program, it is desirable to keep lines rather short, leaving room for later additions.

Just as a program can become a tangled mess if you use too many GOTO statements, a lot of IF-THEN statements can seem redundant. Conditional logic is also possible with the ON statement, and you can often replace several IF-THEN statements with one ON statement. Or, if you find yourself typing the same sequence of statements over and over again, you could use a subroutine. In general, if you ever get the feeling that "there must be a better way," there usually is.

There are some methods that you can use to make a program run faster. Programs run faster when all unnecessary spaces are removed, and the computer can process values of variables faster than it can process constant numbers in a line. Therefore, the plotting subroutine could be speeded up by replacing all constants with variables that have been assigned the correct values. The initialization portion of the program would contain this line:

```
100 P=7:Q=2:R=40:S=248:T=504                    :rem 72
```

Here is the revised subroutine:

```
900 BY=BA+R*(YANDS)+(YANDP)+(XANDT):POKE BY,PEEK(B
    Y)ORQ↑(NOTXANDP):RETURN                     :rem 198
```

The order in which variables are first assigned also makes a difference. If the variable X is the first variable assigned as a program starts executing, it will be processed faster than any other variables.

Inefficient coding is magnified when it is used in a loop. Keep unneeded operations out of loops. The following two lines both clear out a sprite block, but the second one will execute faster:

FOR K=0 TO 63 : POKE BL+K,0 : NEXT K

FOR K=BL TO BL+63 : POKE K,0 : NEXT K

Be on the watch for instances where FOR/NEXT loops can replace loops using IF-GOTO. Here is a revised line-drawing subroutine:

```
910 DX=ABS(XØ-X):DY=ABS(YØ-Y):SX=SGN(XØ-X):SY=SGN(
    YØ-Y):IF DX<DY GOTO 95Ø                    :rem 217
92Ø E=DY-DX/2:FOR K=1 TO DX:IF E<Ø THEN E=E+DY:GOT
    O 94Ø                                       :rem 69
93Ø Y=Y+SY:E=E+DY-DX                           :rem 244
94Ø X=X+SX:GOSUB 9ØØ:NEXT K:RETURN             :rem 145
95Ø E=DX-DY/2:FOR K=1 TO DY:IF E<Ø THEN E=E+DX:GOT
    O 97Ø                                       :rem 75
96Ø X=X+SX:E=E+DX-DY                           :rem 244
97Ø Y=Y+SY:GOSUB 9ØØ:NEXT K:RETURN             :rem 151
```

A final word on FOR/NEXT loops is that they will execute faster if the FOR and NEXT statements can be put on the same line, and although it is not a good programming practice, NEXT:NEXT executes faster than NEXT J,K.

One other way to speed up a program is to invert its structure. The deeper in the program a line is, the longer the computer will take when it executes a GOTO or GOSUB to that line. The lines which need to be executed faster should be placed at the beginning of the program. Therefore, subroutines which are called often might be placed before the main loop, and initialization lines which are executed only once should be put at the end of the program.

Line numbers which have several digits can slow down execution. It is usually sufficient to start numbering lines at 100, rather than 1000. The shorter numbers will also require less typing.

Finally, the IF-GOTO statement executes just a little faster than IF-THEN with a line number.

Just as it is nice to make programs run faster, it is nice to be able to type them in faster. For one last typing shortcut, you should know that many BASIC keywords have abbreviations that can be typed using the SHIFT key. For example, the computer will interpret L and SHIFT-I as the LIST command. When a line entered using abbreviations is listed, the full names of the keywords are printed. This creates the only drawback to using abbreviations; it is possible to enter a line so that it will contain more than 80 characters when it is listed, which makes editing difficult. Refer to the *User's Guide* (Appendix D) for a chart showing all of the abbreviations.

With a little experience and practice, you will find it an easy matter to express a design in BASIC statements.

Debugging

Once you have completely entered your BASIC program, the first thing you should do is save it on tape or disk. Do this before running the program. This is a wise precaution in case the program contains a bug which will cause the computer to crash.

The odds are that your program will not work perfectly the first time. Typing mistakes, missing statements, logic errors, and false assumptions always manage to creep into a program. In the debugging phase, you identify and fix all of the errors that were introduced by the previous development steps.

One of two things can happen when you run a program that contains bugs. The program either aborts and prints an error message, or it keeps on running but produces the wrong results. The first situation is easiest to fix, so let's start with that. The error messages are diagnostic in nature, so they are your key to determining what is wrong. Here is a list of some of the most common mistakes.

SYNTAX. Parentheses in an expression or nested functions are not properly balanced.

Some necessary keywords are not present. Unless you are using IF-GOTO, the IF keyword must be followed by THEN. You cannot say A=1 to 10 without using the keyword FOR. Maybe you forgot the REM keyword in front of a comment line.

You tried to use a statement within a statement, as in PRINT READ P. The correct code should be READ P : PRINT P.

You pressed a nonnumeric key in response to a GET using a numeric variable.

The prompt string in an INPUT statement can be followed only by a semicolon, not a comma.

A line number is outside the range 0 to 63999.

ILLEGAL QUANTITY. The logical operators can work only on numbers in the range −32768 to 32767.

The location number for POKE or PEEK is outside the range 0–65535.

You tried to READ a character string into a numeric variable.

You tried to find the ASCII value of a null string.

The computed value for an ON statement is negative. Be careful when you have to use subtraction to get the computed value into range.

A negative number, or number greater than 32767, was used for an array subscript.

UNDEF'D STATEMENT. The line number referenced by a RUN command or GOTO or GOSUB statement does not exist.

OUT OF DATA. READ is used in an infinite loop, but there is no RESTORE.

You are off by one or more in the number of data elements. For example, you forgot the sixty-fourth byte in a sprite definition.

TYPE MISMATCH. You tried to assign a character string to a numeric variable, or a number to a string variable.

An arithmetic or logical operator has a character string as one of its operands.

The operands for a relational operator are not of the same type.

The argument of a function is not of the correct type.

RETURN WITHOUT GOSUB. Execution accidentally entered a subroutine, perhaps because there is no END statement.

NEXT WITHOUT FOR. The nesting order of variables is wrong. The order of variables used in FOR statements should be reversed in NEXT statements.

The same counting variable has been used in nested loops. This often happens when a subroutine contains a loop.

A FOR/NEXT loop has not been allowed to go to completion. Never jump out of a FOR/NEXT loop before it is finished.

OUT OF MEMORY. The program is too large for the amount of free memory. This can happen when memory is reserved for things like redefined character sets or sprites.

There are too many nested subroutines or FOR/NEXT loops. Check whether a subroutine is calling itself, or is never returning to the main program.

An array dimension is too large. Check the free memory.

BAD SUBSCRIPT. You forgot to dimension the array.
You tried to use a subscript larger than the dimension of the array.

REDIM'D ARRAY. You have already accessed an element in the array, causing the computer to set the default dimension to 10.

Once you have found the mistake, just edit the line and run the program again. If execution aborts at another place, you will have to repeat the above procedure.

Even when a program does not abort due to an error, it may not be working correctly. In the worst case, the computer may crash and not respond to the keyboard. These errors can be much harder to find, and you will have to be a detective in order to track them down. The biggest problem is that since execution never stops, you have little information about the nature of the error and where it occurs.

Therefore, the first step to take in debugging a program that keeps running but produces the wrong results is to press the RUN/STOP key. The message BREAK IN, followed by a line number, will be printed. Sometimes the line number is all you need to know to find the error. If the program stopped at a line which should not have been executed, check the listing to see how the program could have gotten to that point. Perhaps a GOTO or GOSUB statement with a wrong line number is the culprit.

If the program stopped at a line that should have been executed, you are again faced with a situation of having little information. This is when you should use PRINT in the immediate mode to display the values of important variables. Just as variables are keys to figuring out someone else's program, they are keys to finding errors in your own program. Print the value of a counting variable to see how many times a loop has been executed. Print the values of variables that should have been assigned only in the initialization section, such as for sound or sprite locations. If you get the value 0, you forgot to assign values. Or, if you get the wrong value, either the number in the initial assignment statement was typed incorrectly, or the variable is accidentally being changed by another part of the program. Examine all statements which assign the variable in question. The only statements which can change the values of variables are LET, READ, GET, INPUT, FOR, NEXT, and CLR. Also, remember that only the first two characters of variable names are significant.

Once you have found a variable that has an unlikely value, you are probably pretty close to finding the actual mistake. You may want to gather some more information by running and stopping the program several times. The general strategy in debugging is to collect information, such as by printing the values of important variables in order to narrow down the possible causes until you have found the mistake.

Perhaps you have stopped a program but have not found anything wrong, or maybe you have found some variables with suspicious values and would like to see what happens to them later in the program. In such instances you can use the CONT command to make the program continue executing at the exact point where it left off. Later you can stop the program again to see how the values of variables have changed. The CONT command can be used at any time in the immediate mode, provided that you have not caused an error or changed the program in any way. If the program stopped due to an error, or if you did so much as press RETURN while the cursor was on a program line, you will get the CAN'T CONTINUE error when you try to use CONT. Also, CONT can only be used to restart program execution. The command does not apply when you press RUN/STOP to abort a listing or save/load operation.

After stopping and restarting a program many times, you may discover some critical areas that require special attention. Since program execution happens so quickly, it is hard to press RUN/STOP when the computer is executing a particular line. To make execution stop at a specific statement in the program, use the STOP statement. STOP works just like END, except that it also prints the BREAK IN message with the line number. You can insert several STOP statements in one program, and continue after every STOP with CONT. Of course, you will want to remove all of the STOP statements when you are finished debugging the program.

If the STOP and CONT process seems to be taking too long, try replacing some STOP statements with PRINT statements that print the values of important variables. Now you will be able to see exactly what happens as the program runs.

Generally, the kinds of mistakes which prevent a program from working properly are very simple in nature. A common error is to forget the STEP −1 in a FOR/NEXT loop that goes backward. It is easy to forget the rules about precedence of operators. A person can stare for a long time at a statement like IF A=5 OR 6 THEN PRINT before realizing that it should be written IF A=5 OR A=6 THEN PRINT. Another error which is not immediately noticeable is a disparity between READ and DATA statements. If there are not enough numbers in DATA statements, you will get the OUT OF DATA error, but there is no error when some DATA numbers are left unread. A good way to detect errors like these is to use the immediate mode. For the READ error, stop the program at some point when all of the data should have been read. A READ statement in the immediate mode should produce an OUT OF DATA error, and something is wrong if it doesn't. For the IF-THEN statement or expression with many operators, assign the correct values to the variables and enter the line without a line number. Compare the actual results against what you expected would happen. This will help you spot logic errors sooner. The only statements that cannot be used in the immediate mode are GET and INPUT. An immediate mode GOTO statement is the same as a RUN command with a line number, except that the automatic CLR is not performed, and all variables retain their values.

As you are working on the program, you will keep flipping between the writing and debugging phases. It is a good practice to periodically save a copy of your current revision. You should not, however, have to go back to the design phase. If you come across some big oversights and have to make major changes to the program, it is an indication that you did not start with a good design.

It is difficult to prove that a program is entirely free of bugs. Once you think you have found them all, give the program to somebody else to try. You will be surprised at how many errors can be found when a program is tested and evaluated by a second person.

Example Program

As a final demonstration, here is a game developed in the manner described in the previous sections.

Specification

What is the object of the game? The player must survive an attack of menacing robots.

What will the screen look like? The screen will consist of red blocks representing robots, and one white ball representing the player. There will also be barriers at random places.

How will the player move? The player's movement will be controlled by the joystick.

How will the robots move? Each robot will move in the direction of the player, chasing the player across the screen.

How many robots will there be, and how quickly will they move? Fifteen robots should be sufficient. One robot will move per player move.

How will the player evade the robots? A robot will be destroyed if it crashes into a barrier. The player must maneuver so that a barrier is in the way of the robot as it follows the player. The player wins when all of the robots have been destroyed.

Will the player be able to wrap around the screen? No, a border will be drawn around the screen to prevent the player from moving out of bounds. If the player runs into a barrier or border, he will be destroyed.

What sound and graphics features will be used? The different objects will be color-coded. Different sounds will correspond to different events. For example, there will be a small explosion noise every time a robot is destroyed.

Design

The general structure of the program is a main loop which repeats indefinitely. The loop ends when either the player crashes or there are no more robots. The loop will consist of two main sections, one for player movement and the other for robot movement. If the player ever hits anything while moving, the game will end. Every time a robot moves, two things must be checked. If the robot hits the player, it is the end of the game. If the robot hits anything else, it is the end of the robot. The program will also contain sections to set up the screen and handle the player crash. Here is a more detailed look at how the whole program will be put together.

```
Initialization
    screen setup
    clear the screen
    draw border
    plot barriers
    plot robots
    plot starting position of player
```

```
main loop
     player movement
     remember current position
     calculate new position based on joystick
     check if anything already at new position
     if so, jump to end of game
     otherwise, safe to move, plot new position and erase old
robot movement
     remember current position
     calculate new position based on position of player
     check if anything already at new position
     if a player already there, jump to end of game
     if something else already there, robot crashes
     otherwise, safe to move, plot new position and erase old
end of game
     do sound and graphics effect, end
```

Because this is a two-dimensional game, it will be easier to deal with the object movement on a coordinate basis. The player and all of the robots will have their own X and Y coordinate values. The variables X and Y will hold the player's coordinates, and the arrays X and Y will hold the coordinates of the robots. In order to use the "plot new position, erase old position" method of character graphics animation, temporary variables TX and TY will also be needed.

For fast player-movement calculations, we will use the array method introduced in the doodling program at the beginning of the book. However, since coordinates are being used, two arrays, named SX and SY, will be needed. The number returned by PEEKing the joystick location is used as the subscript for the arrays. The array elements, 1, 0, and −1, are offset values to be added to the player's coordinates. Thus, when the stick is pushed right, returning a joystick value of 7, array elements SX(7) and SY(7) will contain the values 1 and 0, respectively.

To actually plot a character on the screen, an offset from the beginning of screen memory will have to be calculated, using the formula X+40*Y. The multiplication by 40 is necessary because every screen row consists of 40 bytes. Since this will be done rather often, the value 40 will be stored in a variable named F. The result of X+F*Y will be kept in the variable Z. The value Z must be added to the values S and C, the base locations for screen and color memory, to derive the correct POKE locations. We will continue to change a screen position's color before changing its character.

The POKE values for characters and colors will be kept in an assortment of variables. Variables P and PC are for the player's character and color, with R and RC for the robots and BR and BC for the barriers. The variable B will contain a 32, which is always used to erase characters.

Implementation

Here is the overall line-numbering structure.

110 initialization
200 screen setup
300 main loop—player movement
400 main loop—robot movement
500 player crash
600 subroutines
700 data

We will start with the initialization section. This is the code which is executed only once, when the program starts running.

```
110 X=0:Y=0:NR=15:DIMX(NR),Y(NR),SX(15),SY(15):
TX=0:TY=0
```

Variables X and Y are used frequently, so they are assigned first for fast processing. Variable NR contains the number of robots that the program will start with.

```
120 S=1024:C=55296:F=40:B=32:P=81:PC=1:R=160:
RC=8:BC=160:BC=3
```

These are the variables used for the character animation technique.

```
130 FOR K=5 TO 15:READ SX(K),SY(K):NEXT
```

The joystick never returns values less than 5, so the first elements in the arrays keep their default value of 0.

```
140 MV=54296:AD=54277:SR=54278:FH=54273:
CT=54276:PW=54275
```

These are the standard assignments for using sounds.

```
150 POKE MV,15:POKE AD,0:POKE SR,192:POKE FH,0:
POKE PW,8
```

Set to full volume, fastest attack, decay, and release, and high sustain level. POKEing FH with a 0 turns off any previous sound. The pulse width is set for a square wave.

Here is the code that sets up the screen before the game starts.

```
200 PRINT CHR$(147):POKE 53280,4:POKE 53281,6:FOR
K=0 TO 960 STEP F
```

```
210 POKE C+K,13:POKE S+K,BR:POKE C+39+K,13:POKE
S+39+K,BR:NEXT
```

Clear the screen, set the border to purple and the background to blue, and do a loop to draw the vertical borders.

```
220 FOR K=1 TO 38:POKE C+K,13:POKE S+K,BR:POKE
C+960+K,13:POKE S+960+K,BR:NEXT
```

Do the loop that draws the horizontal borders.

```
230 FOR K=1 TO 10
```

Start a loop which will plot ten barriers on the screen.

```
240 X=INT(RND(0)*38)+1:Y=INT(RND(0)*22)+1:
Z=X+F*Y:IF PEEK (S+Z)<>B GOTO 240
```

This line chooses a random position on the screen and checks if there is already a character there. If so, another location is chosen. However, we will want to do the same thing when we plot the robots and the player, so this line should be made into a subroutine. Be sure to fix the line number in the GOTO statement at the end of the line.

```
600 X=INT(RND(0)*38)+1:Y=INT(RND(0)*22)+1:
Z=X+F*Y:IF PEEK(S+Z)<>B GOTO 600
610 RETURN
```

Now we can put some more statements on line 230.

```
230 FOR K=1 TO 10:GOSUB 600:POKE C+Z,BC:POKE
S+Z,BR:NEXT
```

Ten barriers will be on the screen after this line has been executed.

```
240 FOR K=1 TO NR:GOSUB 600:X(K)=X:Y(K)=Y:POKE
C+Z,RC:POKE S+Z,R:NEXT:K=1
```

This line not only plots the robots but also sets the initial coordinates in the arrays. We will use the variable K to keep track of which robot is the next one to be moved, so K should be set to start at 1.

250 GOSUB 600:POKE C+Z,PC:POKE S+Z,P
This line plots the player. Variables X and Y contain the player's coordinates.

260 IF (PEEK(56320)AND16)<>0 GOTO 260
Execution will keep looping here until the player presses the joystick trigger to start the game.

Now we come to the main loop, starting with the player movement section.

300 TX=X:TY=Y:JS=PEEK(56320)AND15
Remember the player's old position and get the joystick direction value.

310 X=X+SX(JS):Y=Y+SY(JS):Z=X+F*Y:IF PEEK(S+Z)<>B GOTO 500
Calculate the new position according to the joystick. If there is something already there, jump to the section for a player crash.

320 POKE FH,4*X+3*Y:POKE CT,17:POKE C+Z,PC:POKE S+Z,P:POKE S+TX+F*TY,B
Generate a tone to indicate that the player is moving, and change the pitch according to the position. Plot the new position and erase the old one.

330 POKE CT,16
Stop producing the tone.

The robot movement is more complicated because only one of the robots should move every time through the loop, and there are more things to check at the new position.

400 TX=X(K):TY=Y(K):X(K)=X(K)+SGN(X−X(K)):Y(K)=Y(K)+SGN(Y−Y(K)):Z=X(K)+F*Y(K)
Taking the signum of the player position minus the robot position is what gives the robots the intelligence to chase the player. If the player is to the right of the robot, SGN(X−X(K)) will return a 1 to make the robot move to the right. Other player positions would return the values 0 or −1. This works similarly for vertical movement. All of the array references make this line rather complex. They will probably also slow down the program. With a little closer study, it becomes evident that the line could be rewritten as follows.

400 TX=X(K):TY=Y(K):X(K)=TX+SGN(X−TX):Y(K)=TY+SGN(Y−TY):Z=X(K)+F*Y(K)

This will do the same thing as the version above, only faster, because variables can be processed faster than array elements.

410 IF PEEK(S+Z)=P GOTO 500

If there is a player at the new position, it's the end of the game.

420 IF PEEK(S+Z)=B THEN POKE C+Z,RC:POKE S+Z,R:POKE S+TX+F*TY,B:GOTO 470

If there is nothing but a blank at the new position, it is okay to move there. Plot the new position and erase the old one, then skip around the following code.

430 POKE FH,RND(0)*30+10:POKE CT,129:POKE S+TX+F*TY,B:X(K)=X(NR):Y(K)=Y(NR)

The only way execution can reach this line is if the robot has hit something. Start the explosion noise, erase the old position, and then move the coordinates of the last robot in the array to this position. This is done to keep the array intact.

440 POKE CT,128:NR=NR−1:IF NR>0 GOTO 470

Turn off the noise. Decrement the number of robots. This shortens the array. If there are still some robots left, skip ahead.

450 POKE CT,65:FOR K=10 TO 100 STEP 10:POKE FH,K:FOR J=1 TO 30:NEXT J
460 POKE FH,100−K:FOR J=1 TO 30: NEXT J,K:POKE CT,0:END

The player has survived all of the robots. Perform a loop which plays a little victory sound effect, and then end the program.

470 K=K+1:IF K>NR THEN K=1

This is the line to which previous lines may have jumped. Increment the number of the current robot for the next time through the loop. If the robot just processed was the last one in the array, start at the beginning again.

480 GOTO 300

Jump back to the top of the loop.

The following code displays a simple explosion when the player crashes.

500 POKE CT,33:FOR K=1 TO 30

Prepare to generate some sounds and start a loop which will plot 30 points.

510 TX=X+INT(RND(0)*7)−3:TY=Y+INT(RND(0)*7)−3
Select a position that is up to three characters away from the player.

520 IF TX<0 OR TX>39 OR TY<0 OR TY>24 GOTO 510
If the position is out of bounds, try another one.

530 Z=TX+F*TY:POKE C+Z,RND(0)*16:POKE S+Z,42
Plot an asterisk at the calculated position.

540 POKE FH,RND(0)*50+10:NEXT:POKE CT,0
Choose a random pitch, loop back, and turn off the sound when done.

550 END

We already put a subroutine at line 600. All of the joystick offset values can fit into one DATA statement on line 700. Here is a complete listing of the program. Type it in, be sure to save a copy to tape or disk, and then run it.

```
110 X=0:Y=0:NR=15:DIM X(NR),Y(NR),SX(15),SY(15):TX
    =0:TY=0                                  :rem 107
120 S=1024:C=55296:F=40:B=32:P=81:PC=1:R=160:RC=8:
    BR=160:BC=3                              :rem 206
130 FOR K=5 TO 15:READ SX(K),SY(K):NEXT     :rem 203
140 MV=54296:AD=54277:SR=54278:FH=54273:CT=54276:P
    W=54275                                  :rem 239
150 POKE MV,15:POKE AD,0:POKE SR,192:POKE FH,0:POK
    E PW,8                                   :rem 225
200 PRINT CHR$(147):POKE 53280,4:POKE 53281,6:FOR
    {SPACE}K=0 TO 960 STEP F                  :rem 67
210 POKE C+K,13:POKE S+K,BR:POKE C+39+K,13:POKE S+
    39+K,BR:NEXT                              :rem 72
220 FOR K=1 TO 38:POKE C+K,13:POKE S+K,BR:POKE C+9
    60+K,13:POKE S+960+K,BR:NEXT             :rem 151
230 FOR K=1 TO 10:GOSUB 600:POKE C+Z,BC:POKE S+Z,B
    R:NEXT                                   :rem 229
240 FOR K=1 TO NR:GOSUB 600:X(K)=X:Y(K)=Y:POKE C+Z
    ,RC:POKE S+Z,R:NEXT:K=1                  :rem 110
250 GOSUB 600:POKE C+Z,PC:POKE S+Z,P          :rem 90
260 IF (PEEK(56320)AND16)<>0 GOTO 260        :rem 163
300 TX=X:TY=Y:JS=PEEK(56320)AND15             :rem 20
310 X=X+SX(JS):Y=Y+SY(JS):Z=X+F*Y:IF PEEK(S+Z)<>B
    {SPACE}GOTO 500                          :rem 241
320 POKE FH,4*X+3*Y:POKE CT,17:POKE C+Z,PC:POKE S+
    Z,P:POKE S+TX+F*TY,B                     :rem 159
330 POKE CT,16                               :rem 239
400 TX=X(K):TY=Y(K):X(K)=TX+SGN(X-TX):Y(K)=TY+SGN(
    Y-TY):Z=X(K)+F*Y(K)                       :rem 59
```

```
410 IF PEEK(S+Z)=P GOTO 500                          :rem 205
420 IF PEEK(S+Z)=B THEN POKE C+Z,RC:POKE S+Z,R:POK
    E S+TX+F*TY,B:GOTO 470                           :rem 239
430 POKE FH,RND(0)*30+10:POKE CT,129:POKE S+TX+F*T
    Y,B:X(K)=X(NR):Y(K)=Y(NR)                        :rem 121
440 POKE CT,128:NR=NR-1:IF NR>0 GOTO 470    :rem 229
450 POKE CT,65:FOR K=10 TO 100 STEP 10:POKE FH,K:F
    OR J=1 TO 30:NEXT J                              :rem 225
460 POKE FH,100-K:FOR J=1 TO 30:NEXT J,K:POKE CT,0
    :END                                              :rem 18
470 K=K+1:IF K>NR THEN K=1                           :rem 164
480 GOTO 300                                         :rem 104
500 POKE CT,33:FOR K=1 TO 30                         :rem 205
510 TX=X+INT(RND(0)*7)-3:TY=Y+INT(RND(0)*7)-3
                                                     :rem 110
520 IF TX<0 OR TX>39 OR TY<0 OR TY>24 GOTO 510
                                                     :rem 176
530 Z=TX+F*TY:POKE C+Z,RND(0)*16:POKE S+Z,42
                                                      :rem 73
540 POKE FH,RND(0)*50+10:NEXT:POKE CT,0     :rem 215
550 END                                              :rem 113
600 X=INT(RND(0)*38)+1:Y=INT(RND(0)*22)+1:Z=X+F*Y:
    IF PEEK(S+Z)<>B GOTO 600                         :rem 215
610 RETURN                                           :rem 119
700 DATA 1,1,1,-1,1,0,0,0,-1,1,-1,-1,-1,0,0,0,0,1,
    0,-1,0,0                                         :rem 135
```

Debugging

Assuming that you made no typing errors, there is only one bug in the program. When you run it, as soon as you press the trigger to start the game, you crash. You hear the random noises and see the colorful characters plotted in a zone around the player.

To get the program working correctly, we have to do some debugging. There are only two lines which jump to line 500, the player crash. The lines in question are 310 and 410. To determine which one is causing the early crash, temporarily delete line 500 by typing the number 500 and pressing RETURN. This will force an UNDEF'D STATEMENT error and will tell us which line is trying to jump to 500. Run the program again.

You should have gotten an UNDEF'D STATEMENT error in line 310. That means that our mistake is probably located around line 310. Take a look at the whole section. The only time line 310 will want to jump to 500 is when it finds something other than a blank at the new position. Since the player never moved, it couldn't have hit a robot or barrier. But wait—if the player didn't move, the new position is the same as the old position. The player crashed because it ran into itself! That's easy to fix. The joystick returns the value 15 when it is not being pushed in any direction. Here is a simple addition to line 300:

```
300 TX=X:TY=Y:JS=PEEK(56320)AND15:IF JS=15 GOTO
400
```

If the player is not moving, skip over all of the player movement code and move the next robot.

Restore line 500 and run the program again. You should have a working game.

The game becomes rather easy once you get the hang of it, but it serves as a good example of how to develop a program. You can experiment by changing the assignment to NR or the value after TO in the FOR/NEXT loop of line 230 to display more robots or barriers. To make the program automatically run again, replace every END statement with NR=15:GOTO 200. An obvious enhancement to the game would be to use a redefined character set.

Appendices

A Beginner's Guide to Typing In Programs

What Is a Program?

A computer cannot perform any task by itself. Like a car without gas, a computer has *potential,* but without a program, it isn't going anywhere. The programs published in this book are written in a computer language called BASIC. BASIC is easy to learn and is built into all Commodore 64s.

BASIC Programs

Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, or number is significant. A common mistake is substituting a letter such as O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, you must enter all punctuation such as colons and commas just as they appear in the book. Spacing can be important. To be safe, type in the listings *exactly* as they appear.

Braces and Special Characters

The exception to this typing rule is when you see the braces, such as {DOWN}. Anything within a set of braces is a special character or characters that cannot easily be listed on a printer. When you come across such a special statement, refer to Appendix B, "How to Type In Programs."

About DATA Statements

Some programs contain a section or sections of DATA statements. These lines provide information needed by the program. Some DATA statements contain actual programs (called machine language); others contain graphics codes. These lines are especially sensitive to errors.

If a single number in any one DATA statement is mistyped, your machine could lock up, or crash. The keyboard and STOP key may seem dead, and the screen may go blank. Don't panic—no damage is done. To regain control, you have to turn off your computer, then turn it back on. This will erase whatever program was in memory, *so always SAVE a copy of* *your program before you RUN it.* If your computer crashes, you can LOAD the program and look for your mistake.

Sometimes a mistyped DATA statement will cause an error message when the program is RUN. The error message may refer to the program line that READs the data. *The error is still in the DATA statements, though.*

Get to Know Your Machine

You should familiarize yourself with your computer before attempting to type in a program. Learn the statements you use to store and retrieve programs from tape or disk. You'll want to save a copy of your program, so that you won't have to type it in every time you want to use it. Learn to use your machine's editing functions. How do you change a line if you made a mistake? You can always retype the line, but you at least need to know how to backspace. Do you know how to enter reverse video, lowercase, and control characters? It's all explained in your computer's manuals.

In order to insure accurate entry of each program line, we have included a checksum program. Please read the article called "The Automatic Proofreader" (Appendix I) before typing in any of the programs in this book.

A Quick Review

1. Type in the program a line at a time, in order. Press RETURN at the end of each line. Use the delete and cursor keys to correct mistakes.

2. Check the line you've typed against the line in the book. You can check the entire program again if you get an error when you RUN the program.

How to Type In Programs

To make it easy to know exactly what to type when entering one of these programs into your computer, we have established the following listing conventions.

Generally, Commodore 64 program listings will contain words within braces which spell out any special characters: {DOWN} would mean to press the cursor down key. {5 SPACES} would mean to press the space bar five times.

To indicate that a key should be *shifted* (hold down the SHIFT key while pressing the other key), the key would be underlined in our listings. For example, <u>S</u> would mean to type the S key while holding the SHIFT key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (for example, {10 <u>N</u>}), you should type the key as many times as indicated (in our example, you would enter ten shifted N's).

If a key is enclosed in special brackets, [<>], you should hold down the *Commodore key* while pressing the key inside the special brackets. (The Commodore key is the key in the lower left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as necessary.

Rarely, you'll see a solitary letter of the alphabet enclosed in braces. These characters can be entered by holding down the CTRL key while typing the letter in the braces. For example, {A} would indicate that you should press CTRL-A.

About the *quote mode:* You know that you can move the cursor around the screen with the CRSR keys. Sometimes a programmer will want to move the cursor under program control. That's why you see all the {LEFT}'s, {HOME}'s, and {BLU}'s in our programs. The only way the computer can tell the difference between direct and programmed cursor control is the quote mode.

Once you press the quote (the double quote, SHIFT-2), you are in the quote mode. If you type something and then try to change it by moving the cursor left, you'll only get a bunch of reverse-video lines. These are the symbols for cursor left. The only editing key that isn't programmable is the DEL key;

you can still use DEL to back up and edit the line. Once you type another quote, you are out of quote mode.

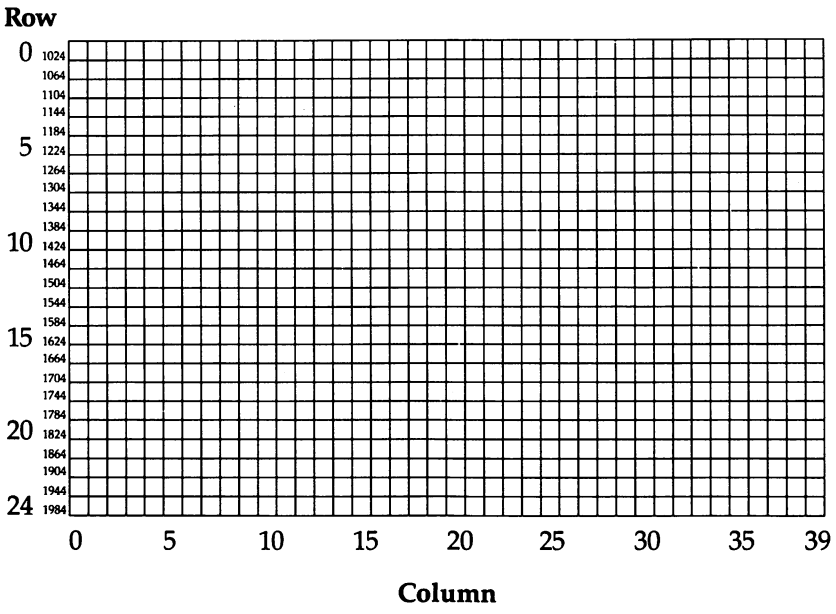
You also go into quote mode when you INSerT spaces into a line. In any case, the easiest way to get out of quote mode is to just press RETURN. You'll then be out of quote mode and you can cursor up to the mistyped line and fix it.

In order to insure accurate entry of each program line, we have included a checksum program. Please read the article called "The Automatic Proofreader" (Appendix I) before typing in any of the programs in this book.

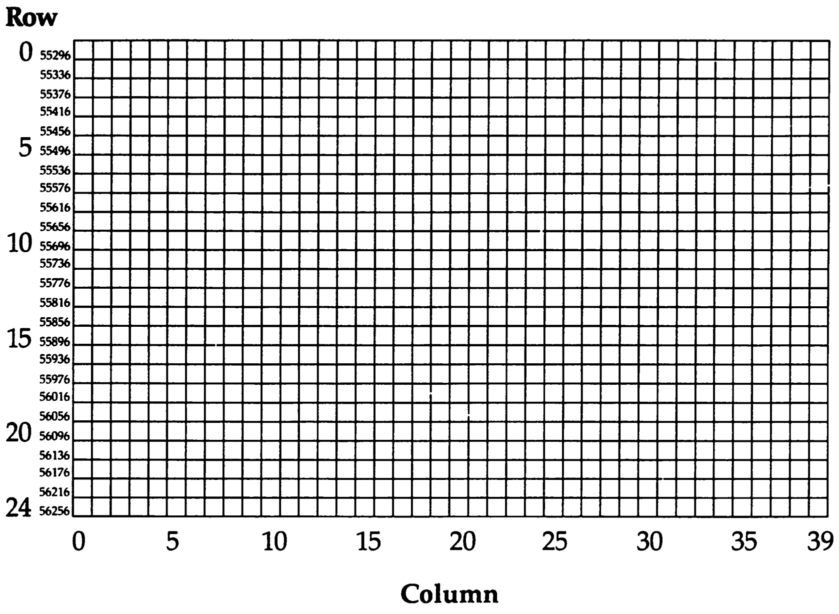
Use the following table when entering cursor and color control keys:

When You Read:	Press:		See:	When You Read:	Press:		See:
{CLR}	SHIFT	CLR/HOME	▨	Ɛ1Ǝ	COMMODORE	1	▨
{HOME}		CLR/HOME	▨	Ɛ2Ǝ	COMMODORE	2	▨
{UP}	SHIFT	↑ CRSR ↓	▨	Ɛ3Ǝ	COMMODORE	3	▨
{DOWN}		↑ CRSR ↓	▨	Ɛ4Ǝ	COMMODORE	4	▨
{LEFT}	SHIFT	← CRSR →	▨	Ɛ5Ǝ	COMMODORE	5	▨
{RIGHT}		← CRSR →	▨	Ɛ6Ǝ	COMMODORE	6	▨
{RVS}	CTRL	9	▨	Ɛ7Ǝ	COMMODORE	7	▨
{OFF}	CTRL	0	▨	Ɛ8Ǝ	COMMODORE	8	▨
{BLK}	CTRL	1	▨	{F1}		f1	▨
{WHT}	CTRL	2	▨	{F2}	SHIFT	f1	▨
{RED}	CTRL	3	▨	{F3}		f3	▨
{CYN}	CTRL	4	▨	{F4}	SHIFT	f3	▨
{PUR}	CTRL	5	▨	{F5}		f5	▨
{GRN}	CTRL	6	▨	{F6}	SHIFT	f5	▨
{BLU}	CTRL	7	▨	{F7}		f7	▨
{YEL}	CTRL	8	▨	{F8}	SHIFT	f7	▨

Screen Location Table



Screen Color Memory Table



Screen Color Codes

Value to POKE for Each Color

Color	Low nybble color value	High nybble color value	Select multicolor color value
Black	0	0	8
White	1	16	9
Red	2	32	10
Cyan	3	48	11
Purple	4	64	12
Green	5	80	13
Blue	6	96	14
Yellow	7	112	15
Orange	8	128	—
Brown	9	144	—
Light Red	10	160	—
Dark Gray	11	176	—
Medium Gray	12	192	—
Light Green	13	208	—
Light Blue	14	224	—
Light Gray	15	240	—

Where to POKE Color Values for Each Mode

Mode*	Bit or bit-pair	Location	Color value
Regular text	0	53281	Low nybble
	1	Color memory	Low nybble
Multicolor text	00	53281	Low nybble
	01	53282	Low nybble
	10	53283	Low nybble
	11	Color memory	Select multicolor
Extended color text†	00	53281	Low nybble
	01	53282	Low nybble
	10	53283	Low nybble
	11	53284	Low nybble
Bitmapped	0	Screen memory	Low nybble‡
	1	Screen memory	High nybble‡
Multicolor bitmapped	00	53281	Low nybble
	01	Screen memory	High nybble‡
	10	Screen memory	Low nybble‡
	11	Color memory	Low nybble

* For all modes, the screen border color is controlled by POKEing location 53280 with the low nybble color value.

† In extended color mode, bits 6 and 7 of each byte of screen memory serve as the bit-pair controlling background color. Because only bits 0–5 are available for character selection, only characters with screen codes 0–63 can be used in this mode.

‡ In the bitmapped modes, the high and low nybble color values are ORed together and POKEd into the *same location* in screen memory to control the colors of the corresponding *cell* in the bitmap. For example, to control the colors of cell 0 of the bitmap, OR the high and low nybble values and POKE the result into location 0 of screen memory.

ASCII Codes

ASCII	CHARACTER	ASCII	CHARACTER
5	WHITE	50	2
8	DISABLE	51	3
	SHIFT COMMODORE	52	4
9	ENABLE	53	5
	SHIFT COMMODORE	54	6
13	RETURN	55	7
14	LOWERCASE	56	8
17	CURSOR DOWN	57	9
18	REVERSE VIDEO ON	58	:
19	HOME	59	;
20	DELETE	60	<
28	RED	61	=
29	CURSOR RIGHT	62	>
30	GREEN	63	?
31	BLUE	64	@
32	SPACE	65	A
33	!	66	B
34	"	67	C
35	#	68	D
36	$	69	E
37	%	70	F
38	&	71	G
39	'	72	H
40	(	73	I
41	)	74	J
42	*	75	K
43	+	76	L
44	,	77	M
45	–	78	N
46	.	79	O
47	/	80	P
48	0	81	Q
49	1	82	R

Appendix F

ASCII	CHARACTER	ASCII	CHARACTER
83	S	120	♣
84	T	121	▕
85	U	122	♦
86	V	123	┼
87	W	124	▒
88	X	125	│
89	Y	126	π
90	Z	127	◥
91	[	129	ORANGE
92	£	133	f1
93	]	134	f3
94	↑	135	f5
95	←	136	f7
96	─	137	f2
97	♠	138	f4
98	│	139	f6
99	─	140	f8
100	▔	141	SHIFTED RETURN
101	▔	142	UPPERCASE
102	▁	144	BLACK
103	▏	145	CURSOR UP
104	▕	146	REVERSE VIDEO OFF
105	╮	147	CLEAR SCREEN
106	╰	148	INSERT
107	╯	149	BROWN
108	└	150	LIGHT RED
109	╲	151	GRAY 1
110	╱	152	GRAY 2
111	┌	153	LIGHT GREEN
112	┐	154	LIGHT BLUE
113	●	155	GRAY 3
114	▁	156	PURPLE
115	♥	157	CURSOR LEFT
116	▏	158	YELLOW
117	╭	159	CYAN
118	╳	160	SHIFT SPACE
119	○	161	▌

ASCII	CHARACTER	ASCII	CHARACTER
162	▄	200	│
163	▔	201	╮
164	▁	202	╰
165	▏	203	╯
166	▒	204	└
167	▕	205	╲
168	▒	206	╱
169	◤	207	┌
170	▕	208	┐
171	├	209	●
172	▗	210	─
173	└	211	♥
174	┐	212	│
175	▂	213	╭
176	┌	214	╳
177	┴	215	○
178	┬	216	♣
179	┤	217	│
180	▎	218	♦
181	▍	219	┼
182	▐	220	▒
183	▔	221	│
184	▀	222	π
185	▃	223	◥
186	┘	224	SPACE
187	▖	225	▌
188	▝	226	▄
189	┘	227	▔
190	▘	228	▁
191	▚	229	▏
192	─	230	▒
193	♠	231	▕
194	│	232	▒
195	─	233	◤
196	─	234	▕
197	─	235	├
198	─	236	▗
199	│	237	└

Appendix F

ASCII	CHARACTER
238	◰
239	▂
240	◱
241	⊞
242	⊞
243	⊞
244	▏
245	▌
246	▕
247	▔
248	▀
249	▄
250	□
251	▖
252	▝
253	◲
254	▘
255	π

0–4, 6, 7, 10–12, 15, 16, 21–27, 128, 130–132, and 143 are not used.

Screen Codes

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
0	@	@	31	←	←
1	A	a	32	-space-	
2	B	b	33	!	!
3	C	c	34	"	"
4	D	d	35	#	#
5	E	e	36	$	$
6	F	f	37	%	%
7	G	g	38	&	&
8	H	h	39	'	'
9	I	i	40	(	(
10	J	j	41	)	)
11	K	k	42	*	*
12	L	l	43	+	+
13	M	m	44	,	,
14	N	n	45	–	–
15	O	o	46	.	.
16	P	p	47	/	/
17	Q	q	48	0	0
18	R	r	49	1	1
19	S	s	50	2	2
20	T	t	51	3	3
21	U	u	52	4	4
22	V	v	53	5	5
23	W	w	54	6	6
24	X	x	55	7	7
25	Y	y	56	8	8
26	Z	z	57	9	9
27	[	[	58	:	:
28	£	£	59	;	;
29	]	]	60	<	<
30	↑	↑	61	=	=

Appendix G

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
62	>	>
63	?	?
64	─	─
65	♠	A
66	│	B
67	─	C
68	─	D
69	─	E
70	─	F
71	│	G
72	│	H
73	╮	I
74	╰	J
75	╯	K
76	└	L
77	╲	M
78	╱	N
79	┌	O
80	┐	P
81	●	Q
82	─	R
83	♥	S
84	│	T
85	╭	U
86	╳	V
87	○	W
88	♣	X
89	│	Y
90	♦	Z
91	┼	┼
92	▌	▌
93	│	│
94	π	▒
95	◤	▨
96	-space-	
97	▌	▌
98	▄	▄
99	▔	▔
100	▁	▁
101	▏	▏
102	▒	▒
103	▕	▕
104	▄	▄
105	◤	▨
106	▕	▕
107	├	├
108	▗	▗
109	└	└
110	┐	┐
111	▂	▂
112	┌	┌
113	┴	┴
114	┬	┬
115	┤	┤
116	▎	▎
117	▍	▍
118	▐	▐
119	▔	▔
120	▀	▀
121	▃	▃
122	┘	✓
123	▖	▖
124	▝	▝
125	┘	┘
126	▘	▘
127	▚	▚

128–255 reverse of 0–127

Commodore 64 Keycodes

Key	Keycode	Key	Keycode
A	10	6	19
B	28	7	24
C	20	8	27
D	18	9	32
E	14	0	35
F	21	+	40
G	26	–	43
H	29	£	48
I	33	CLR/HOME	51
J	34	INST/DEL	0
K	37	←	57
L	42	@	46
M	36	*	49
N	39	↑	54
O	38	:	45
P	41	;	50
Q	62	=	53
R	17	RETURN	1
S	13	,	47
T	22	.	44
U	30	/	55
V	31	CRSR↑↓	7
W	9	CRSR⇄	2
X	23	f1	4
Y	25	f3	5
Z	12	f5	6
1	56	f7	3
2	59	SPACE	60
3	8	RUN/STOP	63
4	11	NO KEY	
5	16	PRESSED	64

The keycode is the number found at location 197 for the current key being pressed. Try this one-line program:

10 PRINT PEEK (197): GOTO 10

The Automatic Proofreader

"The Automatic Proofreader" will help you type in program listings without typing mistakes. It is a short error-checking program that hides itself in memory. When activated, it lets you know immediately after typing a line from a program listing if you have made a mistake. Please read these instructions carefully before typing any programs in this book.

Preparing the Proofreader

1. Using the listing below, type in the Proofreader. Be very careful when entering the DATA statements—don't type an l instead of a 1, an O instead of a 0, extra commas, etc.
2. SAVE the Proofreader on tape or disk at least twice *before running it for the first time.* This is very important because the Proofreader erases part of itself when you first type RUN.
3. After the Proofreader is SAVEd, type RUN. It will check itself for typing errors in the DATA statements and warn you if there's a mistake. Correct any errors and SAVE the corrected version. Keep a copy in a safe place—you'll need it again and again, every time you enter a program from this book, *COMPUTE!'s Gazette,* or *COMPUTE!* magazine.
4. When a correct version of the Proofreader is RUN, it activates itself. You are now ready to enter a program listing. If you press RUN/STOP–RESTORE, the Proofreader is disabled. To reactivate it, just type the command SYS 886 and press RETURN.

Using the Proofreader

All listings in this book have a *checksum number* appended to the end of each line, for example, :rem 123. *Don't enter this statement when typing in a program.* It is just for your information. The rem makes the number harmless if someone does type it in. It will, however, use up memory if you enter it, and it will confuse the Proofreader, even if you entered the rest of the line correctly.

When you type in a line from a program listing and press RETURN, the Proofreader displays a number at the top of your screen. *This checksum number must match the checksum* *number in the printed listing.* If it doesn't, it means you typed the line differently than the way it is listed. Immediately recheck your typing. Remember, don't type the rem statement with the checksum number; it is published only so you can check it against the number which appears on your screen.

The Proofreader is not picky with spaces. It will not notice extra spaces or missing ones. This is for your convenience, since spacing is generally not important. But occasionally proper spacing *is* important, so be extra careful with spaces, since the Proofreader will catch practically everything else that can go wrong.

There's another thing to watch out for: If you enter the line by using abbreviations for commands, the checksum will not match up. But there is a way to make the Proofreader check it. After entering the line, LIST it. This eliminates the abbreviations. Then move the cursor up to the line and press RETURN. It should now match the checksum. You can check whole groups of lines this way.

Special Tape SAVE Instructions

When you're done typing a listing, you must disable the Proofreader before SAVEing the program on tape. Disable the Proofreader by pressing RUN/STOP–RESTORE (hold down the RUN/STOP key and sharply hit the RESTORE key). This procedure is not necessary for disk SAVEs, *but you must disable the Proofreader this way before a tape SAVE.*

SAVE to tape erases the Proofreader from memory, so you'll have to LOAD and RUN it again if you want to type another listing. SAVE to disk does not erase the Proofreader.

Hidden Perils

The proofreader's home in the 64 is not a very safe haven. Since the cassette buffer is wiped out during tape operations, you need to disable the Proofreader with RUN/STOP-RESTORE before you SAVE your program. This applies only to tape use. Disk users have nothing to worry about.

Not so for 64 owners with tape drives. What if you type in a program in several sittings? The next day, you come to your computer, LOAD and RUN the Proofreader, then try to LOAD the partially completed program so you can add to it. But since the Proofreader is trying to hide in the cassette buffer, it is wiped out!

What you need is a way to LOAD the Proofreader after you've LOADed the partial program. The problem is, a tape load to the buffer destroys what it's supposed to load.

After you've typed in and RUN the Proofreader, enter the following lines in direct mode (without line numbers) exactly as shown:

```
A$="PROOFREADER.T":B$="{10 SPACES}": FOR X = 1
TO 4: A$=A$+B$: NEXTX
FOR X = 886 TO 1018: A$=A$+CHR$(PEEK(X)): NEXTX
OPEN 1, 1,1,A$:CLOSE1
```

After you enter the last line, you will be asked to PRESS RECORD & PLAY on your cassette recorder. Put this program at the beginning of a new tape. This gives you a new way to load the Proofreader. Anytime you want to bring the Proofreader into memory without disturbing anything else, put the cassette in the tape drive. rewind, and enter:

```
OPEN1:CLOSE1
```

You can now start the Proofreader by typing SYS 886. To test this, PRINT PEEK (886) should return the number 173. If it does not, repeat the steps above, making sure that A$ ("PROOFREADER.T") contains 13 characters and that B$ contains 10 spaces.

You can now reload the Proofreader into memory whenever LOAD or SAVE destroys it, restoring your personal typing helper.

Incidentally, you can protect the cassette buffer on the Commodore 64 with POKE 178, 165. With this POKE, the 64 will not wipe out the cassette buffer during tape LOADs and SAVEs.

Automatic Proofreader

```
100 PRINT"{CLR}PLEASE WAIT...":FORI=886TO1018:REA
    A:CK=CK+A:POKEI,A:NEXT
110 IF CK<>17539 THEN PRINT"{DOWN}YOU MADE AN ERR
    R":PRINT"IN DATA STATEMENTS.":END
120 SYS886:PRINT"{CLR}{2 DOWN}PROOFREADER ACTIVAT
    D.":NEW
886 DATA 173,036,003,201,150,208
892 DATA 001,096,141,151,003,173
898 DATA 037,003,141,152,003,169
904 DATA 150,141,036,003,169,003
910 DATA 141,037,003,169,000,133
916 DATA 254,096,032,087,241,133
922 DATA 251,134,252,132,253,008
928 DATA 201,013,240,017,201,032
934 DATA 240,005,024,101,254,133
940 DATA 254,165,251,166,252,164
946 DATA 253,040,096,169,013,032
952 DATA 210,255,165,214,141,251
958 DATA 003,206,251,003,169,000
964 DATA 133,216,169,019,032,210
970 DATA 255,169,018,032,210,255
976 DATA 169,058,032,210,255,166
982 DATA 254,169,000,133,254,172
988 DATA 151,003,192,087,208,006
994 DATA 032,205,189,076,235,003
1000 DATA 032,205,221,169,032,032
1006 DATA 210,255,032,210,255,173
1012 DATA 251,003,133,214,076,173
1018 DATA 003
```

Index

What can my computer do? How can I make it do what I want?

All About the Commodore 64: Volume I answers these questions, and more, with a complete beginner's guide to BASIC programming.

You'll learn about the Commodore 64's powerful features and how to control them, gaining hands-on experience as you type in and run the many examples and sample programs that accompany the step-by-step explanations of each command.

Among the topics discussed in this book are:

- A detailed explanation of how every BASIC command works
- How to design, write, and debug your own programs
- Creating animated graphics displays with sprites
- Using loops and subroutines
- How to store your programs on tape or disk
- Creating and playing music on the computer
- Common mistakes and how to correct them
- Using joysticks in your programs
- Character string manipulation

You won't be a beginner for long. With *All About the Commodore 64: Volume I* as your guide, you'll soon become an experienced BASIC programmer.

ISBN 0-942386-40-X

All About the Commodore 64,® Vol. I

COMPUTE! Books