JAMES S. COAN
author of Basic BASIC

Microsoft™ BASIC

Using the SoftCard™

for Apple II Plus® and IIe®

HAYDEN

Microsoft™ BASIC Using the SoftCard™

Microsoft™ BASIC Using the SoftCard™

James S. Coan

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

Equipment Needed

- Apple II, Apple II Plus, or an Apple IIe computer with a minimum of 48K of memory
- Color television set or monitor
- Microsoft SoftCard (at least one disk drive required)

To achieve lowercase characters you will need one of the following:

- Apple IIe, an 80-column card, or an external terminal with an appropriate interface card.

Production Editor: TERRY DONOVAN
Art Director: JIM BERNARD
Cover Design: SHARYN BANKS
Original Artwork: JOHN McAUSLAND
Composition: BI-COMP, INC.
Printed and bound by: ARCATA GRAPHICS CO.:
FAIRFIELD GRAPHICS DIVISION

1 2 3 4 5 6 7 8 9 PRINTING

84 85 86 87 88 89 90 91 92 YEAR

Preface

Who knows how many versions of BASIC there are? All are recognizable as emanating from the original BASIC developed at Dartmouth College under John G. Kemeny and Thomas E. Kurtz. Most versions have added new features.

BASIC-80, developed by Microsoft, is in use on vast numbers of computers today. Therefore, if you learn Microsoft BASIC (also known as MBASIC or MS-BASIC), you will be able to use many computers. In addition, the BASIC we learn will be a good foundation for programming in other versions of BASIC and other computer languages everywhere.

Microsoft has developed BASIC in two modes: one is interpretive, and the other is compiled. The interpretive mode of BASIC is highly interactive. This interactive nature of BASIC has been a primary reason for its tremendous popularity. We can easily write programs and instantly command the computer to execute them for us. We can even command the computer to execute individual BASIC statements immediately right at the keyboard. Our mistakes and typing errors are often pointed out to us in plain English, rather than in undecipherable code. One of the drawbacks of BASIC is its slow execution speed. The reason for this slow speed is that interpretive languages must analyze and translate each statement into a more primitive form each time it is encountered during program execution. If a statement is executed fifteen thousand times, the interpreter must do the analysis and translation fifteen thousand times. The computer derives no benefit from the repetitious nature of the process (see, computers are not as smart as humans!). This interpretation takes more time than the actual program execution.

Compiled BASIC overcomes the slow-speed problem. The compiler performs the line-by-line analysis and translation once and creates an entirely new program, which may then be executed without the aid of the BASIC interpreter program. The time required to transform each program statement is eliminated.

The icing on the cake is that we have, in Microsoft BASIC, one language that may be managed in either mode. Thus it is possible to write

and thoroughly test BASIC programs interactively and then compile the result for more efficient use of computer resources.

This book is about programming in Microsoft BASIC-80 using the SoftCard and CP/M on Apple II, Apple II Plus, and Apple IIe computers. BASIC-80 is an enhanced version of the standard Microsoft BASIC. The primary enhancement is the inclusion of low-resolution and high-resolution graphics. Some of the additions are simply to include features for the convenience of Applesoft programmers. Problems included at appropriate points in the book make it suitable for the classroom as well as for the individual who wants to learn BASIC. Features of BASIC are generally introduced as they are needed to solve a task. Nearly one hundred programs are presented and discussed. The general approach is to begin with small programs to solve real problems and then to build them up into larger programs as required. It is helpful to break any task into small segments. One goal is to write program segments that will fit entirely on one screen. We even organize things in such a way that our list of tasks actually become the REMarks that form the program documentation before we begin to write program statements in BASIC. Larger programs are developed by first writing a control routine that will manage a collection of subroutines. Then we write the subroutines. Sometimes we have already written some of the subroutines to solve earlier programming problems.

We get started with simple calculations and printed messages in Chapter 1. Chapter 2 introduces numeric and string variables. The loop concept comes up in Chapter 3, with the formal introduction of FOR and NEXT in Chapter 4. We learn about functions and subroutines in Chapter 5. Chapter 6 presents numeric and string arrays. Here an alphabet game rounds out our work with string arrays. With many programming concepts and BASIC features in hand, we consider a few interesting applications in Chapter 7. A calendar program, the sieve of Eratosthenes, and different base numbering systems appear here. Chapters 8, 9, and 10 take us from a simple sequential-access file to a workable mailing-list program. Low-resolution and high-resolution graphics are the topics of Chapters 11 and 12.

Each chapter (except for Chapter 7) is followed by a "Sidelight" section that presents special features, concepts, and advanced techniques. Topics range from the use of a question mark for PRINT to how to use POKE to change the "Ok" message issued by BASIC after every task.

Appendix A compares SoftCard BASIC with Applesoft. EDIT Mode is discussed in Appendix B with numerous examples. Use of the disk for maintaining a library of programs is presented in Appendix C, and an ASCII chart is presented in Appendix D. An index of programs in the text appears in Appendix E. Appendix F contains solution programs for the even-numbered problems.

I wish to thank Alan Boyd of Microsoft Consumer Products for providing material assistance. Mike Violano and Chris Varley of Hayden Book Company, Inc., have made important suggestions and provided valuable counsel in the development of this book.

New Hope, Pa. JAMES S. COAN

To the Reader

Learning to program a computer can be very exhilarating. The thrill of seeing your first apparently complicated idea implemented in a clear, simple program is wonderful. You will be well advised to look upon the computer as something to be mastered and not as some impersonal monster that is out to do you in. Everything that the computer does is explainable and predictable. You should take care to evaluate the results that the computer produces; do not blindly accept computer results as faultless. That is not to say that the computer is going to make mistakes. In fact, under normal conditions, the computer will execute your instructions exactly. Mistakes in the results of a program execution are usually caused by errors in the instructions written by the programmer. Resist the temptation to blame anything other than your programming for incorrect or unexpected results.

No one need fear the computer. This is especially true when it comes to learning. We can work with the computer in total privacy. Errors that the computer reports to us become our secret. Nobody else needs to know what we did wrong. Once we have mastered the computer, we may confidently demonstrate our skill to all who will join in. When we learn to ride a bicycle or drive a car, someone is bound to know and we may be humiliated. This need not be the case with the computer. The computer will keep its secret if you will keep yours. The computer has truly infinite patience and will never raise its voice.

Learning to program a computer is not so complicated. You will probably find that an iterative process works best. Read some of the text, try some programs on the computer, and go back to read some more. There are certain things that you cannot possibly know without being told and certain things that make sense based on what is known so far. You will find that reading the text will help with writing the next program and that writing and executing a program will help with reading the text.

A program consists of a set of instructions that causes the computer to perform a task of our choosing. The process of writing those instructions for the computer is called programming. Programs do an amazing variety

of things. You may perform the simplest of arithmetic calculations or the most complex of mathematical manipulations. You can write programs to interact with the user. You may want to do this to make the computer play a game or to fill out a tax return. You may write a computer program to solve an algebra homework problem or organize a directory of names and addresses of your friends or business associates. And you may even be programming a computer just for the fun of it.

I hope that you are soon stimulated by your work in programming to bring to the computer your new and exciting problems to be solved. Above all, to be successful, you will have to be an active participant. Actually write programs and execute them. Then try to see how what you have learned fits into the picture of the BASIC language and programming in general.

Contents

Chapter 1

Chapter 2

Chapter 4

Chapter 5

Chapter 9

Random-Access Files ... 133

Chapter 10

Chapter 11

Chapter 12

Chapter 1

Getting Started

With some things it is best to jump in with both feet. We want to get to the point where we can write programs and see results as quickly as possible. The proliferation of computers has made programming available to the masses. It is becoming easier for all of us to learn about computers. At the same time, it is becoming more and more important for us to do just that.

We are going to write programs in the version of Microsoft BASIC-80 (also known as MBASIC or MS-BASIC) that is supplied with the SoftCard. The SoftCard is designed to plug into an Apple II computer. It also operates in the Apple II Plus and the Apple IIe.

With the SoftCard we get both BASIC and the CP/M (Control Program for Microcomputers) operating system required to run it. Two versions of BASIC are supplied. MBASIC is an enhanced version of the tried and true Microsoft BASIC-80. It is designed to incorporate important features of Applesoft that implement special capabilities of the Apple computers. The most significant enhancement is the low-resolution (Lo-Res) graphics instruction set. GBASIC adds high-resolution (Hi-Res) graphics to MBASIC. Unless the distinction is important we will always simply use the term BASIC.

To get started, we need a computer with the SoftCard plugged in according to the instructions supplied by Microsoft and a disk holding CP/M and BASIC (MBASIC or GBASIC). When we turn on the machine, we will hear some disk noise and see a little message revealing the version of CP/M on the screen followed by

```
A>
```

That symbol is the prompt familiar to anyone who has ever used the CP/M operating system. The prompt is a reminder to the user that nothing is going on—that the computer is ready for us to enter some instruction. The easiest way to use BASIC is to type

```
MBASIC
```

from the keyboard and press the RETURN key. Nothing will happen until we press the RETURN key. This will be on the same line with the CP/M prompt. It looks like this on the screen

```
A>MBASIC
```

If CP/M doesn't find what we want it will respond with

```
MBASIC?
```

Insert the proper disk and try again. That instruction tells CP/M to copy MBASIC from the disk to computer memory so that we may write programs in BASIC. After some more whirring, a message about the version of BASIC will appear on the screen. The last line is "Ok"! Suppose we want to get back to CP/M. The instruction SYSTEM is used for this purpose. When we issue this command, the CP/M prompt appears at the bottom of the screen. We are ready to begin.

....Introduction to Programming

Programming is the process of writing instructions to control a computer. Each programming language has its list of available instructions and its rules about how to put the instructions together. In BASIC many of the instructions use English words. This makes BASIC easy to learn. We run a program with RUN. We instruct the computer that we are at the end of the program with END. We tell the computer to stop everything right there with STOP. And so it goes. It is the programmer's job to select the most appropriate instructions from those available and put them together in a sensible and efficient fashion to solve the problem at hand.

In order for us to be assured that the computer has actually done something for us, we should always include some instructions to display a message. Therefore, we begin with the PRINT statement.

1-1...Displaying Messages (PRINT and LPRINT)

Here is a complete program:

```
10Ø  print "My first Microsoft BASIC program."
```

This is just a one-liner. It causes the computer to "PRINT" the message contained within quotes. If we are using a TV or other CRT video display, the message will last until we push it off the screen with additional messages. On a printer, the message will be printed on the paper for more permanent use. We will use LPRINT instead of PRINT whenever we want the display to go to the printer.

....RUN (Executing a Program)

If we type just this one line there will be no action. For the message to actually be displayed, we must command BASIC to "RUN" the program. This is done by simply typing RUN and by pressing the RETURN key. It looks like this:

```
RUN
My first Microsoft BASIC program.
Ok
```

Whenever BASIC types "Ok", that means it has completed its work. Nothing is going on anymore, and nothing else will happen until we make an entry.

Interactive BASIC obeys instructions in two ways—now and later. The PRINT statement of our program above is a "later" instruction. The RUN statement we used is a "now" instruction. These two types are called "deferred" and "immediate" instructions. The BASIC language consists of a collection of instructions and a set of rules governing their use. Each instruction is called a keyword. PRINT and RUN are keywords. Many instructions may be used for both immediate and deferred execution. The thing that determines whether we have a deferred instruction or an immediate instruction is whether it has a line number. If we give the statement a line number, it is a deferred instruction. Without the line number it is executed immediately. Instructions prepared for later execution make programs that can be retained for future use. The numbered program lines that we type are accumulated to form the complete program. Then the whole set of instructions may be reused. Immediate statements are one-shot statements; once used they are gone.

....LIST and LLIST

The LIST instruction causes the statements of the current program to be displayed just as they were entered at the keyboard. LIST displays right on the screen. LLIST sends the display out to the printer.

```
LIST
100  PRINT "My first Microsoft BASIC program."
Ok
```

Program 1-1. Our first program.

BASIC has converted "print" to "PRINT". BASIC converts all instructions to uppercase. If your Apple configuration has only uppercase, then it doesn't matter. But if you are using an Apple IIe, an 80-column card, or an external terminal then you might like to take advantage of lowercase display for some things. For some comments on an external terminal see Sidelight 11. Note that "P RINT" (with a space) will not be recognized, and you will have to retype the line. The words inside quotes are not instructions to BASIC. They remain exactly as we type them. Normally, any instruction must have a space before and a space after for BASIC to properly recognize it. Further, there must not be any spaces within the word itself. BASIC preserves the line just as we type it. This allows us to put in extra spaces between keywords to line things up nicely for easy reading.

Try it! You will benefit greatly from "hands-on" experience. If at all possible, sit down in front of a computer now and type in Program 1-1. Type exactly what you see above. If you make a typing error, there are lots of ways to correct it. You can press the left arrow as many times as it takes to fix the mistake. Or you can hold down the CTRL key and press H to achieve the same effect. CTRL-@ works too. Now suppose you have a line that is too messed up to fix this way. No problem: just hold down CTRL and press X. The line is canceled and you can begin again. At the end of the line, press the RETURN key—nothing will happen until RETURN is pressed. If the computer talks back to you, it is probably because you typed something wrong. If this happens to you, simply press the RETURN key once more and type the line correctly. Don't let this be upsetting. You can't hurt the computer with program instructions. This is one of the nice things about programming: no error can damage the computer. On the other hand, if you were to experiment with the electronics hardware, any little flaw or error could damage or destroy your machine. Appendix B discusses the wonderful things that we can do to recover from computer "back talk." Your attitude should always be that you know you are going to master the computer. Don't be put off by anything the computer does.

Now, back to our program. LIST it. To do this press five keys in turn: "L", "I", "S", "T", and "RETURN". RUN it. This one takes four keystrokes. We must always notify BASIC that we have completed the current line by pressing the RETURN key.

Next, type

```
100 PRINT "My second Microsoft BASIC program."
```

LIST it. We have replaced our program line with another one. RUN it.

Next, type 100 and press the RETURN key. LIST the result. Nothing is displayed—"Ok" appears by itself. We have eliminated our program line. Since our program consisted of only a single line, we have eliminated the entire program. Later we will see programs of many lines. It is imprac-

tical to eliminate entire programs a line at a time. Therefore, BASIC provides the keyword NEW for this purpose.

....NEW

NEW is used only when we wish to eliminate an entire program. Once we have entered the RETURN key after NEW any program we had is gone forever. Whenever we desire to begin a new programming project we should use the NEW keyword. Otherwise, we will simply merge our new program with our old one. This can be disastrous.

We may display any message by enclosing it within quotes in a PRINT statement or an LPRINT statement.

Let's consider some longer messages. What we are going to do here will come through very vividly if you can follow along by typing right into BASIC. Type

```
100 PRINT "Let's get some practice at"
110 PRINT "displaying messages.  We are"
120 PRINT "using the computer for the purpose of"
130 PRINT "improving our minds."
```

Program 1-2. Practice printing messages.

RUNning this program will produce the messages in quotes exactly as typed in the program. RUN it.

```
RUN
Let's get some practice at
displaying messages.  We are
using the computer for the purpose of
improving our minds.
Ok
```

Figure 1-1. Printing a four-line message.

Once again we see the "Ok" from BASIC, assuring us that all is well. Next, let's tinker with our little program. Let's eliminate the second line by typing

```
110
```

Now LIST it.

```
100 PRINT "Let's get some practice at"
120 PRINT "using the computer for the purpose of"
130 PRINT "improving our minds."
Ok
```

Program 1-3. Changing Program 1-2.

And the comforting "Ok" makes its appearance. (To avoid intense boredom we won't mention "Ok" in this context again.) When we RUN this new program the corresponding message is displayed.

```
Let's get some practice at
using the computer for the purpose of
improving our minds.
```

Figure 1-2. Execution of Program 1-3.

And finally we eliminate the new second line by typing

```
120
```

to produce Program 1-4. If we try to remove a line that isn't there, say 160, BASIC gently reports our error with the

```
Undefined line number
```

message.

```
100  PRINT "Let's get some practice at"
130  PRINT "improving our minds."
```

Program 1-4. Eliminate a line from Program 1-3.

Just as we should expect by now, another line has been removed, leaving a two-liner. We are getting a little experience here at simply manipulating a program. Let's RUN it.

```
Let's get some practice at
improving our minds.
```

Figure 1-3. Execution of Program 1-4.

We could easily restore the original program by retyping the two lines we eliminated earlier. This exercise should give us some idea of how to do some of the beginning things in BASIC. We have entered a program, LISTed it, RUN it, and modified it for further execution.

....Line Numbers

All the lines in our programs have been given line numbers. BASIC uses these line numbers to keep track of what it is doing at every moment. The line number serves to label or name the line. Thus, when we typed 110 followed by the RETURN key we informed BASIC that the line named "110" should be eliminated. And earlier when we typed 100 PRINT "My second Microsoft program." we directed BASIC to replace line 100 with a new line 100. BASIC programs are always arranged in order of increasing line number. The order in which we type the program lines doesn't matter.

Line numbers are important for program management. We may want to refer to line numbers in a LIST or RUN instruction. LIST 100 displays the single line for us to see. LIST 90-210 lists only those lines in the range 90 to 210. LIST 90,210 works just as well. The dash is standard for Mi-

crosoft BASIC. The comma is allowed for the convenience of Applesoft programmers. LIST-210 displays all of our program from the beginning through line 210. And LIST 560- produces all lines from 560 to the end of the program. We may even RUN 210 to begin program execution at line 210.

....DELETE

DELETE is a command that we use to purge lines in a program. We delete line 100 with

```
DELETE 100
```

And, to delete lines 305 through 389, we use

```
DELETE 305-389
```

We may delete from the beginning of the program to line 790 with

```
DELETE -790
```

But DELETE 600- is rewarded by the

```
Illegal function call
```

error message. This feature is really for our own protection! It is just too dangerous. For some reason, DELETE works on a protected program. (For more about protected programs, see Appendix C.) But we cannot enter those deleted lines again. For the convenience of Applesoft programmers, the SoftCard version of BASIC-80 allows DELETE to be shortened to DEL. Further, we may use a comma instead of the dash.

Let's make one more change in our message-printing program. As written, our program displays its message on two separate lines. It would be nice to have it produce a one-line display. In BASIC that is really very easy. We simply place a semicolon at the end of line 100. See Program 1-5.

```
100  PRINT "Let's get some practice at"[;]
130  PRINT "improving our minds."
```

Program 1-5. Two PRINT statements display on a single line.

The semicolon at the end of the PRINT statement in line 100 is an instruction to BASIC to continue further display on the same display line. Of course, we are limited by the screen width here. See Figure 1-4.

```
Let's get some practice atimproving our minds.
```

Figure 1-4. Execution of Program 1-5.

Oops! We really want a space between "at" and "improving". If we want a space, then we must include it in our instructions to the computer. Since

anything enclosed within quotes is displayed as typed, all we need to do is include a space at the appropriate spot. This we do in Program 1-6.

```
100  PRINT "Let's get some practice at ";
130  PRINT "improving our minds."
```

Program 1-6. Include the space this time.

Now RUN it.

....At the Keyboard

If you have been performing these exercises on your own computer, there is a fair chance that you have seen some disapproving message from BASIC. Probably you have witnessed the

```
Syntax error
```

message. "Syntax" simply refers to form. Everything we enter into the computer must have a correct form or syntax. If we enter an incorrect form, BASIC cannot determine what action to take. Thus, it reports a "Syntax error". Perhaps you typed "RIN" at the keyboard, intending to RUN a program. "RUIN" is a favorite. That doesn't work either. Or perhaps "LOST" instead of "LIST". No harm has been done. Simply type the instruction correctly and proceed.

If we enter a numbered program statement with such an error, something different occurs. Suppose we type something such as

```
100  PRONT "This is a sample error"
```

Nothing happens—until we execute the program. At that time, BASIC detects the problem and reports

```
Syntax error in 100
```

generously notifying us of the line where the error was found. Following this, BASIC goes into EDIT Mode by displaying the line number for us. At first in our programming career it may be best to simply get out of this new mode by pressing the RETURN key. We're doing enough new stuff as it is. Next, the beginner will replace the program line by retyping it. As you get a little more experience, you will want to use the magical powers of EDIT Mode. These powers are revealed in Appendix B. Check out EDIT Mode whenever you are ready.

....SUMMARY

When an Apple with a SoftCard is turned on with a CP/M diskette in the disk drive, the CP/M system is in control. Typing "MBASIC" puts Microsoft BASIC in control. Whenever we desire to return to CP/M from BASIC, we may type "SYSTEM".

We have spent some time here becoming familiar with BASIC by causing the computer to display word messages. This has given us a chance to see how to build programs and make changes in them. Programs are built up by typing numbered instructions having a syntax or form that we know BASIC can analyze and act upon. We change an existing program line by typing a new one with the same line number.

There are certain words that we may use in BASIC to instruct the computer. We have seen the keywords PRINT and LPRINT, RUN, LIST and LLIST, NEW, and DELETE or DEL.

We may display messages by enclosing them within quote marks in a PRINT statement. On a PRINTed line we may combine messages by separating them with a semicolon in PRINT statements. The computer will carry out the instructions of our program when we enter the RUN instruction. RUN 300 begins execution with line 300. We may examine our entire program with the LIST instruction. LIST 400, LIST 200-250, LIST -600, and LIST 440- are all forms of LIST that allow us to list selected lines of our programs. We erase a program from the computer with the keyword NEW. We may eliminate selected lines from our program using DELETE. DELETE 100, DELETE 100-190, and DELETE -300 are useful forms of the DELETE instruction.

Problems for Section 1-1 .....

Don't limit yourself to the problems listed here. As you begin to understand BASIC and programming, you will want to draw on problems of special interest to you. The process of learning to program a computer is unique in that the computer will provide you with some measure of your success. You don't need a teacher or an answer book to give important feedback on your progress. It is especially satisfying to be able to formulate your own problems, program their solution, and verify the result—all on your own.

1. Think up any message that you would like the computer to display—for example, "Now is the time for all good people to come to the aid of their country." LIST the program and RUN it. Try other messages.
2. Write a program to display the following message: "Programming is fun. The computer will solve many problems for us." Use two PRINT statements—one for each sentence. Have the message displayed on a single line. RUN it from just the second line using RUN (line number).

1-2...Calculations

The ability to display messages is crucial to good programming. Well-thought-out messages and result labels are very important. Every computer program should display some message.

The message is not the only thing. Often it is the ability of the computer to perform calculations with lightning speed that makes it so useful. Even if our real interest lies in graphics, games, voice, learning systems, word processing, or any other seemingly nonmathematical application, it is the arithmetic power of the computer that makes it perform so many different tasks. For this reason, it is important for us to learn how to direct it to calculate for us.

We can calculate an important number with a simple program such as Program 1-7.

```
100 PRINT 24 * 365
```

Program 1-7. Calculate hours in the year.

If we type this program in and RUN it we will instantly see the display of Figure 1-5.

```
8760
```

Figure 1-5. Execution of Program 1-7.

We can easily direct the computer to perform calculations right in the PRINT statement. We have used the "*" (asterisk symbol) to indicate multiplication. We might even want to use immediate mode to display such a simple result. We could enter the line

```
PRINT 24 * 365
```

In this case, we won't need to direct the computer to RUN the program. The result will be displayed instantly without it.

We really ought to dress up our program a little by including a label to tell us what that number is. All we have to do is add a quoted message. Program 1-8 does the job.

```
100 PRINT 24 * 365; "Hours in a year"
```

Program 1-8. Labeling a calculated result.

Here we have used the semicolon to place the message right on the same line as the calculated result. Look carefully at the execution of Program 1-8 in Figure 1-6.

```
RUN
 8760 Hours in a year
```

Figure 1-6. Execution of Program 1-8.

Note that we got a space between the 0 in 8760 and the H in Hours. BASIC always inserts a space following the display of a numeric value.

Program 1-9 is a simple program to demonstrate multiplication, addition, subtraction, and division of two numbers.

```
   110  PRINT "The numbers are  192  and  235"
-->120  PRINT
   200  PRINT "   The product is"; 192 * 235
   210  PRINT "       The sum is"; 192 + 235
   220  PRINT "The difference is"; 192 - 235
   230  PRINT "  Dividing we get"; 192 / 235
```

Program 1-9. Demonstrate simple calculations.

We can see in Program 1-9 that * is used to multiply, + is used to add, – is used to subtract, and / is used to divide.

```
The numbers are  192  and  235

   The product is 45120
       The sum is 427
The difference is-43
  Dividing we get .817021
```

Figure 1-7. Execution of Program 1-9.

Here in one short program we have done several calculations. How many calculations we might direct the computer to perform is limited only by the number of statements we are willing to type. Note that we used a blank PRINT statement in line 120 to improve the appearance of the program display. That decimal value for the division is necessarily approximate. If we do that problem out "all the way" we find a 46-digit repetition. In this case BASIC rounded off the result to 6 digits. That will be enough for many, many applications. Just in case we ever want more digits, be assured that we can easily obtain 16 digits (but we'll get to that later). And if we are willing to write a little program we can get as many as we like. Note that the three positive results are preceded by a space, while the negative value is indicated in the conventional manner.

We may direct the computer to perform arithmetic in any order by using parentheses. While 2 + 3 * 4 evaluates to 14, (2 + 3) * 4 evaluates to 20.

It is important to realize that very soon we will see more convenient ways to perform calculations on the computer. Right now we are trying to approach learning programming with a minimum of new detail at each step along the way. As each idea becomes familiar to us, then we will be ready to tackle the next feature or programming technique. We are going to learn to write programs by progressing from the known to the unknown.

....SUMMARY

So now we have the ability to display messages and perform arithmetic calculations in PRINT statements. We even have the ability to produce an empty line for attractive display using the blank PRINT. BASIC uses the

asterisk (*) to indicate multiplication. Plus (+), minus (−), and slash (/) are used for addition, subtraction, and division respectively. We rely on BASIC to provide six-digit precision without any special effort on our part.

Problems for Section 1-2

1. Write a program to calculate the sum of the counting numbers from 1 to 10.
2. Write a program to find the average of 78, 89, and 82.
3. Write a program to calculate the number of days since you were born.
4. Write a program to calculate the number of hours since you were born.
5. Write a program to find the simple interest at 11.98% on $4949 for one year.
6. Add 283.4, 658, 385.8, and 17.
7. Add $19234.30 and $123.45. Comment on the result. Remember that we said six digits.

SIDELIGHT 1

We presented some fundamentals in Chapter 1. In this section we will look at some extras.

....HOME

The SoftCard version of BASIC includes the HOME instruction especially for Applesoft programmers. HOME simply clears the screen and places the cursor at the upper left corner awaiting further instructions. This is very nice for presenting information clearly formatted for the reader. It is also nice for programmers who want to clear the screen and LIST a few program lines for careful study.

....Question Mark (?) in PRINT

A question mark may replace the keyword PRINT in any print statement. This saves four keystrokes every time we use it. The question mark may be used in both deferred and immediate modes. To find the number of hours in a year, type

```
?24*365
```

and the computer will respond with 8760 as quick as a flash.

Deferred mode has an added wrinkle. Suppose we type

```
10?24*365
```

Running this program will produce the expected result. But LIST reveals that something else has happened.

```
LIST
10 PRINT 24*365
```

We typed a question mark, but BASIC lists it as PRINT. No problem here. You have the choice of either typing ? or PRINT. The result is the same.

Chapter 2

Adding Features

Now that we have written a few programs and are familiar with the computer and BASIC, it is time to add some simple but powerful features. We will learn how to supply values for our programs to work on in a much more general way than in Chapter 1. The ability to remember values will be revealed. We'll be doing more with calculations and displaying messages.

2-1...More Calculations

It is your job to buy the eggs this week. That's not so tough: just go to the grocery store and buy some eggs. But you are also asked to get the best buy. So you read the paper and learn that small eggs sell for $.72 a dozen, medium eggs for $.87, large eggs for $.95, and extra large eggs for $1.00. Which should you buy? How do you decide which is the best buy? You probably want the lowest cost per ounce. So you want to know the weight per dozen. That is easy: look at the egg cartons. There you will find that the four sizes of eggs listed above weigh 18, 21, 24, and 27 ounces per dozen, respectively. Those figures are actually the minimum weights, but we'll use them for comparison. It is easy to determine the price in cents per ounce. The problem is solved for us by Program 2-1.

```
100 PRINT 100/27; 95/24; 87/21; 72/18
```

Program 2-1. Calculate egg values.

Here we have a one-line program to tell us which size eggs to buy to get the most egg for the money.

```
3.7037  3.95833  4.14286  4
```

Figure 2-1. Execution of Program 2-1.

Clearly we get the lowest per-ounce price by purchasing the extra large eggs. If the store is out of them, then large eggs are the ones to get. On the other hand, we often serve eggs individually, or we use recipes that call for a fixed number of eggs. In this case we usually expect medium eggs, and the computer's results will not change anything.

Program 2-1 may have solved a problem for us. It listed four numbers. By remembering that the sizes are listed decreasing from left to right we can interpret the results. However, it is a very primitive program. At the very least, we should have the computer label each of the values for us. One way to do this is to write four separate PRINT statements as shown in Program 2-2.

```
100  PRINT "Extra large"; 100/27
110  PRINT "      Large"; 95/24
120  PRINT "     Medium"; 87/21
130  PRINT "      Small"; 72/18
```

Program 2-2. Label egg values.

It is always a good idea to arrange programs and program output (display) in such a way that items are easy to read. The leading spaces within quotes in lines 110, 120, and 130 will help produce a nicely arranged report. The extra spaces following the semicolons on those same lines help to make the program easier to read. BASIC does convert all keywords to uppercase; otherwise any extra spaces we type for clarity (or even by accident) are left in. This feature gives us a chance to insert lots of space to make programs more readable. This is all for your benefit—the computer doesn't care.

```
Extra large 3.7037
      Large 3.95833
     Medium 4.14286
      Small 4
```

Figure 2-2. Execution of Program 2-2.

It might be nice to insert a line 90 to display a message announcing "Egg prices in cents per ounce" or something such as that.

Calculating cents to the nearest ten-thousandth isn't relevant to our problem. One of the things we will be looking for a little later on is a way to round off numeric results.

....Number Pigeonholes (Numeric Variables)

We can program the computer to perform many useful and interesting calculations using just the arithmetic available in PRINT and LPRINT statements. That gives us a hand-held calculator for the price of a computer. Tremendous additional problem-solving power is unleashed with each new programming feature available in BASIC. Thus far we have been using our video display or a piece of paper to save any results produced by a program.

BASIC contains the ability to save results within a program without having to display them. We simply think of a nice name for a value and tell the computer to remember a number by that name. Computer people call this name a variable. If we want to save a number, 765.50, representing a person's wages, then we might well ask the computer to use a variable called WAGES. We will have to remember that it is a measure of money. The computer won't do that for us. So, in a program, if we need to calculate a 2.4% wage tax we would use an expression such as

```
WAGES * .Ø24
```

And we could have the computer remember that value in another variable—perhaps TAX, or WTAX. We should select names that help us to remember what the number represents. For quick, short programs, though, we will often use just a single letter. This makes for faster typing.

There are some restrictions on what names we may use for variables. They must begin with a letter and be 40 or fewer characters. We may use any letters, any digits, and the decimal point in variable names. The variable name must not be a keyword or a BASIC instruction. Thus, names like NEW, PRINT, and LIST are no-no's. However, we may use OLDLIST or even NEWLIST if it suits our purpose. OLD.LIST, NEW.LIST, and PRINT.LIST are all legitimate variable names. This gives tremendous flexibility.

....The Assignment Statement (LET)

BASIC has a special statement that allows us to direct it to remember values. It is called the LET statement and can be used as follows:

```
2ØØ  LET WAGES = 765
```

To retain the value for taxes in TAX we would use a statement such as

```
22Ø  LET TAX = WAGES * .Ø24
```

The LET statement is referred to as the assignment statement because it causes the computer to assign a value to the variable named on the left of the equals sign.

Next, we might want to know how much is left and to save that value in NPAY or even NET.PAY.

```
230  LET NET.PAY = WAGES - TAX
```

Finally we ought to have the program display the result.

```
290  PRINT "Net pay is:"; NET.PAY
```

By putting these together we get Program 2-3.

```
-->200  LET WAGES = 765
   220  LET TAX = WAGES * .024
   230  LET NET.PAY = WAGES - TAX
   290  PRINT "Net pay is:"; NET.PAY
```

Program 2-3. First program with variables.

RUNning this program produces the output of Figure 2-3.

```
Net pay is:  746.64
```

Figure 2-3. Execution of Program 2-3.

We have created a program that consists of a sequence of instructions leading to a problem solution. This program may be saved and used again with another value for WAGES in line 200. By making a slight change in the program we may solve the same problem for any wages figure we like. To find the net pay for a person having wages of 635, simply replace line 200 in Program 2-3 with

```
200  LET WAGES = 635
```

and RUN the program. The LET statement is probably one of the most frequently used statements in BASIC.

....Optional LET

The use of the LET keyword itself is optional. We may use the statement

```
200  WAGES = 635
```

to assign the value 635 to the variable WAGES. This is still called an assignment statement, and it will perform in exactly the same manner as the equivalent LET statement. Many people strongly encourage the beginner to continue using the LET keyword until he or she is quite comfortable with programming. For this reason we will use LET for a little while longer in this book.

We may also cause the computer to READ values stored elsewhere in the program as DATA.

....READ and DATA

Earlier in our wages program we changed the value of WAGES by typing the LET statement in line 200 with the new number. We can simplify this process a little by using READ and DATA. Consider Program 2-4.

```
-->200  READ WAGES
   220  LET TAX = WAGES * .024
   230  LET NET.PAY = WAGES - TAX
   290  PRINT "Net pay is:"; NET.PAY
   900  DATA  765
```

Program 2-4. Introduce READ and DATA.

This program will produce exactly the same result as our first wages program. The action of line 200—READ WAGES—is to search the program for a DATA statement. Upon finding it READ assigns the value found there to the variable named in the READ statement. Thus the statement pair

```
200  READ WAGES
900  DATA  765
```

does exactly the same job as the statement

```
200  LET WAGES = 765
```

Furthermore, as we will see shortly, the READ-and-DATA combination is used to supply many values for variables during the execution of a program. In the meantime we will look at a third method for assigning values to variables.

So far we have the LET statement and the READ-and-DATA combination. Both techniques require that all values be stored as part of the program and be known before the program is executed. The third method allows us to enter values at the keyboard while the program is actually running.

....Entering Values from the Keyboard (INPUT)

We often find that a program solves a problem based on just a few items of information, such as wages. In addition we would like to create programs that nonprogrammers can run with ease. Sometimes when we run a program we don't even know what numbers to enter until we see some results from another part of the program. In situations like this we may use the INPUT statement of BASIC.

```
INPUT WAGES
```

is another way to assign a value to a variable. The INPUT statement provides the program operator with an opportunity to type values at the keyboard. Following a carriage RETURN the typed value is assigned to the variable named in the INPUT statement, in this case WAGES. This is much better than having to replace a whole program statement. With this capability people who are not programmers can feel confident using our programs to solve their own problems. It is a good idea to display a label to describe the value that is to be entered. See Program 2-5.

```
-->200  PRINT "Enter wages";
  210  INPUT WAGES
  220  LET TAX = WAGES * .024
  230  LET NET.PAY = WAGES - TAX
  290  PRINT "Net pay is:"; NET.PAY
```

Program 2-5. Demonstrate the INPUT statement.

Notice that our program uses three different BASIC statements: INPUT, LET, and PRINT. Programming is the process of putting together those statement types required to solve the problem at hand. Now the program may be used to solve the net-pay problem for many values of wages without changing the program itself. If is this kind of capability that makes the computer such a useful machine. When this program RUNs it will display the message of line 200 followed by a question mark. This is the signal for us to enter our number. See Figure 2-4.

```
Enter wages? 635
Net pay is: 619.76
```

Figure 2-4. Execution of Program 2-5.

In Figure 2-4 we typed the value 635 and the program displayed everything else. Notice that we got the question mark on the same line as our message by using a semicolon in the PRINT statement.

If you should happen to press the RETURN key without entering a value, BASIC will take the value to be zero. In the event that you happen to lean on the repeat (REPT) key and enter 40 digits BASIC will cough. Two messages will be displayed, followed by another question mark.

```
Overflow
?Redo from start
?
```

That means just what it says: reenter your value; the number you entered is too large.

Now we have three methods for providing values for a program to work on: LET, READ and DATA, and INPUT. LET assigns a value according to the expression following an equals sign. READ and DATA may be combined to supply values right in the program itself for use during execution. INPUT assigns a value entered from a keyboard.

....PRINT USING and LPRINT USING

Suppose in Program 2-5 we respond with 777, then what? Let's try it. See Figure 2-5.

```
Enter wages? 777
Net pay is: 758.352
```

Figure 2-5. Try a different value for wages in Program 2-5.

It is difficult to spend .352 dollars. We would like to have our results in this calculation rounded off to the nearest cent. That is easy to do with PRINT USING. PRINT USING allows us to lay out the form we would like to see for the display. We may use number signs (#) to describe how we want the results to look. To allow for three digits to the left of the decimal and two to the right, we use the following statement:

```
290  PRINT USING "###.##"; NET.PAY
```

When our program executes line 290, BASIC will use only the spaces occupied by the number signs for digits. Further, we may include our descriptive label in the quotes. Now, line 290 looks like this:

```
290  PRINT USING "Net pay is: ###.##"; NET.PAY
```

Including this new line 290 in Program 2-5 produces the display of Figure 2-6.

```
Enter wages? 777
Net pay is: 758.35
```

Figure 2-6. Program 2-5 with PRINT USING.

The value in NET.PAY is still 758.352, but the display is rounded off to two decimal places. Eventually, we will see how to round off values for saving in variables.

There is another situation in which it would be nice to employ PRINT USING. Suppose we enter 800 when we run Program 2-5. BASIC will display the result as 780.8. It would be nice to show a zero in the cents column. PRINT USING is just the ticket for this. Running Program 2-5 with the latest version of line 290 will display 780.80. If we enter a value that calls for more digits in the display than the number signs allow to the left of the decimal, the BASIC displays a percent sign (%) to the left of the result. So it is a good idea to allow plenty of space.

Since Program 2-5 talks about money, let's get PRINT USING to include a dollar sign ($) in the display. See Figure 2-7.

```
290  PRINT USING "Net pay is: $###.##"; NET.PAY
RUN
Enter wages? 777
Net pay is: $758.35
```

Figure 2-7. A dollar sign in PRINT USING.

We may include up to two dollar signs there. Only one of them will be displayed. The other one acts just like a number sign—it holds a space for a digit if needed. If only one dollar sign is used, then it will be displayed in the column where it appears in the PRINT USING statement. For two dollar signs, BASIC places a dollar sign right up against the figure in the display.

There are some additional features of PRINT USING that we will look at as it seems appropriate.

....Multiple INPUT and Multiple READ

The INPUT statement is powerful enough just as we have seen it. In addition we may easily enter several values using a single INPUT statement. We simply list all of the variables we wish to assign following the keyword INPUT, separating them with commas. While we may list many variables in a single INPUT statement, it is a good idea to limit the number to three or four at the most. It is difficult to type a very long list of numbers on one line without getting lost somewhere. Two to three is ideal. Incidentally, we may use EDIT Mode when responding to an INPUT request. See Appendix B. CTRL-A lets us correct our typing errors before pressing the RETURN key. CTRL-A only works with Apples that don't have an 80-column card that traps it for upper/lowercase conversion.

When the program runs, the operator must type the values separated with commas. So, to enter the four egg prices from our earlier program, we may use

```
120  INPUT PE, PL, PM, PS
```

where PE stores Price Extra large. The READ statement may be used in the same way.

```
100  READ WE, WL, WM, WS
```

Here we store Weight Extra large in WE. It makes sense to use READ and DATA for the egg weights per dozen since they will never change. It makes sense to use INPUT to assign values for egg prices because they often change. Next we supply some sensible messages and display the cents per ounce as before, and the program is done. This time we have a program that may be used by anyone. See Program 2-6.

```
100  READ WE, WL, WM, WS
110  PRINT "Enter prices in cents"
115  PRINT "Extra large, Large, Medium, Small"
120  INPUT PE, PL, PM, PS
190  PRINT
200  PRINT "Extra large"; PE/WE
210  PRINT "      Large"; PL/WL
220  PRINT "     Medium"; PM/WM
230  PRINT "      Small"; PS/WS
900  DATA  27, 24, 21, 18
```

Program 2-6. Making the eggs program more flexible.

Notice that we have blended the use of INPUT and READ nicely in the same program. READ is appropriate for values that seldom change; INPUT is used for values that usually change.

```
Enter prices in cents
Extra large, Large, Medium, Small
? 100,95,87,72

Extra large 3.7037
      Large 3.95833
     Medium 4.14286
      Small 4
```

Figure 2-8. Execution of Program 2-6.

By using the multiple-value capability of READ and INPUT we have gotten the equivalent of eight LET statements into one READ, one INPUT, and one DATA. The number of values possible here is limited only by the line-length limit (255 characters) and the readability of the program.

....RESTORE

Occasionally we would like to READ DATA more than once. Normally, if the program runs out of data BASIC delivers the

```
Out of DATA in 290
```

error message and execution terminates. The 290 names the line in our program where the READ statement appears. We can change that with the

```
RESTORE
```

statement. All DATA is restored to the program and the next item read by a READ statement will be the very first item in the first DATA statement. Further, if we have a program with several blocks of DATA, we may use RESTORE with a line number to cause the program to begin reading DATA at the line number of our choice.

```
RESTORE 210
```

will cause the next item of data to be selected from the beginning of the DATA statement at line 210. The line number in the RESTORE statement must be a real line number; however, it need not be a DATA statement. So, be careful. In fact, RESTORE is a little-used statement, but from time to time it solves an interesting problem.

....SUMMARY

We have seen the use of numeric variables. Numeric variables store numeric values within a program. Values are stored in numeric variables using LET, READ, and INPUT. The LET statement assigns the value on the right of an equals sign to the variable named on the left. READ copies values from DATA statements to variables. INPUT looks to the keyboard for its source of information.

Both READ and INPUT may be used for several variables by separating them with commas. We may reREAD DATA by using RESTORE or RESTORE 900 to restore all data or just that beginning with line 900.

We can display results rounded off and with a dollar sign. We just put the pattern we want in quotes in a PRINT USING statement.

Problems for Section 2-1

1. What will the following program display?

```
100  LET WAGES = 432
110  LET TAXES = WAGES * .022
130  PRINT WAGES
```

2. Write a program to request three numbers from the keyboard and calculate the average.
3. Write a program to READ three numbers from DATA and calculate the average.
4. Program the computer to request an interest rate in percent and a dollar amount. Have your program display the interest and the amount for simple interest for one year.
5. READ three digits into three variables. Then display all possible arrangements in six PRINT statements. The first PRINT will be

```
140  PRINT A; B; C
```

2-2...Additional Arithmetic Operators

We have become used to working with the conventional arithmetic operations of addition, subtraction, multiplication, and division. Three more operations are available to us. We may raise a value to a power using an exponent. We may command BASIC to perform modular arithmetic and integer division. These operations may be programmed with extra statements, but it is very nice to have them directly accessible.

....Order of Operations

BASIC does addition, subtraction, multiplication, and division exactly the way we would do them on paper. Multiplication and division are done first, followed by addition and subtraction. We may also use parentheses to change that order just as we would in mathematical expressions. Thus, if we need to divide 7 by the sum of 6 and 9, we might type the following:

```
PRINT 7 / (6 + 9)
```

We will quickly get used to writing all these things on a single line. It's just like using an electronic calculator.

....Raising to a Power

We can easily square a number by multiplying it by itself. For higher powers this may not be the best way. Since we cannot write X cubed by writing a superscript, BASIC uses the "^" symbol to indicate "to the power". This symbol is found at "shift-N" on the Apple II and Apple II Plus and at "shift-6" on the Apple IIe. We find the exponentiation symbol in a variety of locations on different terminals used as an external terminal. So, we write X cubed as

```
X^3
```

Raising to the power is carried out in BASIC before addition, subtraction, multiplication, and division just as we ordinarily do it. An expression such as

$$\frac{x^2 + y^2}{x^2 - y^2}$$

is written in BASIC as

```
(X^2 + Y^2) / (X^2 - Y^2)
```

....MODular Arithmetic

Many calculations are cyclic in nature. One common example is the reckoning of time of day. We keep track of time in 12-hour segments. Some institutions use a 24-hour clock. This is a modular process. When we add some number of hours to a given time, we determine the resulting time using modular arithmetic. The days of the week rotate in a modular fashion.

For days of the week we think of a seven-day rotation. Using modular arithmetic we would label the days from zero through six inclusive. Thus if we choose to designate Sunday as day zero, then Thursday becomes day four and Saturday becomes day six. In this situation we say the modulus is seven. Using the MOD operation of BASIC it becomes a simple matter to determine the day of the week that is 17 days from a Tuesday. We simply code a statement such as

```
PRINT (17 + 2) MOD 7
```

and the computer will promptly report a five, which corresponds to a Friday. As with other operations we should be aware of the order in which the computer will do things. Note that we surrounded an expression with parentheses. This is because the MOD operator has a higher priority than addition. Without the parentheses in that statement the computer would display 19 because it would take 2 MOD 7 first and then add 17. The priority of the MOD operator follows multiplication and division and precedes addition and subtraction.

The MOD operator expects values in the range −32768 to 32767 or we will get an "Overflow" error message.

....Integer Division

Integer division simply ignores any remainder after division. While 18/7 is 2.57143, the result of integer division is 2. The symbol for this new operation is the backslash (\). This symbol is produced on an Apple II and Apple II Plus with CTRL-B. It is found just above the RETURN key on the Apple IIe.

```
PRINT 18 \ 7
```

produces the desired result. The −32768 to 32767 limit applies here, too.

Table 2-1 shows the order in which BASIC carries out the various arithmetic operations. As we have already seen, we may use parentheses to alter that order in any expression in our program.

ARITHMETIC OPERATIONS

Symbol	Name	Example
^	Exponentiation	X ^ 3
*,/	Multiplication and Division	X * Y X / Y
\	Integer division	A \ B
MOD	Modular Arithmetic	A MOD B
+, −	Addition and Subtraction	X + Y, X − Y

Table 2-1. Order of operations in BASIC.

Problems for Section 2-2 .....

1. Write a program to print a value for

$$\frac{\frac{1}{2}+\frac{1}{3}}{\frac{1}{4}-\frac{3}{5}}$$

Do this by assigning values as follows: A = 1, B = 2, C = 3, D = 4, and E = 5.

2. Write a program to print a value for

$$\frac{\frac{2}{3}+\frac{3}{4}}{\frac{5}{6}+\frac{2}{3}}$$

Assign variables as follows: A = 2, B = 3, C = 4, D = 5, and E = 6.

3. Write a program to calculate

$$\frac{(17.45 - 6.92)^4}{6.98^3 - 96.2^2}$$

4. Write a program to request two numbers. Have the program print the first number MOD the second. Experiment with a variety of values.
5. Write a program to request two numbers. Print the result of integer division of the first number divided by the second one. Experiment with a range of values.

2-3...More Messages

We have been displaying messages by enclosing them in quotes in PRINT statements. Sometimes the message depends on the program results. For example, we might be looking for the day of the week with the highest temperature or the lowest sales volume. Or we might want to do something as simple as programming the computer to display someone's name to attract attention. BASIC has many features for handling nonnumeric values with ease.

....Word Pigeonholes (String Variables)

We may assign a message to a variable. Such a variable is different from a numeric variable, so we need to use a special kind of variable name. Any variable name that ends with a $ (dollar sign) may be used for this purpose. Variables of this type are usually called "string variables" because they may store a string of characters. Let's see an example. Look at Program 2-7.

```
100  LET MYNAME$ = "Jim"
120  PRINT MYNAME$; " is nice."
```

Program 2-7. Demonstrate string variable.

This little program simply assigns a string value to the string variable MYNAME$ in line 100 and then displays the contents with a little message in line 120. That's all.

```
Jim is nice.
```

Figure 2-9. Execution of Program 2-7.

Suppose we try using NAME$ as a variable name.

```
100  LET NAME$ = "Jim"
```

Execution of this statement will bring forth an unexpected result. BASIC uses NAME as a keyword. NAME is used to change the name of a disk file. (See Appendix C.) So trying to use NAME as a variable is an error. RUNning a program with this line will produce the following display:

```
RUN
Syntax error in 100
Ok
100 ■ <-- cursor here
```

and we swing into EDIT Mode. (See Appendix B.) Since we won't know all keywords in advance, this will occasionally happen to us. We solved this problem in Program 2-7 by using MYNAME$. We could just add a period to the end of any keyword to obtain a legal variable name. So we might also use NAME.$ in this case.

We may work with string variables in many of the ways in which we work with numeric variables. For instance, any of the following statements may appear in a program:

```
100  LET A$ = "First"
100  READ A$
100  INPUT A$
100  PRINT A$
```

String variables may store from 0 to 255 characters at any time. In order to READ A$ we must provide a corresponding DATA statement. If we want to include a comma in the string, then we must enclose the string in quotation marks. Without the quotation marks, any comma is interpreted as the end of the current DATA item. Since string variables may be used with INPUT and READ as well, we could easily change line 100 of Program 2-7 and get the computer to say something nice to our friends.

Program 2-8 is a little demonstration of READing more than one data string.

```
-->100  READ L$, F$
   110  PRINT F$; " "; L$
   900  DATA  Lincoln, Abraham
```

Program 2-8. Demonstrate READing string values.

Line 100 reads the first string into L$ and the next string into F$.

```
Abraham Lincoln
```

Figure 2-10. Execution of Program 2-8.

Suppose we really want to store "Lincoln, Abraham" in a string variable. We simply use quotes as mentioned earlier. It looks like Program 2-9.

```
100  READ NAME.$
110  PRINT NAME.$
900  DATA "Lincoln, Abraham"
```

Program 2-9. Demonstrate READing a comma into a string variable.

....Adding Strings (Concatenation)

Sometimes we want to build up one string from other strings. We can attach strings with the plus sign (+). Plus will not mean numeric addition but "putting together" or concatenation. We might want to use a person's name in a variety of ways in a program. We might want to use the first name sometimes and the full name at others. Consider Program 2-10.

```
  100  READ F$, L$
-->110  LET FULL$ = F$ + L$
  130  PRINT "First name "; F$
  140  PRINT " Last name "; L$
  150  PRINT " Full name "; FULL$
  900  DATA George, Washington
```

Program 2-10. Demonstrate string concatenation.

```
First name George
 Last name Washington
 Full name GeorgeWashington
```

Figure 2-11. Execution of Program 2-10.

Oops! We must change line 110 in Program 2-10 to read:

```
110  LET FULL$ = F$ + " " + L$
```

Now we will get a space between George's first and last name. We may use the plus sign to join strings as long as the total number of characters does not exceed 255. If this limit is exceeded we will see the

```
String too long in 110
```

error message.

Don't get carried away with string operations. If we try to subtract strings we will evoke another error message from BASIC. The line

```
100  C$ = B$ - D$
```

will produce the following:

```
Type mismatch in 100
```

It just isn't defined. Minus is for numbers, not strings. BASIC allows two data types: strings and numerics. We will get the same treatment for a statement such as:

```
100  LET A$ = 65.45
```

We simply can't indiscriminately mix strings and numerics.

We will be adding more string capabilities to our repertoire as time goes on, but direct subtraction and direct arithmetic will not be among them. To get the characters 65.45 stored in string variable A$, just enclose them in quotes.

```
100  LET A$ = "65.45"
```

....SUMMARY

String variables may be used to store nonnumeric data. We may use strings with LET, INPUT, READ and DATA, and PRINT statements. Two strings may be joined using a plus sign. To include a comma in a string it is necessary to surround the string data with quotes.

Problems for Section 2-3 .....

1. Rewrite Program 2-6 to READ the egg-size names from DATA along with the weight per dozen.
2. Write a program to READ the days of the week into seven variables and display them.
3. Write a program to request a person's name from the keyboard. Have your program respond with "Hello there 'your name'".
4. Write a little program to request a single string using INPUT and display the string variable with PRINT. Experiment. Enter a string with and without a comma. Verify that you can get a comma into the string by using quotes. Work with this until you are comfortable with string INPUT.
5. Write a program to request a person's name in two strings, first name first. Display the name in the form last name, comma, first name. For example, for "John, Jones" entered at the keyboard, display as "Jones, John".

SIDELIGHT 2

A Word about Precision

....Single Precision

The numbers we have been working with are called single-precision numbers. They are displayed with up to six significant digits as required. This enables us to perform calculations for a wide range of numeric values. We get single-precision numbers automatically in BASIC. Later we will discuss how to get up to 16-digit precision and how to limit calcula-

tions to the range of integers from −32768 to 32767. The ranges are related to the amount of computer memory required to store numeric values. In addition, more execution time is required to operate on values of greater precision. For now, let's explore the world of single precision.

Even with single precision, BASIC distinguishes between whole-number values within the integer range and those outside. Let's type in a simple program and list it. See Figure 2-12.

```
NEW
100  LET X = 3241
110  LET Y = 65000
120  LET Z = 65000.9
LIST
 100  LET X = 3241
-->110  LET Y = 65000!
-->120  LET Z = 65000.9
```

Figure 2-12. Demonstrate numeric values in program LISTing.

See what BASIC did to line 110? An exclamation point is appended to the 65000. This serves as a reminder to us that, while that number might look like a true integer, it is stored as a single-precision value. Look at line 120. There is no exclamation point there, but we know that 65000.9 is not an integer. Suppose we LET X = 9999999. BASIC does something quite different. We are plunged into the world of E-format.

....E-format

Let's construct a program that will produce a result too big or too small for six digits. When this happens BASIC resorts to a special notation that displays six-digit precision and a power of ten. It is called E-format and is just like scientific notation. Consider Program 2-11.

```
100  N1 = 93.326
110  N2 = 24398.9
200  PRINT "The numbers are :"; N1; "and"; N2
220  PRINT "     Add :"; N1 + N2
230  PRINT "Subtract :"; N1 - N2
240  PRINT "Multiply :"; N1 * N2
250  PRINT "  Divide :"; N1 / N2
```

Program 2-11. Demonstrate E-format.

```
The numbers are : 93.326 and 24398.9
     Add : 24492.2
Subtract :-24305.6
Multiply : 2.27705E+06
  Divide : 3.82501E-03
```

Figure 2-13. Execution of Program 2-11.

Look at the results for "Multiply" and "Divide". There we see 2.27705E +06 and 3.82501E −03. The expression 2.27705E +06 means 2.27705 times ten to the sixth power. Let's get PRINT USING to tell us what those numbers really look like. We can do it in immediate mode right now.

```
PRINT USING "########  ###.#########"; N1 * N2, N1 / N2
 2277050    0.00382501
```

The 0 on the end in 2277050 is not exact. The 5 is the result of rounding the calculation off to six significant digits.

We can even get PRINT USING to put in commas after every three digits if we want. This is done by placing a comma in the PRINT USING pattern just to the right of the position of the units digit. If the pattern has a decimal point, the comma goes next to it on the left.

```
PRINT USING "########,"; N1 * N2
2,277,050
```

....The Biggest Number?

What is the biggest number we can get? We can easily ask the computer. Just type

```
PRINT 1 / 0
```

Normally we don't divide by zero, but let's do it just this once to learn about the limits of BASIC. We get

```
Division by zero
 1.70141E+38
```

The error message reminds us that division by zero is frowned upon. But the rest of the program will be executed just the same. The displayed value is simply the largest value BASIC can produce for us. Likewise, for −1 / 0 we get −1.70141E +38 as the smallest value available. In either we get the "Division by zero" warning.

Chapter 3

Writing a Program

In this chapter we will be adding numerous new BASIC features. We will see how to use BASIC to repeat procedures, make decisions, and perform special calculations. Here we will begin to develop the habit of describing our program as we go along. As our programs grow, it will become clearer that we should do some planning before we begin entering code into the computer. This should nicely round out our programming ability and give us a good framework on which to hang many interesting and powerful tools.

3-1...Do It Again

Many, if not most, computer applications involve repetitious operations. Often that requires counting of some kind. Counting is one of the earliest mathematically oriented skills that we learn in life. If we can teach the computer to count, we will be well on the way to managing repetitious calculations of all kinds.

Think about counting. We set up at 1. Then we get to 2 by adding 1. Then we get to 3 by adding 1. We always get to the next number in line by adding 1.

....Our First Counting Program

We easily set up at 1 with a simple assignment statement.

```
100  COUNT = 1
```

Then we get to the next number by adding 1.

```
140  TEMP  = COUNT + 1
150  COUNT = TEMP
```

If we were limited to the statements offered so far, we would use another pair of statements such as 140 and 150, followed by two more statements to add 1, and so on. That would lead to very long programs. What we need is an instruction that causes statements 140 and 150 to be executed again and again. We simply need to divert program execution from the usual increasing-line-number sequence to execute line 140 after each time it executes line 150. At line 160 we need a statement that says "go to line 140".

....GOTO

The statement

```
160  GOTO 140
```

will always cause line 140 to be executed next regardless of what statements might follow with higher line numbers in the program. GOTO is sometimes called an unconditional transfer statement. Thus program execution will loop back to continuously repeat the statements just processed. Now let's collect our program in one place. See Program 3-1.

```
100  COUNT = 1
140  TEMP  = COUNT + 1
150  COUNT = TEMP
160  GOTO 140
```

Program 3-1. Our first counting program.

You should be a little suspicious of Program 3-1. What makes it stop? Certainly nothing inherent in the program conveys the idea that it will end. And it won't—until you press CTRL-C, press the RESET key, press CTRL-RESET, or pull the plug. CTRL-C is an emergency procedure that will halt execution of any program regardless of the instructions in the program. Let's not create an emergency here. This is your classic endless loop. We're not done yet! Not only that, but the program never tells us where it is. It counts to itself. Let's make it count "out loud".

The "out loud" part is easy. All we have to do is include a PRINT statement in the right place and make sure that it is executed for all values of COUNT. The result is Program 3-2.

```
-->100  COUNT = 1
   130  PRINT COUNT
   140  TEMP  = COUNT + 1
   150  COUNT = TEMP
-->160  GOTO 130
```

Program 3-2. Counting "out loud" this time.

We inserted line 130 to display the counting value and changed line 160 to read GOTO 130. Now we are ready to tell our counting program where to stop. Let's have our program count from 1 to 7.

What we really need is the ability to execute the display in line 130 only so long as the value of COUNT is within range. In our case, if COUNT is greater than 7 we want to stop the repetition. For this we will use the IF statement of BASIC.

....IF . . . THEN

We can have our program display a little message at line 190 when the counting is completed. Therefore we want to divert execution to line 190 when the value of COUNT exceeds 7. We may insert an IF statement just before line 130 to do this.

```
120  IF COUNT > 7 THEN 190
```

does the job. Here the symbol > is used to represent "greater than". We have six options in an IF statement:

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

These symbols are called relational operators. Any BASIC expression may appear on either side of a relational operator.

Line 120 will transfer the flow of execution of the program to line 190 as soon as the value of COUNT passes 7. Thus the IF statement used in this way is sometimes called a conditional transfer. Now we must change line 160 to GOTO 120 in order to execute the IF test for each value of COUNT.

The resulting program simply counts "out loud" from 1 to 7. See Program 3-3.

```
   100  COUNT = 1
   120  IF COUNT > 7 THEN 190
   130   PRINT COUNT
-->140   TEMP  = COUNT + 1
-->150   COUNT = TEMP
   160   GOTO 120
   190  PRINT "Done"
```

Program 3-3. Counting from 1 to 7.

It is important to note that the value of COUNT actually overshoots by 1. So, when this program terminates, the value of COUNT will be 8. In this program we have also begun a practice that is intended to make programs

easier to read. We have indented the statements following the IF statement that work together as a group.

Many articles have been written about the evils of the GOTO statement in BASIC (and some other programming languages). Indeed the beginning programmer is likely to overuse it. Programs that have a lot of GOTOs are very difficult to read. We like to read programs and segments of programs pretty much from top to bottom. Too many GOTOs interrupt this natural way to read. Thus, after three or four detours to follow GOTOs we begin to become confused. We can't remember whether we have read the whole program and we lose track even of what the program is supposed to do. As we plan programs we will use it sparingly. There are some programming situations, however, where GOTO is the simplest solution to the problem. We will use it for those situations.

Lines 140 and 150 of this program deserve some discussion. All we want to do is increase the value of the variable COUNT by 1. We did this by using TEMP as an intermediate variable. We really want the variable COUNT to take on the value COUNT + 1. In an assignment statement in BASIC the equals sign implies exactly that.

```
140  LET COUNT = COUNT + 1
```

is perfectly legal and proper in a computer program. In this situation the equals sign does not declare an equivalence but describes an action for the computer to carry out. We may think of the equals sign as a little arrow pointing to the left when it is used in this way. The computer must calculate the value of COUNT + 1 defined on the right and store it in the variable COUNT named on the left. By using this simplified method of adding 1 to COUNT we have shortened our program by one line. Whenever we can shorten a program without making it any harder to read it is a good idea. In this case it seems like a good idea. See Program 3-4.

```
100  COUNT = 1
120  IF COUNT > 7 THEN 190
130   PRINT COUNT
-->140   COUNT = COUNT + 1
160   GOTO 120
190  PRINT "Done"
```

Program 3-4. Counting from 1 to 7 with COUNT = COUNT + 1.

This program will produce exactly the same results as the first one.

Our counting program has four distinct components. These four ingredients play a part in all program loops.

1. We initialize the counting variable.
2. The value of the counter is tested to determine whether to recycle or exit the repetition.

3. Some action is programmed. In our example, we display the current value of the counter.
4. Increment the counter, and loop to step 2.

Later on we will be taking advantage of an automatic "loop maker" in BASIC. We have designed our first loop program to perform in exactly the same way as the automatic feature of BASIC.

```
 1
 2
 3
 4
 5
 6
 7
Done
```

Figure 3-1. Execution of Program 3-4.

With our loop maker it is easy to make small changes to alter how the program will count. We can change line 100 to begin the count at any number we like. We can change line 110 to end the count anywhere. And we can alter line 140 to count by twos or sixes or nines or whatever. We could even have our program count backwards by subtracting in line 140 (note that, by using the technique presented here, if the final value precedes the initial value then nothing happens).

Usually we have some higher purpose in mind for counting than merely displaying the value of the counter. We want to scan the days of the week, or the months of the year, or the years of the life of a mortgage, or the names on a customer list. Or we might just want to flip a coin so many times. Maybe we want to roll so many dice or just display "I like BASIC" nine times.

If we bounce a hard steel ball on a hard surface it will bounce many times. How high and how many times depend on the elasticity of the material. Suppose we have a ball that recovers nine-tenths of its height on each bounce. If we drop such a ball from 10 meters it will bounce to 9 meters on the first bounce and 8.1 meters on the second. It is not hard to develop a formula to calculate the height after any number of bounces, but it is also not hard to write a program to simulate the bouncing of the ball. It will then be very easy to modify our program to calculate additional values for us. Let's bounce the ball five times. All we need for that is to change our counting program to stop at 5 instead of 7. We need to include a statement that calculates the new height for every bounce. We need a PRINT statement to display the number of bounces and the height. It would be nice to include a PRINT statement to label the two columns of figures.

```
-->80   PRINT "Bounce Height"
   90   LET HEIGHT = 10
  100   LET COUNT  = 1
  110   IF COUNT > 5 THEN 190
  120    LET HEIGHT = HEIGHT * .9
-->130   PRINT COUNT; "     "; HEIGHT
  140    LET COUNT = COUNT + 1
  160    GOTO 110
  190   PRINT "Done"
```

Program 3-5. Bouncing a steel ball.

```
Bounce Height
 1      9
 2      8.1
 3      7.29
 4      6.561
 5      5.9049
Done
```

Figure 3-2. Execution of Program 3-5.

....Comma Spacing

In Program 3-5 we took some pains to line up the columns of figures with the column labels displayed in line 80. That is because we are still using the semicolon to separate items in PRINT statements. If we use a comma instead, BASIC automatically forms columns 14 digits wide. All we have to do is make small changes in lines 80 and 130. In line 80 we separate the two labels with a comma. In line 130 we remove the extra spaces and replace the semicolon with a comma. Let's see how that looks. See Program 3-6.

```
-->80   PRINT "Bounce", "Height"
   90   LET HEIGHT = 10
  100   LET COUNT  = 1
  110   IF COUNT > 5 THEN 190
  120    LET HEIGHT = HEIGHT * .9
-->130   PRINT COUNT, HEIGHT
  140    LET COUNT = COUNT + 1
  160    GOTO 110
  190   PRINT "Done"
```

Program 3-6. Program 3-5 with comma spacing.

```
Bounce          Height
 1               9
 2               8.1
 3               7.29
 4               6.561
 5               5.9049
Done
```

Figure 3-3. Execution of Program 3-6.

For many purposes it is quite satisfactory to use comma spacing. This feature allows us to quickly produce a nicely arranged display without the bother of having to count spaces and go to the extra trouble to line things up. BASIC does it for us. Beyond this we could employ PRINT USING.

....SUMMARY

In this section we have learned to count—or, rather, we have learned how to make the computer count. To do this we have mastered the GOTO and IF statements. GOTO is used to unconditionally divert the order in which the statements of the program are executed. The IF statement is used to conditionally determine which statement will be executed next. We have seen that the LET statement in BASIC may name the same variable on both sides of the equals sign. A comma may be used to separate items in a PRINT statement. This sets up the display screen into columns that are 14 characters wide.

Problems for Section 3-1 .....

1. Write a program to display "I like BASIC." eight times.
2. Modify Program 3-4 to count from 1 to 19.
3. Modify Program 3-4 to count from 1 to 7 by twos.
4. Modify Program 3-4 to count from 1 to 100 and calculate the sum of the numbers in the sequence. You might not want to display all 100 values of the counting variable.
5. Modify Program 3-4 to count from 2 to 42 by twos.
6. Modify Program 3-4 to count backwards from 10 to −10.

3-2...Do It Again (When We Don't Know How Many)

It is easy to tell the computer to do something a certain number of times when we know how many times we want, but that is not always the case. In fact we might want the program to perform a certain calculation until some point is reached and tell us how many times it took. We might want to know how many bounces the steel ball makes before it bounces less than half the original height. We might want to have a program keep asking for values from the keyboard until a special value is entered as an instruction to stop requesting values.

How about a program to calculate test averages? One person might enter three test scores while the next might wish to enter five test scores. For this we need to know the total of the scores entered and the number of scores. How will the program "know" when the person has entered all the scores? Let's choose a special value to signal that. How

about −1? If the operator enters −1, the program should proceed to the average calculating statements. If the operator enters any positive score then the program should add that score to the current sum and ask for a new test score.

....A Little Planning

This program is shaping up to be a little more involved than those we've done up to this point. So this is an opportunity to work on the process of program development. There are really three things this program ought to do.

1. Tell the user what the program does.
2. Request test scores from the keyboard.
3. Calculate and display the average.

Once the programmer understands the problem and what steps are involved in the solution it is a good idea to define these steps right in the program itself. We could put the three steps listed above in PRINT statements, but that might not be appropriate for the running program. It is usual to have messages in the program for programmers alone. In BASIC this is done with the REM statement.

....REMark

Whatever follows on the line after REM in a program statement will be ignored by BASIC. Thus what we type functions as a "remark" to anyone reading our program, rather than an instruction to the computer to perform any action. This is a part of what we call program documentation.

Good REM statements are brief and succinct. It takes a little skill to develop good remarks in a program. REMarks like "Increment J1" and "Subtract NUMBER from OLDNUMBER" are quite uninformative. We could better see those actions from the program statements themselves. Such REMarks actually make a program harder to read. On the other hand, REMarks like "Initialize accumulated mileage" or "Terminate on negative INPUT" describe the intended action and are helpful to anyone reading our program.

Programs that are crystal clear to us right now will be foggy and mysterious a few weeks from now. So we ought to include REMs right away. A lot of people write their programs and then go back to insert REMs. A few people write the REMs first and then write the program. Let's do it that way.

Taking the three segments above, let's write some REMs. We will select line numbers for the REMs so that they serve as labels for the program segments that actually perform the computer task. So we need to think about how to lay out the line numbers in the final program. This is a process that will come with practice.

1. Tell the user what the program does.

That is really the instructions. We do that with a few PRINT statements. Let's begin at line 100 and place the REM at line 98 thus:

```
98   REM ** Instructions
```

The stars tend to set this statement off a little. If our remark runs over to several lines we will omit the stars on the rest.

2. Request test scores from the keyboard.

This is where we request data, add the scores to a summing variable, and recycle to the INPUT statement using GOTO. Let's begin this segment at line 200 and place the REM at 198.

```
198  REM ** Request test scores
```

That ought to do it.

3. Calculate and display the average.

And finally we need to report the results to the program user. Let's leave a little extra room and place this segment at line 400.

```
398  REM ** Calculate average
```

Using this method of program planning is very beneficial. We have separated the programming job into little tasks that are relatively easy to do one at a time. We have created some program documentation in advance. All too often programmers, being human, work feverishly until the program performs the desired task. Then it is very difficult to discipline oneself to go back and produce good documentation. After all, the program is done, isn't it?

Doing it our way, when the program is done, at least part of the documentation is done too. At this point we have nice labels for the three parts of the program.

```
98   REM ** Instructions
198  REM ** Request test scores
398  REM ** Calculate average
```

Program 3-7a. REMs for average calculation program.

We can use these REMs as our program outline during the process of writing the BASIC program statements. The process of writing the program statements is often called "coding" because we are converting our ideas into "code" that the computer can work with. The statements are called "code."

Now we can easily write program statements for each of the three parts without having to think about any of the other parts. This segmenting of the program helps free our mind for clear thinking.

Now for the instructions:

```
98   REM ** Instructions
100  PRINT "Test score averaging"
110  PRINT
120  PRINT "Enter test scores - one at a time"
130  PRINT "Enter -1 after last score"
140  PRINT
```

Program 3-7b. Instructions segment.

We have tried to say what the program does and what the user should do. It is important to write short and easy-to-read instructions. It is worth the effort to produce clear, succinct displays for the user. Long instructions that fill the screen are difficult to read. Long, hard-to-read REMs are also not good.

Next, the keyboard entry segment:

```
198  REM ** Request test scores
200  NUMBER = 0
210  SUM    = 0
220  PRINT "Score"; NUMBER + 1;
230   INPUT SCORE
240   IF SCORE = -1 THEN 400
250    SUM     = SUM + SCORE
260    NUMBER = NUMBER + 1
290   GOTO 220
```

Program 3-7c. Keyboard entry segment.

Again we have added some little touches here that may make our program a little easier to read. Where we have a group of statements that will be executed repeatedly we have indented to show the extent of the repeated code. And we have indented code following the IF statement to show that what follows is grouped in another way. You should adopt any of these practices that make your programs easier for you to read.

As you type in your programs, you may have occasion to employ EDIT Mode (see Appendix B) to get it right. Take the time to master EDIT Mode. It really is worth the effort.

And finally, the average calculation and display:

```
398  REM ** Calculate average
400  AVG = SUM / NUMBER
420  PRINT
430  PRINT "Average ="; AVG
```

Program 3-7d. Calculate average segment.

Programmers often refer to little segments of programs as "routines." Every programmer has his or her favorite routine to integrate the left framis or digitize the window turn.

Now we have a nice routine to request test scores from the keyboard and display the average. See Program 3-7.

```
   98   REM ** Instructions
   100  PRINT "Test score averaging"
   110  PRINT
   120  PRINT "Enter test scores - one at a time"
   130  PRINT "Enter -1 after last score"
   140  PRINT
   198  REM ** Request test scores
   200  NUMBER = 0
   210  SUM    = 0
   220  PRINT "Score"; NUMBER + 1;
   230   INPUT SCORE
   240   IF SCORE = -1 THEN 400
   250    SUM    = SUM + SCORE
   260    NUMBER = NUMBER + 1
   290  GOTO 220
   398  REM ** Calculate average
-->400  AVG = SUM / NUMBER
   420  PRINT
   430  PRINT "Average ="; AVG
```

Program 3-7. Calculate average.

All that remains is to RUN it. See Figure 3-4.

```
Test score averaging

Enter test scores - one at a time
Enter -1 after last score

Score 1 ? 100
Score 2 ? 91
Score 3 ? 71
Score 4 ? -1

Average = 87.3333
```

Figure 3-4. Execution of Program 3-7.

We might want to take advantage of PRINT USING here. This would allow us to round off results to suit our purpose.

```
430  PRINT USING "Average = ##.#"; AVG
```

rounds off to the nearest tenth for us. It is easy to do and greatly improves the appearance of our results.

There is one more little wrinkle in Program 3-7 that deserves some more attention. Suppose someone runs this program and then finds that there really aren't any scores to enter. Or suppose someone enters −1 as the first score by accident. When execution arrives at line 400 the value of NUMBER is zero.

```
400  AVG = SUM / NUMBER
```

Line 400 causes the computer to attempt to divide by zero. That's bad news. Let's see what BASIC does with it.

```
Test score averaging

Enter test scores - one at a time
Enter -1 after last score

Score 1 ? -1
Division by zero

Average = 1.70141E+38
```

Figure 3-5. Demonstrate division by zero.

We saw this condition in Sidelight 2. When BASIC tries to divide by zero it just comes up with 1.70141E+38 and proceeds with that value. A laudable goal for computer programmers is never to subject the person who uses our programs to error messages from BASIC. At line 400 we can determine if the value of NUMBER is zero or not. If it is, we want to terminate the RUN. If it is not zero, we want to allow the calculation and display to proceed. So let's move line 400 to line 410 and put in an IF statement at line 400. What will the IF statement say? We can do exactly what we did in Program 3-3. We can put in a line 490 PRINT "Done". That has worked for us before, but let's explore other options. BASIC includes a special statement that causes execution to end. It is the END statement.

....END

Up to this point we have allowed our programs to terminate by simply "running off the end." We may be more explicit with the END instruction. The END statement says "go no further". It is an orderly way for a program to terminate. So we may include the statement

```
490  END
```

and the statement

```
400  IF NUMBER = 0 THEN 490
```

Now our little routine at line 400 reads as follows:

```
398  REM ** Calculate average
400  IF NUMBER = 0 THEN 490
410   AVG = SUM / NUMBER
420   PRINT
430   PRINT "Average ="; AVG
490  END
```

There is another way to do this. It turns out that the IF statement has several forms.

....IF . . . THEN Revisited

We have been using only the feature of the IF statement that transfers control to some line in the program. THEN may also be used to execute any BASIC statement—even another IF. Thus

```
185  IF X = 5 THEN GOTO 212
```

produces the same effect as

```
185  IF X = 5 THEN 212
```

Using this new concept we can simply move line 400 to 410 and code the following line at 400:

```
400  IF NUMBER = 0 THEN END
```

It's that simple.

....STOP

We may use STOP to terminate program execution. The line 245 STOP will cause the following message:

```
Break in 245
```

Ordinarily, we would only use STOP to interrupt execution in an effort to hunt down an error while we are writing a program. STOP should only be used to indicate an extraordinary condition. It is very helpful to have the line number displayed for us here.

There is another aspect of our average-calculating program that we ought to think about. What happens if we enter a negative score other than −1? It is summed right in with the others. We could easily include an IF test to see if a negative score other than −1 is entered. A nice touch would be to display a message that the score entered is out of range and to request a value again. Well-written programs verify response from the keyboard. Values absolutely out of range are refused. The operator is required to enter another value. For our averaging problem, we might exclude all scores above 100 as well. This is left as an exercise.

....SUMMARY

The REM statement has been introduced. REM is used to include messages about the program to humans who will be reading it. REM has no effect upon the execution of a RUNning program. We used REM statements judiciously to help us organize our thoughts prior to actually writing BASIC program statements. The END statement may be used for an orderly program termination. In addition to changing the order in which the computer executes program statements, IF . . . THEN has the ability to execute any statement that follows THEN on the same program line.

Problems for Section 3-2

1. Modify Program 3-6 to determine how many times the ball bounces before it fails to recover half the original height.
2. In Problem 1 calculate the total distance traveled by the steel ball. Remember that it travels down one distance and up another.
3. Modify Program 3-7 so that the user cannot enter any negative score other than −1. Also reject any score above 100.
4. Write a program to convert money into coins. For example: 99 U.S. cents becomes

   Enter cents:? 99
   1 Half dollars
   1 Quarters
   2 Dimes
   4 Pennies

   Place the coin values and names in DATA statements.

3-3...IF . . . THEN . . . ELSE

Let's develop a program to display the passing of the hour. It will just repeat 1 through 12 over and over again. The primary consideration is that whenever HOUR reaches 12 we reset it to 1 and continue. That requires two statements following THEN. In order to do that we need the colon delimiter.

....Multiple Statements on One Line (:)

In BASIC we may separate two or more statements on the same line by using a colon. Thus we may code lines such as

```
300  X = 5 : Z = 18
```

Use this with caution. Don't make program lines too long; they may become difficult to read. It is very nice to be able to place statements that belong together on the same line. Getting back to our digital clock: if the HOUR is less than 12 we simply increase HOUR by 1. That is easy:

```
 110  HOUR = 12
-->200  IF HOUR = 12 THEN HOUR = 1 : GOTO 250
 210   HOUR = HOUR + 1
 250  PRINT HOUR;
 280  GOTO 200
```

We have here the makings of a digital clock. See the colon in line 200. If HOUR = 12, HOUR will be set to 1 and execution will proceed to line 250. Three things are needed to make this quite realistic. First, we should

clear the Apple text screen at the beginning of the program. BASIC includes the HOME statement for just this purpose.

```
100  HOME
```

clears the screen and places the cursor in the upper left corner. Second, each new time should overwrite the previous one. The HTAB statement allows us to place PRINTed display anywhere on the current line.

```
250  HTAB 1
```

places the cursor in the first printing position of the line for us. This is an absolute position. The cursor can be moved forward or backward with HTAB. Third, we should program in at least enough delay so that we can read the display of each hour as it appears on the screen. Of course, to be completely realistic, we would leave each value in place for exactly one hour. We could put in a little counter that does nothing but take up time. With a little experimentation, we could determine the upper limit that would keep the clock display on the screen for the hour. For testing we will let it move much faster.

```
260  X = 1
270  IF X < 200 THEN X = X + 1 : GOTO 270
```

will hold the display for a few seconds. See Program 3-8.

```
  98   REM ** Display digital hours
 100   HOME
 110   HOUR = 12
-->200  IF HOUR = 12 THEN HOUR = 1 : GOTO 240
-->210   HOUR = HOUR + 1
 240   HTAB 1
 250   PRINT HOUR;
 260   X = 1
 270   IF X < 200 THEN X = X + 1 : GOTO 270
 280   GOTO 200
```

Program 3-8. An hourly digital clock.

Enter Program 3-8 into the computer and try it.

Look at lines 200 and 210. BASIC allows us to program two alternative actions right in the IF . . . THEN itself. We can do all that in a single IF . . . THEN . . . ELSE statement. Here is what it looks like:

```
200  IF HOUR = 12 THEN HOUR = 1 ELSE HOUR = HOUR + 1
```

It is that simple. Sometimes it is nice to rearrange statements like this one for easier reading. See if you like the following version:

```
200  IF HOUR = 12 THEN HOUR = 1
                  ELSE HOUR = HOUR + 1
```

This is done with another new feature of BASIC.

....Multiple Lines per Statement (CTRL-J)

CTRL-J has a special meaning in a BASIC program. This character generates a new physical line on the screen without signaling a new program line. We are free to use this to make our programs easier to read. The computer doesn't care. We could replace line 200 in Program 3-8 and erase line 210 to produce exactly the same result. See Program 3-9.

```
   98   REM ** Display digital hours
   100  HOME
   110  HOUR = 12
-->200  IF HOUR = 12 THEN HOUR = 1
                     ELSE HOUR = HOUR + 1
   240  HTAB 1
   250  PRINT HOUR;
   260  X = 1
   270  IF X < 200 THEN X = X + 1 : GOTO 270
   280  GOTO 200
```

Program 3-9. The digital clock with IF . . . THEN . . . ELSE.

....SUMMARY

We may use a colon to separate BASIC statements on a single program line. The IF . . . THEN statement allows us to program two options in the same statement using the ELSE keyword. We may extend a program line onto additional physical lines by using the CTRL-J character. This is a convenience for making programs readable.

We are beginning to accumulate quite a collection of ways to specify how our printed display will look. We may label our results with quoted messages. The semicolon gives us close spacing, while we may set up 14-character columns with a comma. The PRINT USING statement allows us to display nice labels, place our numeric result anywhere, and round off results as needed. HTAB allows us to declare any position on a line for the next PRINT position. This position is independent of where the last item appears.

Problem for Section 3-3.....

1. Extend the digital clock of Program 3-9 to show minutes and seconds.

SIDELIGHT 3

More about INPUT

The INPUT statement has a number of interesting features. Later, we will even use it for accessing data in disk files. We have often used a

PRINT statement to label our INPUT requests. These labels are sometimes called prompts.

....INPUT with Prompt

We may display a message within the INPUT statement itself.

```
100  INPUT "Enter two integers"; A, B
```

will display the message in quotes, output the usual question mark followed by a space, and request two numeric values. This is exactly the behavior we produced with statements such as

```
200  PRINT "Enter two integers";
210  INPUT A, B
```

But with prompted INPUT we have some other options. Suppose we don't want the question mark. It can be suppressed by using a comma at the end of the message in the INPUT statement. We might prefer to distinguish a particular question by using some other symbol.

```
100  INPUT "Enter two integers: ", A, B
```

Now we have the following screen display:

```
Enter two integers: 23,98
```

where 23, 98 was typed at the keyboard. The rest of the display was produced by the INPUT with the prompt statement.

Further, we might like to ask several questions on the same line. We can do this by suppressing the carriage return coming from the keyboard with a semicolon before the message in the INPUT statement.

```
100  INPUT; "Age"; YEARS
110  INPUT; "  Weight"; POUNDS
120  INPUT  "  Sex (M or F)"; GENDER$
```

These three INPUTs will be strung out on one line to produce the following display:

```
Age? 32  Weight? 134  Sex (M or F)? M
```

where 32, 134, and the final M were all entered from the keyboard. Of course, in the situation where we have several INPUTs on the same line, the

```
Redo from start
```

error message applies only to the most recent INPUT statement.

We are still free to display our messages with PRINT statements and use the special features of INPUT by putting a null message in prompted INPUT. In any situation where our prompt might change from one execution of a statement to the next, we might use something such as this:

```
200  PRINT A$; : INPUT "- ", X, Y
```

This way we have a very flexible capability.

....LINE INPUT

We may want to enter a comma in a string INPUT request. We may achieve this by enclosing the entry in quotes. Using quotes may not be the best way, however, especially for a program that may be used by a beginner. This is also a problem if sometimes we need a comma and sometimes not. It may be a nuisance to remember whether or not we need quotes. The LINE INPUT statement is designed to accept an entire line of string input without regard to commas.

```
200  LINE INPUT "Enter a name ", A$
```

This LINE INPUT statement will display our little message and no question mark. Then we have the opportunity to enter whatever is appropriate, commas or not as we choose. If we want a question mark displayed in this situation, we simply include it in the message. Replacing the comma with a semicolon won't do it.

Chapter 4

Loops

In the last chapter we developed the idea of looping in programs. Many computing situations involve repetitious operations. In this chapter we will use the BASIC control structures for loops. The counting process pervades computer programming to such an extent that it makes sense to automate the instructions. The FOR and NEXT statements allow us to set values for beginning and ending limits of a repetition.

4-1...Counting with FOR and NEXT

Let's examine another program to count from 1 to 7.

```
   200  FOR COUNT = 1 TO 7
   220    PRINT COUNT
   240  NEXT COUNT
-->290  PRINT "Done"
```

Program 4-1. Counting with FOR and NEXT.

A FOR statement in BASIC is used to set the beginning and ending values for a selected variable. We used COUNT in this case. It is called the loop variable. All statements between here and a matching NEXT statement will be executed as long as the selected variable is within the range indicated. The repetition ends with NEXT COUNT. NEXT COUNT automatically adds 1 to the value of COUNT until COUNT exceeds 7. At this point the statement following the NEXT statement will be processed. Thus, in our little program, the word "Done" will be displayed. At line 290 the value of COUNT will be 8.

Suppose we code a routine such as

```
200  FOR COUNT = 7 TO 3
210   PRINT COUNT
290  NEXT COUNT
```

What will happen? That depends on which revision of BASIC-80 you are using. The revision number is displayed whenever you invoke BASIC-80 on your computer. If someone else has always done this before you get to the computer, don't despair. All you need to do is RUN this little three-line program to find out. If the program produces no display, then you have revision 5.0 or newer. If the program displays the number 7, then you have revision 4.51 or older. It doesn't matter which you have; both versions perform looping in a satisfactory manner. But it is important to know how your FOR and NEXT loops will behave.

....CTRL-S and CTRL-O

Suppose we program something that flies off the screen before we can get a good look at it. Some loop going from 1 to 10000, for example. We can freeze the display with CTRL-S. Holding down the CTRL key and pressing S will do it. Now we can relax to study the screen. One of the important uses for this is to LIST a long program looking for a particular segment of interest. Once we have found it, then we must use LIST to display the desired range of line numbers. We should take great care that our final programs control the screen in such a way that the user does not have to frantically lunge at the keyboard searching for CTRL-S. Press CTRL-Q to resume output. (Any key will work.)

CTRL-O does something quite different. The display freezes, all right, but program output continues just the same. To pick up the display wherever the computer has gotten in the meantime, simply press CTRL-O again. No other key will do it. But CTRL-C will halt execution.

....STEP

The STEP feature of the FOR statement allows us to specify our own increment. If we want to count by twos we can easily do it with FOR and NEXT. See Program 4-2.

```
100  FOR COUNT = 2 TO 10 STEP 2
110   PRINT COUNT
190  NEXT COUNT
```

Program 4-2. Counting by twos with STEP.

```
2
4
6
8
10
```

Figure 4-1. Execution of Program 4-2.

Of course, the value of COUNT will be 12 following execution of Program 4-2. If we ever want the value of a FOR variable that was the last one actually used in the loop, then we should code a statement such as

```
102  TEMP = COUNT
```

Then we are assured that the variable TEMP has saved the last value of COUNT no matter how execution exits the loop.

With STEP we can easily count backwards. Just use a negative STEP value and be sure that the limits in the FOR statement are correct.

The limits in the FOR statement may be variables. The limit values need not be integers. But when using decimal values the computer may round things off so that an unexpected final value is produced.

FOR and NEXT are widely used in programs. They are good shorthand. Whenever we need to count in a program it is easy to think in terms of FOR and NEXT. We don't have to think about initializing a variable, incrementing it, and testing it to see if we are through. All this is done automatically and painlessly, so our minds can remain uncluttered and free to consider the higher purposes of our program.

In addition, FOR and NEXT are very helpful to us when reading existing programs. When we see FOR we know that a repeated process follows, ending with the matching NEXT statement. Furthermore, the beginning and ending values appear right in the first statement of the repetition code. The statement

```
225  FOR I = 1 TO 17
```

readily conveys that we are going to count from 1 to 17. On the other hand, the statement

```
912  IF COUNT > 17 THEN 190
```

could mean other things besides the end of the loop process.

Now suppose we begin a loop with

```
300  LET J = 1
```

and then inadvertently signify the end with

```
490  NEXT J
```

Mercifully BASIC will tell us what happened. The message

```
NEXT without FOR in 490
```

will report where to look for the trouble.

Similarly, BASIC will report a missing NEXT statement with the following:

```
FOR without NEXT in 310
```

These messages are very helpful to us when we are testing our programs. Of course, one of our objectives is never to cause these messages to ap-

pear. It is possible to write programs that work the first time. But we will occasionally bring forth error messages in spite of our best efforts not to.

....SUMMARY

FOR, NEXT, and STEP are revealed as the way to set up repeated operations when we know where we want to start and end. FOR NUMBER= FIRST TO LAST STEP JUMP begins the loop. NEXT NUMBER ends the loop.

Problems for Section 4-1 .....

1. Write a FOR . . . NEXT loop to count from 10 to 20.
2. Write a FOR . . . NEXT loop to count from 93 to 80 by twos.
3. Write a program to display "I like BASIC" six times.
4. Write a program to display the integers from 1 to 15 paired with their reciprocals.
5. Write a program to display decimal values for sevenths. That is, display 1/7, 2/7, . . . 6/7, and 7/7.
6. Do Problem 5 for elevenths.
7. Write a program to calculate the sum of the counting numbers from 1 to 100. (You probably don't want the computer to display the values of the loop variable in this one.)
8. Examine the following program:

```
100  FOR I = 1 TO 1.3 STEP .1
110   PRINT I
120  NEXT I
```

   What values do you think it will display? RUN it. Do you get what you expect? Change line 100 to FOR I = 1 TO 1.2 STEP .1.
9. Write a loop to display the four numbers you expected in the first part of Problem 4-8.

4-2...More Bounce to FOR and NEXT

In Programs 4-1 and 4-2 we simply displayed the FOR variable to demonstrate that the structure performs as advertised. We usually are interested in other things.

Let's pursue the bouncing steel ball a little more. By writing a routine to simulate the actual bouncing we can supplement it to calculate more information. For example, suppose we want to learn the total distance the ball travels from the point we release it to the top of the eighth bounce. First, we write the bouncing simulation with FOR and NEXT. See Program 4-3.

```
   98   REM ** Simulate a bouncing steel ball
   150  PRINT "Bounce","Height"
   180  HEIGHT = 10
-->196  :
   198  REM ** Bounce here
   200  FOR BOUNCE = 1 TO 8
-->210   HEIGHT = HEIGHT * .9
   220   PRINT BOUNCE, HEIGHT
   290  NEXT BOUNCE
   500  PRINT "Done"
```

Program 4-3. Bouncing a steel ball with FOR and NEXT.

Program 4-3 simply uses FOR and NEXT to perform the job of our earlier ball-bouncing program. We have added a feature here that sometimes makes programs easier to read. We have used a colon by itself in line 196. This just creates some white space within the program.

Now it is a simple matter to incorporate the logic to calculate the total distance traveled. We simply initialize DISTANCE to zero outside the FOR . . . NEXT loop and add in the length of the downward and the length of the upward path. On the downward path the ball travels the old height; on the upward path the ball travels the new height. Just insert two distance-adding statements—one before line 210 and one after line 210 in Program 4-3. Next, it would be a good idea to display the distance along with the bounce number and the height. This is done in Program 4-4.

```
   98   REM ** Simulate a bouncing steel ball
-->150  PRINT "Bounce", "Height", "Total Distance"
   180  HEIGHT = 10
   190  DISTANCE = 0
   196  :
   198  REM ** Bounce here
   200  FOR BOUNCE = 1 TO 8
-->205   DISTANCE = DISTANCE + HEIGHT 'Add downward path
   210   HEIGHT = HEIGHT * .9
-->215   DISTANCE = DISTANCE + HEIGHT 'Add upward path
-->220   PRINT BOUNCE, HEIGHT, DISTANCE
   290  NEXT BOUNCE
   500  PRINT "Done"
```

Program 4-4. Calculate the distance for a bouncing ball.

....Apostrophe

Look at lines 205 and 215 in Program 4-4. We have used a new feature of BASIC to document those statements. The apostrophe may be used just like a REM statement. The apostrophe is especially convenient when we want to comment on the purpose of a single line. Everything following an apostrophe on a line is ignored during program execution. Here it is crystal clear what those two lines do. The apostrophe may be used instead of a colon to provide an almost-blank line. Techniques of this sort may be used

to make programs ever more readable. Anything that makes programs more readable makes them easier to work with.

```
Bounce          Height          Total Distance
 1               9               19
 2               8.1             36.1
 3               7.29            51.49
 4               6.561           65.341
 5               5.9049          77.8069
 6               5.31441         89.0262
 7               4.78297         99.1236
 8               4.30467         108.211
Done
```

Figure 4-2. Execution of Program 4-4.

Once again, we might want to pretty up our program display. Look at lines 150 and 220. Let's replace them with

```
150  PRINT "Bnce  Height  Total distance"
220  PRINT USING "##    ##.##   ###.##"; BOUNCE, HEIGHT, DISTANCE
```

And now see the display in Figure 4-3.

```
Bnce  Height  Total Distance
 1     9.00    19.00
 2     8.10    36.10
 3     7.29    51.49
 4     6.56    65.34
 5     5.90    77.81
 6     5.31    89.03
 7     4.78    99.12
 8     4.30   108.21
Done
```

Figure 4-3. Execution of Program 4-4 with PRINT USING.

....SUMMARY

We have used a program simulating a bouncing ball to demonstrate FOR and NEXT. In this program we are interested in several quantities besides the value of the loop variable. The apostrophe has been introduced as an alternative vehicle for including comments right within a BASIC program. This is especially desirable when we would like to comment on the purpose of a single line in a program. A colon or an apostrophe may be used as the only character on a numbered line to break up a program listing. This technique can be used to make programs significantly easier to read.

Problems for Section 4-2 .....

1. Write a program to print a table containing a number, its square, and its cube. Do this for values in a range from 1 to 20.

2. Write a program to display the first 20 Fibonacci numbers. This is a sequence for which the first two elements are both ones and succeeding values are obtained by adding the previous two elements.
3. Factorials are used a great deal in probability calculations. Factorial four is written 4! and is calculated by multiplying all the counting numbers from 1 to 4. That is:

   4!= 4 * 3 * 2 * 1

   Write a program to display the factorial of a value entered using an INPUT statement. Note: 0! is defined as 1.
4. In the song "The Twelve Days of Christmas," gifts are given to the singer according to a progression of numbers. The first day she got a partridge in a pear tree. On the second day she got two turtledoves and a partridge in a pear tree. On the final day she received 12 + 11 + . . . + 2 + 1 gifts. Write a program to display the total number of gifts she received. If she had to return one each day, on what day would she return the last gift?

4-3...Let's Explore Interest

Interest is paid for the use of money. It is a kind of rent. Simple interest is paid on an annual basis. Simple interest at 18% on $1000 comes to $180 for the year. Compound interest is calculated and added to the principal at intervals. With the proliferation of computers, daily compounding has become commonplace. If interest is compounded daily, the interest is added to the principal daily. The annual percentage is prorated. That is, an 18% annual interest rate comes to 18/365 or about .04931506849315069% per day. While there is a formula for this, we can easily calculate compound interest using a FOR loop. See Program 4-5.

```
100  PRINT "Calculate compound interest."
110  PRINT
198 :
200  INPUT "  Principal"; P
210  INPUT "Annual rate"; AR
212  AR = AR / 100
215  PRINT
220  DR = AR / 365 'Daily Rate
230  FOR DAY = 1 TO 365
240   INTEREST = P * DR
250   P = P + INTEREST
260  NEXT DAY
270  PRINT P, "After one year"
```

Program 4-5. Calculate compound interest.

```
Calculate compound interest

  Principal? 1000
Annual rate? 18

 1197.16       After one year
```

Figure 4-4. Execution of Program 4-5.

We see that we pay an extra $17.16 for compounding instead of using simple interest. Of course, if we are doing the lending, that looks pretty good.

....Fibonacci Numbers

Fibonacci numbers describe a number of natural phenomena and are of interest to mathematicians. A sequence of numbers is involved. The first two numbers in the sequence are both 1. Following this, each number in the sequence is the sum of the previous two values. So, the third element is 1 plus 1 or 2. Let's write a little program to display a few Fibonacci numbers. See Program 4-6.

```
98   PRINT "Fibonacci numbers:"
100  B = 0 : FIB = 1
200  FOR J = 1 TO 10
210   PRINT FIB;
220   A = B : B = FIB
250   FIB = A + B
290  NEXT J
```

Program 4-6. Display Fibonacci numbers.

Here we save the last two elements in the sequence in variables A and B at all times. To get the sequence going we artificially set them both equal to 0.

```
Fibonacci numbers:
 1  1  2  3  5  8  13  21  34  55
```

Figure 4-5. Execution of Program 4-6.

As we get more and more into programming we will find ourselves using loops everywhere. We will get lots of practice with FOR and NEXT throughout the rest of our programming career.

Problems for Section 4-3 .....

1. Compare the interest on $1000 for one year at 18% with the interest at 12%.
2. Modify Program 4-6 to display the square of an element in the sequence and the product of the elements immediately before and

immediately after. Also display the result of subtracting one from the other.

3. How many Fibonacci numbers can be expressed with six or fewer digits?

4-4...Nested Loops

Nested loops occur whenever we program one loop within another.

....Another Look at Compound Interest

Suppose we need to calculate compound interest over several years. Let's display the compound amount each year on $1000 for five years. We can easily do this by enclosing the yearly-interest calculation of Program 4-5 within a FOR loop that enumerates the years. See Program 4-7.

```
   100  PRINT "Calculate compound interest."
   110  PRINT
   198 :
   200  INPUT "  Principal"; P
   210  INPUT "Annual rate"; AR
   212  AR = AR / 100
   215  PRINT
   220  DR = AR / 365 'Daily Rate
-->225  FOR YEAR = 1 TO 5
   230   FOR DAY = 1 TO 365
   240    INTEREST = P * DR
   250    P = P + INTEREST
   260   NEXT DAY
-->270   PRINT P, "After"; YEAR; "years"
-->280  NEXT YEAR
```

Program 4-7. Compound interest for several years.

In Program 4-7, we simply added lines 225 and 280 to carry the calculation through five years using a FOR loop with YEAR as the variable, and we changed line 270 to display a more appropriate message. Notice that the DAY loop is entirely enclosed within the YEAR loop. It looks like this:

```
225  FOR YEAR = 1 TO 5
230   FOR DAY = 1 TO 365
260   NEXT DAY
280  NEXT YEAR
```

These are known as nested loops. We have indented each loop to show what statements belong together here. The loops must be closed with NEXT in the reverse order of that in which they were opened with FOR.

```
Calculate compound interest

  Principal? 1000
Annual rate? 18

 1197.16      After 1 years
 1433.2       After 2 years
 1715.78      After 3 years
 2054.07      After 4 years
 2459.06      After 5 years
```

Figure 4-6. Execution of Program 4-7.

....Pythagorean Triples

There is an interesting set of right triangles with sides whose lengths are integers. Any three integers that can represent the sides of a right triangle are referred to as a Pythagorean triple. If we label the sides of a right triangle as LEG1, LEG2, and HYPOT, then the Pythagorean theorem tells us that the sum of the squares of the two legs equals the square of the hypotenuse or

LEG1*LEG1 + LEG2*LEG2 = HYPOT*HYPOT

Suppose we want to find all Pythagorean triples with either leg up to 25. We will write a program based on the following nested loops:

```
   110  FOR LEG1 = 1 TO 25
-->120    FOR LEG2 = 1 TO 25
-->140      FOR HYPOT = 1 TO 50
   190      NEXT HYPOT
   200    NEXT LEG2
   210  NEXT LEG1
```

Here we have nested loops three deep. That's perfectly okay, but we should be a little cautious about this process. Even though computers are fast, it is easy to program a task that will take too long for the computer to do. We have programmed 31250 steps for the computer. While this is not much of a challenge for the computer, we will wait a few minutes for the full results. It will be worthwhile for us to study the problem with an eye toward eliminating some work for the computer.

If we let values for both legs range from 1 to 25, then we will get 3, 4, 5 and 4, 3, 5 in the result. We can easily eliminate duplication by changing line 120 so that the value of LEG2 begins with the current value of LEG1, or even LEG1 + 1.

```
120    FOR LEG2 = LEG1 + 1 TO 25
```

That saves a lot of unnecessary steps for the computer.

Now look at line 140. Certainly we do not have to make the computer use values for the hypotenuse beginning with 1. We could safely begin

with LEG2. The hypotenuse must be at least as long as the longer leg. so line 140 becomes

```
140    FOR HYPOT = LEG2 TO 50
```

Using 50 as the upper limit is fine. We will use other means to ensure that the computer doesn't have to test values higher than necessary.

For each set of three numbers we need to compare the sum of the squares of the two legs with the square of the hypotenuse. If the sum of the squares of the legs is greater than the square of the hypotenuse, then we try the next value for the hypotenuse. If the sum of the squares of the legs is less than the square of the hypotenuse, then we have overshot on the hypotenuse and it is time to try a new value for LEG2. If they are equal, we display the three lengths and proceed to the next value for LEG2.

```
  100  PRINT "Pythagorean triples"
  110  FOR LEG1 = 1 TO 25
  120   FOR LEG2 = LEG1 + 1 TO 25
  140    FOR HYPOT = LEG2 TO 50
-->145     IF LEG1*LEG1 + LEG2*LEG2 < HYPOT*HYPOT THEN 200
-->150     IF LEG1*LEG1 + LEG2*LEG2 > HYPOT*HYPOT THEN 190
-->180     PRINT LEG1; LEG2; HYPOT
  182      GOTO 200
  190    NEXT HYPOT
  200   NEXT LEG2
  210  NEXT LEG1
```

Program 4-8. Display Pythagorean triples.

You might wonder why we didn't square values using an exponent in lines 145 and 150. Because of the way that BASIC raises to a power, small calculation errors could result in missing some of the triples we want.

Line 145 will cause execution to jump out of the loop before the loop variable has run its course. In this situation, BASIC keeps track of the fact that that particular loop is still active. This is done in an area of memory called the stack. In very large programs this can cause the stack to overflow. We can avoid this situation by using the following replacement for 145:

```
145     IF LEG1*LEG1 + LEG2*LEG2 < HYPOT*HYPOT THEN HYPOT=50 : GOTO
        190
```

```
Pythagorean triples
 3  4  5
 5  12  13
 6  8  1Ø
 7  24  25
 8  15  17
 9  12  15
 1Ø  24  26
 12  16  2Ø
 15  2Ø  25
 18  24  3Ø
 2Ø  21  29
```

Figure 4-7. Execution of Program 4-8.

....TAB( )

We could line up those columns nicely with comma spacing on an 80-column display. Or we could use TAB in the PRINT statement of line 180. TAB(X) causes the next printed item to begin in a column numbered X. The first column is numbered 1. While the HTAB statement positions absolutely on the line, TAB(X) cannot back up on the line. An attempt to do so results in the display moving to the next line. We could replace line 180 in Program 4-8 with

```
18Ø     PRINT LEG1; TAB(5); LEG2; TAB(1Ø); HYPOT
```

to achieve a nicely spaced display. Notice in Figure 4-8 that the columns are left-justified. If what we want is right justification, then we can easily employ PRINT USING to do the display. BASIC is providing us with a wide variety of options for producing nicely formatted reports.

```
Pythagorean triples
 3   4    5
 5   12   13
 6   8    1Ø
 7   24   25
 8   15   17
 9   12   15
 1Ø  24   26
 12  16   2Ø
 15  2Ø   25
 18  24   3Ø
 2Ø  21   29
```

Figure 4-8. Execution of Program 4-8 with TAB( ) in PRINT.

Problems for Section 4-4.....

1. Here is a formula for compound interest:

$$A = P(1 + I)^N$$

where

A = Compound amount
P = Principal
I = Interest rate per interest period
N = Number of interest periods

Write a program to calculate interest using this formula.

2. 9, 40, 41 and 12, 35, 37 are Pythagorean triples. They do not appear in the execution of Program 4-8. Find some additional Pythagorean triples using Program 4-8 by raising the upper limit on LEG2 to 50 and the upper limit on HYPOT to 75.
3. Write a program to display a multiplication table. Select an upper limit so that you can produce a nice display on the screen. Use nested loops and PRINT USING.

4-5...More about NEXT

BASIC allows us to close a FOR loop with the keyword NEXT by itself. This is good and it is bad. It is bad because the lack of the variable name obscures which loop is being terminated. The computer will take care of it, but we will have a hard time following the program by just reading the code. The benefit is that NEXT by itself executes somewhat faster then NEXT with a variable. It also takes one byte less memory for each character in the variable name. You are encouraged always to include the variables unless speed or memory becomes more important than having a readable program.

If several FOR loops have a common end point it may be designated with a single NEXT statement. For example,

```
NEXT T, N, L
```

In this case the single NEXT statement must name the variable for each loop involved. Omitting any will evoke the

```
FOR without NEXT
```

error message. Including the loop variable in all NEXT statements and matching every FOR with a corresponding NEXT enables us to maintain nice spacing and clear documentation.

We describe these short cuts, not for you to use them, but to equip you to recognize them in other people's programs. People do write programs with no variables in NEXT. We just don't want you to be thrown off by that.

Consider Program 4-9.

```
100  FOR I = 1 TO 2
110   FOR J = 1 TO 3
120    FOR K = 1 TO 4
150     PRINT K, J, I
170    NEXT K
180   NEXT J
190  NEXT I
```

Program 4-9. Note nicely matched NEXT statements.

Naming the variable in each NEXT and using proper indentation together make this program very easy to read.

SIDELIGHT 4

Another Look at Precision

As we said in Sidelight 2, BASIC normally works with single-precision numeric values. This is entirely adequate for most computing projects. For many years it was the only degree of precision available.

BASIC offers two more precisions for numeric values. We can restrict values to the range of integers from −32768 to 32767. Results outside that range are considered overflow values. We can work with double-precision values. Double precision gives us 16 significant digits and a range from about −1.7E+38 to 1.7E+38.

....%, !, and # Precision Indicators

There are several ways to get the precision we want. At any time we can declare a variable as integer, single-precision, or double-precision by appending a special character to the variable name. This is the way we distinguish between a string variable and a numeric variable. A percent sign (%) designates an integer, an exclamation point (!) designates a single-precision variable, and we get double precision with a number sign (#). We saw the exclamation point in Sidelight 2. Any numeric variable without any of these symbols is considered a single-precision variable. The variable X and X! are the same. Any of these symbols may be used for constants as well. So, X% is an integer variable and 12345# is a double-precision value.

....Some Double-Precision Examples

Besides explicitly typing the number sign, we can cause BASIC to work with double precision in other ways. When numbers are entered in certain forms, they are processed and stored in standard double precision. When a number is expressed in eight or more digits or when we use D, instead of

E, to specify an exponent, the result is in double precision. D-format is just like E-format, except that the decimal part is expressed with up to 16 digits. We can demonstrate this by entering some program statements to see how they are transformed in a LISTing.

```
100  X# = 12345678
110  Y# = 12345D15
LIST
100  X# = 12345678#
110  Y# = 1.2345D+19
```

We can determine the actual range for double-precision values by asking BASIC to divide by zero in double precision.

```
PRINT 1# / 0#
Division by zero
 1.701411834604693D+38
```

It can be fun to explore things in different precisions.

```
100  A# = 3  / 17  : PRINT A#
120  B# = 3# / 17  : PRINT B#
140  C# = 3  / 17# : PRINT C#
160  D# = 3# / 17# : PRINT D$
```

These four lines will tell us something very important about how numbers of different precisions are handled. Let's execute them.

```
.1764705926179886
.1764705882352941
.1764705882352941
.1764705882352941
```

Now what do we do? Which of the two different values is correct? Since each of the last three values is generated by appending a number sign to the numeric value itself, we might favor that result. But can all 16 digits really be right?

Let's write a little program to perform infinite-precision division for us. See Program 4-10.

```
   300  N = 3 : D = 17
-->320  Q = N \ D : PRINT Q;
-->360   N = (N - Q*D) * 10
   370   GOTO 320
```

Program 4-10. Infinite-precision division.

Note the use of integer division in line 360. If this is really infinite precision, then we will just have to press CTRL-C to stop it. This program simply performs long division a digit at a time. Let's see. . .

```
 Ø  1  7  6  4  7  Ø  5  8  8  2  3  5
 2  9  4  1  1  7  6  4  7  Ø  5  8  8
 2  3  5  2  9  4  1  1  7  6  4  7  Ø
 5 ^C
Break in 32Ø
```

Figure 4-8. Execution of Program 4-10.

That's our answer. But it's so hard to read! We need to work out a way to eliminate the spaces. This is an interesting place to take advantage of PRINT USING. If we include a semicolon at the end of a PRINT USING statement, the next display continues on the same line. So line 320 becomes

```
32Ø  Q = N \ D : PRINT USING "#"; Q;
```

Now let's see it. Look at Figure 4-9.

```
Ø1764705882352941176470588235294117647Ø5
8823529411764705882352941176470588235294
1176470588235294117647Ø5882352941^C
Break in 32Ø
```

Figure 4-9. Execution of Program 4-10 with PRINT USING.

Well, you get the idea. It just happens to come out to a 16-digit repeating decimal. We could spruce this program up a bit by inserting the decimal point in the correct place. It goes between the 0 and the 1 on the first line of the program execution display. Now we can confirm that this particular calculation is accurate to 16 significant digits as long as we force the calculation routine into double precision by using the number-sign designation on one of the numbers in the calculation. While the result for 3/17 displayed 16 digits, we can easily see that they are not any more accurate than single precision.

....Integer Values

If our program has lots of loops that are running pretty slowly, we can consider using integer variables for the loop index.

```
1ØØ  FOR I% = 1 TO UPPERLIMIT
12Ø  NEXT I%
```

runs in about 60% of the time it takes for

```
1ØØ  FOR I = 1 TO UPPERLIMIT
12Ø  NEXT I
```

Furthermore, calculations on integer variables require less computer

time than calculations on single-precision variables. And single-precision calculations take far less computer time than double-precision ones. Even double-precision display is very slow. We may not use a double-precision variable as the loop variable.

Chapter 5

Packages in BASIC
(Functions and Subroutines)

BASIC is made up of several kinds of features. There are the keywords and the loop structures. We have line numbers, numeric variables, and string variables. There are the various arithmetic operations available to us. In this chapter we are going to add a tremendous collection of tools to our programming repertoire. We are going to discuss many of the functions available. In addition, we are going to learn about subroutines.

A function is a process that returns a value. For example, there is a function that returns the square root of a number. While we could write the necessary BASIC code to do that, it is desirable to have the programming language do it for us. Why should every user of BASIC have to write it? Another function returns the number of characters in a string variable. BASIC includes many such functions as features of the language. In addition, we may create our own programmer-defined functions.

A subroutine is like a mini-program. It is a segment of a program that we may isolate and use from anywhere else in our program. Subroutines are useful whenever we need the same calculation at many different places in a program. Furthermore, subroutines make it possible to partition the work of a big program into little jobs. This helps us to clear our mind to concentrate on a smaller programming task at any one time.

5-1...Introduction to Numeric Functions

....SQR (Square Root)

We can easily write a little program to display the square roots of the integers from 1 to 10. The square root function is designated by SQR. The

value for which we require the square root is enclosed in parentheses. The value in parentheses is called the "argument" of the function. See line 110 in Program 5-1.

```
   100  FOR J = 1 TO 10
-->110   PRINT J, SQR(J)
   120  NEXT J
```

Program 5-1. Display some square roots.

Here we simply display the value on the screen.

```
1             1
2             1.41421
3             1.73205
4             2
5             2.23607
6             2.44949
7             2.64575
8             2.82843
9             3
10            3.16228
```

Figure 5-1. Execution of Program 5-1.

In Program 5-1, SQR(J) takes on a value during execution. We may use that value in many of the ways that we use other values. The following statements demonstrate some ways.

```
100  X = SQR(HEIGHT)
100  SIDE = SQR(X*X + Y*Y)
100  T = SQR(J) - 2*T + 18*R9
```

....INT (Greatest Integer)

The INTeger function returns the largest integer not greater than the argument. So INT(5.0) becomes 5 and INT(5.99) also becomes 5. INT(−6.5) becomes −7. The result shouldn't be −6 because that is greater than −6.5. INT is not the same as doing integer division using \. The difference shows up for negative results.

```
INT(6 / -4)  evaluates as  -2
```

while

```
6 \ -4     works out to  -1
```

This function is often used to round off values. The most frequent application is in financial calculations. We always round off to the nearest hundredth of a dollar to come out to whole cents. All values half a cent or more we round up and values less than half a cent we round down. So what we do first is multiply the dollar value by a hundred to get cents.

Now if we just cut off the decimal part we will always round down. Since we only want to round down for values less than half a cent, we add half a cent and then cut off the decimal part. We end up with an expression that performs the cents rounding for us.

```
INT( DOLLARS * 100 + .5 )
```

tells us the number of cents. Now all we need to do is divide by a hundred to get the decimal point back in. The final expression is

```
INT( DOLLARS * 100 + .5 ) / 100
```

If this evaluates to $54.40, BASIC will PRINT without the trailing zero. And if it comes out to a whole dollar, that is the way it will display. In order to have any trailing zeros to the right of the decimal point displayed, we should display with PRINT USING.

....Factors

Since we can now take the greatest integer of a value, we can also decide whether one number divides evenly into another. This means that we can find factors of integers. Factors occur in pairs. See Table 5-1.

2 * 6 = 12
3 * 4 = 12
4 * 3 = 12
6 * 2 = 12

Table 5-1. Factor pairs.

In fact, all pairs of factors occur twice, except in the case of a perfect square. Also, every factor greater than the square root is paired with a factor less than the square root. We can test all integers up to the square root of a number and we will have all possible factor pairs. Study Program 5-2.

```
100  INPUT "Find factor pairs of"; N
120  FOR D = 2 TO SQR(N)
130   Q = N / D
140   IF Q <> INT(Q) THEN PRINT D; Q
180  NEXT D
```

Program 5-2. Find factor pairs.

....SUMMARY

We have introduced the SQR( ) and INT( ) functions. These are built-in features of BASIC to calculate the square root and greatest integer.

Problems for Section 5-1 .....

1. Write a program to display the square roots of integers from 1 to 20 rounded off to the nearest tenth.

2. Do Problem 1, displaying the values with PRINT USING.
3. Modify Program 5-2 to find the smallest prime factor of an integer.
4. Write a program that requests a numeric date in the form YYMMDD. One of the reasons that we like to use this numeric YYMMDD form for the date in a computer is that if we sort by that number, the result will be in chronological order. First, verify that it is a possible date. Then display the date in the form YY MM DD. That is: 790121 becomes 79 1 21.

5-2...String Functions

There are a number of functions that will help us handle string values. Included are functions to determine the length of a string, pick strings apart, and functions that generally simplify programming with strings.

....LEN (Length of a String)

LEN(A$) returns the number of characters actually stored in the string variable A$. This value can range from 0 to 255. Whenever we analyze a string, character by character, the LEN function is useful for determining how many characters to look at.

....ASC (ASCII Value)

The ASC function returns a number corresponding to the internal code that BASIC uses to represent characters. For example,

```
ASC("A")
```

has a value of 65. The ASCII value for Z is 90. ASCII is the American Standard Code for Information Interchange. It is used by a great many computer systems. A partial table of ASCII values is presented in Appendix D. ASC(V$) returns the ASCII value associated with the first character in the string variable V$.

....CHR$ (Character Whose ASCII Code Is Given)

The CHR$ function is the reverse of the ASC function.

```
CHR$(65)
```

returns the character "A" since "A" is represented by 65 using ASCII.

....STR$ (Convert Numeric to String)

The STR$ function converts a numeric value to string format. STR$(N) converts the internal binary code used to represent the numeric value of N into the ASCII code used for each of the digits. Let's examine the effect of a statement such as

```
100  T$ = STR$(X)
```

While X stores a numeric value that we may use directly in arithmetic calculations, T$ stores the digits of the number as string characters. Thus, T$ permits us to manipulate the digits using string functions and techniques. T$ will begin with a space or a minus sign. That is just the way it is.

....VAL (Value of a String)

VAL is the reverse of STR$. VAL(N$) returns a numeric value. If N$ is a string of digits, VAL converts them into the binary format used for storing numbers. If the first character could not be part of a number, a zero is returned. If the function is successful in converting the beginning of a string, it continues to the end of the string or until it encounters an impossible character. Thus,

```
VAL("7 Days in the week")
```

will convert to

```
7
```

The VAL function handles E-format just fine. The numeric value will be converted to standard form.

```
VAL("312E-2")
```

becomes

```
3.12
```

and all is well. D-format works, too.

....LEFT$, MID$, and RIGHT$

The LEFT$, MID$, and RIGHT$ are among the most often used string functions. These functions return a string value. As the names imply, these functions enable us to pick strings apart.

LEFT$(A$,5) returns the five leftmost characters in the string variable A$. If A$ contains fewer than five characters, the LEFT$ returns whatever is there. RIGHT$ performs the exact same duty on the right end of a string.

MID$ allows us to extract characters from the interior of a string. Well, we don't remove them, we just obtain a copy of them. MID$(A$,7,3) refers to the three characters of A$ beginning with the seventh character. And MID$(A$,K,1) specifies the Kth character in the A$ string.

All numeric arguments must be in the range 0 to 255 or you will get the

```
Illegal function call
```

error message. In addition, for the expression MID$(A$,J,K), we get the same error message if the value of J goes to zero.

Let's enter some days of the week in a string and separate them for display. See Program 5-3.

```
100  WEEK$ = "MonTueWedThuFri"
120  PRINT LEFT$(WEEK$,3)
130  PRINT MID$(WEEK$,7,3)
140  PRINT RIGHT$(WEEK$,3)
```

Program 5-3. Demonstrate LEFT$, MID$, and RIGHT$.

The expression LEFT$(WEEK$,3) represents the three leftmost characters in the character string WEEK$, or Mon. Similarly, RIGHT$-(WEEK$,3) becomes the three rightmost characters in WEEK$, giving us Fri. MID$ is a little different. MID$(WEEK$,7,3) is the three characters beginning with the seventh character in the string WEEK$, producing Wed.

```
Mon
Wed
Fri
```

Figure 5-2. Execution of Program 5-3.

We have used MID$ as a function here. MID$ may also be used as a command to assign characters anywhere in a string variable. If

```
X$ = "MonTueWed"
```

and the following statement is executed:

```
MID$(X$,4,3) = "Feb"
```

then X$ will contain "MonFebWed".

....INSTR

INSTR is used to find a string of characters in another string. For example, suppose we need to enter the day of the week and have the program use the day number for further calculation. Program 5-4 is a little routine that will report the character position of a day.

```
689  REM ** What day is this?
700  WEEK$ = "SUNMONTUEWEDTHUFRISAT"
710  INPUT "Day"; DAY$
720  DAY$ = LEFT$(DAY$,3) '3 characters to match
740  P    = INSTR(WEEK$,DAY$)
760  PRINT "Found in position"; P
```

Program 5-4. Demonstrate INSTR.

We have set up a string with the days of the week for our program to match against a day name entered from the keyboard. Note that we will have to

enter all uppercase to obtain a match. We have saved only the first three characters as well. INSTR requires two strings. The function will look for an occurrence of the second string in the first one. The number we get is the position of the first character in the match. Let's see it run in Figure 5-3.

```
Day? FRIDAY
Found in position 16
Ok
RUN
Day? Friday
Found in position 0
```

Figure 5-3. Execution of Program 5-4.

We can see that INSTR returns 0 when it fails to find a match. While we found FRIDAY in position 16, we really would like to know which group of three characters that is to know the day number. That is easy. Just divide by 3 using integer division and add 1. But now we have to test for P = 0 first to rule out a bad response from the keyboard.

INSTR also allows us to begin anywhere for the search.

```
PRINT INSTR(4,"ABXDEFGHIJXK","X")
 11
```

INSTR didn't report the X found at 3 because we told it to begin with character number 4.

....STRING$ (String of Character)

STRING$(1,J) simply returns the character whose ASCII value is J—just like CHR$. STRING$(K,J) returns K of them. STRING$(5,37) becomes five percent signs. This function is not essential for programming. However, it is handy for filling in reports and dressing up program display in general. Just don't let the value of K or J become less than 0 or greater than 255 or you will get an

```
Illegal function call
```

STRING$ may also be used with a string argument to indicate the character to be repeated.

```
STRING$(10,"-")
```

will become ten dashes. This function can be used to output spaces in a display, but a special SPACE$ function is provided for this.

....SPACE$

SPACE$(15) becomes 15 spaces. SPACE$(X) becomes X spaces. This SPACE$ string can be used in general string assignment and display statements of all kinds.

....SUMMARY

We have introduced LEN, ASC, CHR$, STR$, VAL, LEFT$, MID$, RIGHT$, INSTR, STRING$, and SPACE$. We can use all of the earlier string concepts here. Thus, expressions such as the following all perform sensibly:

```
445  B$ = B$ + SPACE$(3) + MID$(A$,J,2)
560  C$ = MID$(X$,4,6) + LEFT$(3)
210  D$ = RIGHT$(D$,LEN(D$)-1)
```

MID$, besides being a function to isolate characters within a string, may be used as a command to assign characters anywhere in a string.

Problems for Section 5-2 .....

1. Write a program to display the contents of a string backwards.
2. Using HOME, HTAB 1, and string features from this section, write a program that scrolls a message across the screen horizontally.
3. Write a program to request a name, last name first followed by a comma and the first name. Have your program search for the comma and rearrange the name in first-name-first format. Thus, "Kennedy, John F." will become "John F. Kennedy". Note: the response to INPUT will have to be enclosed in quotes to get the comma into the string. Or use LINE INPUT. See Sidelight 3.
4. Modify Program 5-4 so that it calculates the correct day number rather than the character position.
5. Looking at the ASCII chart in Appendix D, note that the uppercase alphabet is found in the range from 65 to 90 and the lowercase alphabet from 97 to 122. In Program 5-4 use this information to force the characters in the string coming from the keyboard to uppercase.
6. Write a program that requests a date in a string in the form YY/MM/DD. Note that dates in this form can be sorted to arrange in real chronological order. First verify that it is a possible date. Then display the date in the form YY-Mmm-DD. That is: 79/02/21 becomes 79-Feb-21.
7. Write a program that requests a numeric date in the form YYMMDD. One of the reasons that we like to use this numeric YYMMDD form for the date in a computer is that if we sort by that number, the result will be in chronological order. First verify that it is a possible date. Then display the date in the form YY-Mmm-DD. That is: 790221 becomes 79-Feb-21.

5-3...Miscellaneous Functions

....ABS (Absolute Value)

The ABSolute value function is occasionally useful. ABS(X) changes the

sign of all negative arguments, returns zero when X equals zero, and gives X for positive values of X.

There is an interesting application that uses ABS. We can determine the minimum of A and B with the following expression:

```
(A + B - ABS( A - B )) / 2
```

A simple change makes this work for maximum. Alternatively we could obtain the minimum with an IF test such as

```
905  MIN = A
907  IF A > B THEN MIN = B
```

And here is yet another way:

```
200  IF A > B THEN MIN = B
               ELSE MIN = A
```

Either one produces the desired value.

Suppose we are testing for X = Y in an IF statement. If both values are decimal numbers, we may be satisfied if they are within .0000001 of each other and we don't care which is larger. This is just the place for ABS.

```
905  IF ABS(X-Y) < .0000001 THEN Code for match here
```

....SGN (Sign)

The SiGN function returns −1, 0, or 1 according to whether the argument is positive, zero, or negative. It doesn't get much of a workout.

....RND (Random Numbers)

One popular function is the random-number generator. The ability of the computer to bring forth random numbers makes it easy to create programs that do different things each time they are run. The function RND returns random values in the range 0 to 1. Let's see how it works. See Program 5-5.

```
100  FOR I = 1 TO 10
110   PRINT RND
120  NEXT I
```

Program 5-5. Demonstrate random numbers.

Program 5-5 simply displays each value. We may use these numbers in all the same ways we use any numbers in a program. We may assign them to variables and use them in calculations of all kinds.

```
.245121
.305003
.311866
.515163
.0583136
.788891
.497102
.363751
.984546
.901591
```

Figure 5-4. Execution of Program 5-5.

We might want to use random numbers to simulate some activity. This could be as simple as flipping a coin or as complex as modeling the traffic on a proposed road bridge.

Let's flip a coin. A coin can come up either heads or tails (coins land on edge infrequently enough so that we can ignore the possibility). We could divide the random numbers evenly by splitting the interval from zero to one at the .5 mark. It is almost as easy to multiply all values by two to make the interval become from zero to two. If we then apply the INT function, only two values are possible—zero and one. The beauty of this concept is that it also applies easily to many other random events. To roll dice we simply multiply by six to get integers from zero to five. In the case of the dice we can then add one to obtain the six different faces. Program 5-6 flips a coin ten times.

```
100  FOR I = 1 TO 10
110   COIN = INT( 2 * RND )
120   IF COIN = 0 THEN PRINT "Heads"
130   IF COIN = 1 THEN PRINT "Tails"
190  NEXT I
```

Program 5-6. Flip a coin ten times.

```
Heads
Heads
Heads
Tails
Heads
Tails
Heads
Heads
Tails
Tails
```

Figure 5-5. Execution of Program 5-6.

If we RUN Program 5-6 again we will get the same results. Computer random-number generators can be like that. They produce a sequence of numbers that is repeatable. Because the list is quite long, however, it is useful for most purposes. What we need is a way to change where we start

the list from one execution of a program to the next. Of course, BASIC provides for this. There are two methods. One is to place a value in parentheses to go with RND. The other is to use the RANDOMIZE statement.

....RND(X)

RND(X) is affected by the value of X. For a given negative value of X we get the same value of RND(X). Each different negative value for X gives a different starting point. For X equals zero we get the most recently generated random value. And if X is positive we get the same result as we get for RND without an argument. So to get different results from RUN to RUN simply execute RND(X) with a negative value for X once and positive values after the first. Making the value of X different each time provides the variety. You can get a value by asking for the person's name. Use the number of characters in the name or use the date or the time.

....RANDOMIZE

The RANDOMIZE statement permits us to enter a number from the keyboard that "seeds" the random-number generator. Give it a different seed and get a different sequence. When BASIC encounters the RANDOMIZE statement during execution it displays the following message:

```
Random Number Seed (-32768 to 32767)?
```

Simply give it a number off the top of your head.

Sometimes we would rather not have our program deliver the "Random Number Seed" message. In such a program we may use an alternative form:

```
RANDOMIZE N
```

where n is different from one execution to the next. We can get different values for N the same way we got them for X above. When we learn about data files we can even use a file to keep track of how many times the program has been executed and use that number to seed the random-number generator.

....FRE (Free Memory)

FRE(X) returns the number of unused bytes of memory. This is helpful to the programmer working on large programs. RUNning the program first will give a more realistic number. Each character requires one byte. An integer occupies two bytes. A single-precision number takes up four bytes, while a double-precision value uses eight bytes. See Sidelight 6 and Chapter 7. Memory use is especially important to the programmer working with arrays. See Chapter 6.

Any value, even a string, may be used for the argument of FRE. It is convenient to use 8 or 9 since they are right there with the right and left parentheses on the keyboard.

....Trigonometric Functions

ATN(Z), COS(Z), SIN(Z), and TAN(Z) are the four trigonometric functions. In each case the value of Z must be in radians rather than degrees. The trigonometric values are converted to six-digit precision.

Problems for Section 5-3 .....

1. Write a little routine to request a person's name and use the number of characters in the name to seed the random-number generator.
2. Flip a coin 200 times. Report on the number of heads and tails.

5-4...Programmer-Defined Functions (DEF FN)

....Numeric Functions

Earlier we saw an expression to round off values to the nearest hundredth.

```
INT( DOLLARS * 100 + .5 )
```

Every time we want a rounded-off value we have to repeat this expression. Sometimes we would rather use such an expression once and refer to it whenever needed. The programmer-defined function capability exists for just this purpose. Once, usually early in our program, we use a statement such as the following:

```
DEF FN ROUND(X) = INT( X * 100 + .5 )
```

Then wherever we need to round off to the nearest hundredth we simply incorporate FN ROUND(DOLLARS) as appropriate. The DEFined function statement must execute before any statement that refers to it. Failure to do this will evoke the

```
Undefined user function
```

error message. It is a good idea to place all DEF statements at the very beginning of your programs. Program 5-7 is a simple demonstration of a rounding-off function.

```
   90   PRINT "ROUNDED", "UNROUNDED"
-->100  DEF FN ROUND( X ) = INT( X * 100 + .5 ) / 100
   120  READ DOLLARS
   130  IF DOLLARS = 0 THEN END
-->140  PRINT FN ROUND( DOLLARS ), DOLLARS
   150  GOTO 120
   900  DATA 1.091, -17.569, 100.999
   910  DATA 17.569
   990  DATA 0
```

Program 5-7. Demonstrate rounding off with DEF FN.

Look at line 100. Note that while the variable used to define the function is X, when the computer gets to line 140 it replaces X everywhere with DOLLARS. The computer simply matches up the variable in parentheses in line 140 with whatever variable appears in parentheses in line 100. The X used in this way is called a dummy variable. It simply serves to tell the computer where to use the value named in the referencing statement later on. If we happen to use X for some other purpose in our program, that is all right. The two uses of the variable X do not interact at all.

```
ROUNDED        UNROUNDED
 1.09           1.091
-17.57         -17.569
 101            100.999
 17.57          17.569
```

Figure 5-6. Execution of Program 5-7.

We can round values off and store the result in a variable using a rounding-off function such as this. While we may also do rounding with PRINT USING, the results appear only in the display and are not stored in variables in the program.

Suppose we want to round off to different degrees of precision at different points in a program. We could do a DEF FN for each degree of precision. Or we could do one DEF FN in terms of precision for the whole job. We would like to have two values go into our function: the figure to be rounded and the degree of precision. We could use the number of decimal places as the degree of precision. Thus for rounding to the nearest hundredth we would use a 2. We can easily define such a function as follows:

```
100  DEF FN ROUND( X, Y ) = INT( X * 10^Y + .5 ) / 10^Y
```

We may use as many variables as is practical. Now if we code a line such as

```
190  PRINT FN ROUND( DOLLARS, P )
```

the computer will replace X with DOLLARS and replace Y with P. So it will evaluate

```
INT( DOLLARS * 10^P + .5 ) / 10^P.
```

Our function reference must have the same number of arguments as the function definition.

```
Syntax error in 140
```

will remind us that we failed to match the number of arguments. BASIC will swing into EDIT Mode to help us. We will see the same message if there is a syntax error in the DEF statement. Even though the error is in the DEF statement, BASIC will report the line number of the statement that references FN ROUND instead. So, if you don't see anything wrong in the statement that refers to the function, go right to the DEF statement and look at it. You will save a lot of staring.

If we name a variable in the function definition that is not in the argument list in parentheses following DEF FN . . . , then the computer will simply use that variable's actual value for the calculation.

....String Functions

Occasionally we want to define our own string functions. Take the case where we are trying to match a keyboard response to a set of possible responses stored in a string. If we are working with the full upper- and lowercase character set, we need to consider both upper- and lowercase. Notice that uppercase characters are in the range 65 to 90 and lowercase characters range from 97 to 122. See the ASCII chart in Appendix D. If we have an uppercase character we can change it to lowercase by adding 32. And to go the other way, we subtract 32. But outside the ranges 65 to 90 and 97 to 122, we want to leave the character unchanged. While there are a number of ways to do this, we can define a string function to do the job. Let's do lower to upper.

We need an assignment statement that will subtract 32 from any ASCII codes in the range from 97 to 122 and leave the others alone. With IF . . . THEN we would use a statement such as

```
300  IF ASC(A$)>96 AND ASC(A$)<123 THEN A$ = CHR$(ASC(A$)-32)
```

The logical expression ASC(A$)>96 AND ASC(A$)<123 in the IF statement evaluates as 0 for false and −1 for true. So, if we multiply that expression by 32, the result is 0 or −32. Now we have

```
32*( ASC(A$)>96 AND ASC(A$)<123 )
```

Then we add that to ASC(A$) to get the ASCII value for the required character. CHR$( ) gets it back to a string character. Putting this all together we have a DEFined function:

```
100 DEF FNLU$(X$) =
        CHR$( ASC(X$) + 32*( ASC(X$)>96 AND ASC(X$)<123 ) )
```

Now if we want to convert uppercase to lowercase, all we need to do is change the limits in our defined function and subtract instead of add. It is easy to write a little routine to test this.

```
200  INPUT "Enter a character"; C$
210   PRINT FNLU$(C$)
220   GOTO 200
```

Problems for Section 5-4

1. Define a function to convert centigrade to Fahrenheit. To get from centigrade to Fahrenheit we multiply by 9/5 and add 32.
2. Define a function to convert Fahrenheit to centigrade. To go from Fahrenheit to centigrade we subtract 32 and multiply by 5/9.
3. Define a function to return the average of three numbers.
4. Write the function to convert from upper- to lowercase.

5-5...DEF INT, SGL, DBL, STR (Variable Typing)

BASIC does most of its numeric calculations in single-precision mode. Occasionally we want to work in other modes. We saw in Sidelight 4 that we could designate specific variables for other modes of calculation by appending a percent sign or a number sign to the variable name. We can also use a special DEF statement to declare that all variable names beginning with certain letters shall be of a specific type.

```
100  DEFINT A-C,Q
110  DEFSGL H
120  DEFDBL X,Y,Z
130  DEFSTR N-P
```

Line 100 declares all variables beginning with letters A, B, C, and Q as integer variables. Line 110 sets H to single precision. X, Y, and Z are set to double precision in line 120. Any variable beginning with N, O, or P will be a string variable according to line 130. Any other variable names are unaffected by this. Variables redefined evoke no error message and the last definition prevails.

5-6...Subroutines (GOSUB and RETURN)

A subroutine is a side excursion. The program suspends what it is doing to execute program statements in another part of the program. Following this, it comes back to work on what it was doing when it took the excursion in the first place. We direct the computer to make the side excursion with the keyword GOSUB, and we signal the end with the keyword RETURN.

GOSUB 900 causes the computer to begin executing program statements beginning with line 900. This is just like GOTO 900 except that BASIC remembers its place with GOSUB. For GOSUB 900 to function

properly BASIC must encounter a RETURN statement. The program statements beginning with line 900 and ending with the RETURN statement make up the subroutine. If we cause the program to execute a RETURN statement in the absence of a GOSUB, we get

```
RETURN without GOSUB
```

The way to avoid this is to place subroutines at higher line numbers and include an END statement just before the first subroutine in the program. BASIC does not check for GOSUB without RETURN. That is the programmer's responsibility.

Suppose we are writing a program that has a lot of yes-no questions in it. We can write a subroutine to do this and use it from many places in the program. It is common practice to accept Y for yes and N for no. We should allow either upper- or lowercase. See Program 5-8.

```
1198  REM ** Yes-No processor
1200  PRINT QUESTION$; : INPUT ANSWER$
1205   ANSWER$ = LEFT$(ANSWER$,1)
1210   IF ANSWER$ = "Y" OR ANSWER$ = "y" THEN ANSWER = 1 : GOTO 1290
1220   IF ANSWER$ = "N" OR ANSWER$ = "n" THEN ANSWER = 0 : GOTO 1290
1230   PRINT "Answer Yes or No, Please"
1240   PRINT
1250  GOTO 1200
1290  RETURN
```

Program 5-8. Subroutine to process yes-no questions.

We might also consider defining the case-converter function here and using it instead of two separate tests for Y and y. Now we have a subroutine that we can use from anywhere in our program with

```
420  QUESTION$ = "Next menu" : GOSUB 1200
430  IF ANSWER = 1 THEN Yes code...
                   ELSE No code...
```

This technique saves us from having to include the actual code to process input from the keyboard at numerous points in our program. Further, it enables us to associate the whole idea of handling the keyboard with the simple statement GOSUB 1200. Thus we can concentrate on another portion of the program. For tasks of any size, it is impossible to hold the entire solution in our head at any particular instant in time. So any device we can develop that helps to simplify what we have to think about at any one moment is desirable.

It often works out that subroutines we write for one program are useful in other programs. Once this begins to happen, it becomes worthwhile to develop more sophisticated routines than we might for just one application. For example, suppose we are working on a routine to accept the date from the keyboard. We might, say, keep the year in the range 0 to 99, keep the month in the range 1 to 12, and keep the day in the range 1 to 31 and

be done with it. At the next level of sophistication we might additionally check that for month number 2 the day is in the range 1 to 29. Finally, we might develop a routine that distinguishes the 30- and 31-day months and allows for leap year. Once we have this routine fully tested we may then use it in all future programs dealing with a calendar.

Problems for Section 5-6

1. From Program 5-2 to find factor pairs, write a subroutine to find all prime factors of a value entered from the keyboard.
2. Write a subroutine to process the date in the form YY/MM/DD. Note that dates in this form can be sorted to arrange them in real chronological order. Verify that it could be a real date with a subroutine. Then create a string holding that date in the form YY-Mmm-DD. Example: 75/12/25 becomes 75-Dec-25.
3. Write a subroutine to process the date in the form YYMMDD. Note that dates in this form can be sorted to arrange them in real chronological order. Verify that it could be a real date with a subroutine. Then create a string holding that date in the form YY-Mmm-DD. Example: 751225 becomes 75-Dec-25.

SIDELIGHT 5

PEEK and POKE

....PEEK

PEEK is a function that returns a value.

```
PEEK(X)
```

reports the value stored in memory location X. On a 64K machine, memory runs from −32768 to 32767. Everything that the computer does is based on things that go on in memory. In addition, we have ways to use external storage such as disk. But PEEK(X) applies only to computer memory. That is where our programs run.

....POKE

POKE is a BASIC statement that writes a value to an address in memory. PEEK and POKE are used for advanced programming. However, it is instructive and interesting to consider a simple application.

....Ok

Suppose we tire of the "Ok" message that we see so much in BASIC. Since it must be in memory somewhere, we ought to be able to find it. We

need to look for an uppercase letter "O" followed by a lowercase letter "k". Referring to the ASCII chart in Appendix D, we see that "O" is represented by 79 and "k" is coded as 107. The rest is easy. We need a little program to look for a 79 followed by a 107. See Program 5-9.

```
98   REM ** Look for the Ok message
100  FOR MEM = 0 TO 32766
110   IF PEEK(MEM)   <> 79  THEN 190
120   IF PEEK(MEM+1) <> 107 THEN 190
130   PRINT MEM
190  NEXT MEM
```

Program 5-9. Look for "Ok" in memory.

Running this program reveals that "Ok" occurs at memory cells 3349 and 3350. So what? So we can now POKE an alternate message using a little program. It is even easy to do in immediate. Suppose we prefer "Hi" to "Ok". Look at the ASCII chart and find that "H" is 72 and "i" is 105. Let's do it:

```
POKE 3349, 72 : POKE 3350, 105
Hi
```

And we have our new greeting, just like magic. Or we might like to have a colon instead. That can easily be done with a colon followed by a space. Checking the ASCII chart again, or we could use ASC(":") and ASC(" "), we find that the numbers are 58 and 32.

```
POKE 3349, 58 : POKE 3350, 32
:
```

....Screen Window

In Applesoft, POKE can be used to create a window on the text screen. There must be a way to do it in SoftCard BASIC, too. Let's look at the layout. See Table 5-2.

	APPLESOFT	
Item	**Address**	**Value**
Left edge	32	0
Width	33	40
Top line	34	0
Bottom line	35	24

Table 5-2. Screen parameters in Applesoft.

The SoftCard documentation includes a chart that would enable us to determine the appropriate parameters for SoftCard BASIC. If that documentation isn't readily available, we can simply modify Program 5-9 to help us here, too.

```
98   REM ** Look for window parameters
100  FOR MEM = Ø TO 32764
110   IF PEEK(MEM)    <>  Ø THEN 190
120   IF PEEK(MEM+1) <> 40 THEN 190
130   IF PEEK(MEM+2) <>  Ø THEN 190
140   IF PEEK(MEM+3) <> 24 THEN 190
150   PRINT MEM
190  NEXT MEM
```

Program 5-10. Look for window parameters.

Program 5-10 leaves us without any results. We have looked at only part of memory. Let's look at the other part from 0 to −32768. Changing line 100 to

```
100  FOR MEM = Ø TO -32765
```

ought to do it. We get −4064. Following this, we see that the screen is converted into full-screen Lo-Res graphics. There is lots to explore here. We will just pursue the text window. Now we can control the text window according to Table 5-3.

	SOFTCARD	
Item	**Address**	**Value**
Left edge	−4064	0
Width	−4063	40
Top line	−4062	0
Bottom line	−4061	24

Table 5-3. Screen parameters in SoftCard BASIC.

This works only with the 40-character Apple screen that comes from memory-mapped video. Don't expect to use it with an external terminal. If you want to move the left edge, then you must set the width first. If the cursor stays outside the window when the POKEs are used, we have to take action to put it inside the window. HOME does it. And when we want our old screen back, we use TEXT. TEXT is a statement normally used to restore the text screen after using graphics. Pressing the RESET key does it, too. You may have to hold down the CTRL key at the same time. BASIC responds with

```
Reset error
```

Never mind. Unless you are working with files, just proceed with what you want to do.

Chapter 6

Pigeonholes Galore (Arrays)

We have been working with numeric variables for some time now. These variables have been very useful for many programs. We use them in FOR loops and calculations of all kinds. Numeric variables are important in making the computer such a useful tool. Likewise, we have taken advantage of the string-variable features of BASIC. The variables we have been using are all classed as simple variables. They hold a single value.

In this chapter we are going to take a quantum leap forward. While we have used a variable for each value in the past, we are going to see how to use the variable concept to encompass a large number of values, all with a single name. We are going to enter the world of the computer array.

Arrays are used for storing information that naturally belongs together. Tax tables, pricing structures, inventory information, and life insurance premiums are all appropriate for using arrays. Many times an array is useful for storing information about the workings of the program itself. We may use arrays for storing test scores, temperatures, random numbers, and lists of all kinds. We might use an array to store the days of the week or the months of the year.

6-1...Numbers, Numbers, and More Numbers (Numeric Arrays)

If we were going to store the high temperatures for each day of the week we might use SUNDAY, MONDAY, . . . , FRIDAY, and SATURDAY as

variables. That would be cumbersome. We would probably prefer to do the necessary calculation by hand. An array variable is a new kind of place to store values. An array may have as many pigeonholes as we need for any problem. We can designate WEEK as an array variable to contain values for the seven days of the week. To distinguish the several values stored in any array we use a value written in parentheses following the variable name. The value written in parentheses is called a subscript and each data value stored in the array is called an element. Thus, the temperature for SUNDAY could be stored in

```
WEEK(1)
```

In this case WEEK is the array name, and one (1) is the subscript. The temperature for Sunday is stored in the element designated as number 1. We read WEEK(1) as "WEEK-sub-one". In our example "WEEK-sub-seven" would be used for the temperature on Saturday. We can just as well code WEEK(X) OR WEEK(J9).

The first occurrence of any reference to a variable such as WEEK(1) establishes the array named WEEK. BASIC automatically provides eleven elements numbered from 0 through 10. In the next section we will learn how to specify exactly the number of elements we need for our situation.

The benefits of arrays are immediately available to us with no new requirements or keywords to learn. They are just like simple variables but with a special naming convention. We may use BASIC to assign values in all the ways we already know. Assignment (LET), INPUT, and READ all work the same as for simple variables. Array variables are used in calculations and in PRINT, LPRINT, PRINT USING, and LPRINT USING statements with ease. We may test the value of an array element in an IF statement.

Let's write a program to READ temperatures for a week, calculate the average, and find the highest and lowest temperatures. In order to do this we will set three initial values equal to the temperature of day 1. That is, on Sunday the SUM and the HIGH and LOW temperature are each equal to Sunday's temperature. Then for each of the other days of the week we will perform three tasks. We will add the day's temperature to the SUM. We will see whether today's temperature is lower than the current LOW. And we will determine whether today's temperature is higher than the current HIGH. Finally, we must display the results. See Program 6-1.

```
90   REM ** Enter the temperatures in array WEEK
100  FOR J = 1 TO 7
110   READ WEEK(J)
120  NEXT J
146 :
148  REM ** Set up initial conditions
150  SUM  = WEEK(1)
```

```
160  HIGH = WEEK(1) : LOW = WEEK(1)
196 :
198  REM ** Scan the week's temperatures
200  FOR J = 2 TO 7
210   SUM = SUM + WEEK(J)
220   IF WEEK(J) < LOW  THEN  LOW  = WEEK(J)
230   IF WEEK(J) > HIGH THEN  HIGH = WEEK(J)
240  NEXT J
290 :
300  PRINT "Average temp:"; SUM / 7
310  PRINT "Highest temp:"; HIGH
320  PRINT " Lowest temp:"; LOW
896 :
900  DATA 71, 77, 82, 76, 79, 72, 74
990  END
```

Program 6-1. Find average, highest, and lowest temperatures.

```
Average temp: 75.8571
Highest temp: 82
 Lowest temp: 71
```

Figure 6-1. Execution of Program 6-1.

As is often the case, there are lots of things we might do to change this program. We might want to round off the average temperature to the nearest degree. We might want to know on which days the high and low temperatures occurred. We might want to know how many times the temperature increased and decreased. These are left as exercises.

....Drawing Random Numbers from a Hat

Suppose we wish to simulate drawing numbers from a hat. We can easily do it with random numbers, provided that we may return each number to the hat before drawing the next one. If we must simulate drawing without replacement, then we must have a way of keeping track of what has been drawn. Here is an ideal application for an array. We simply set each element of an array equal to 1 and make the value 0 when that element has been selected. If the selected element is 1 then we know that it is available for use: use it and set it to 0. If a selected element is 0 then we know that it is not available for use and we must select again. Let's look at such a program to draw five numbers at random from among ten. See Program 6-2.

```
50   RANDOMIZE
90   REM ** Drawing five random numbers from among ten
96  :
98   REM ** Make all values available
100  FOR J = 1 TO 10
110   A(J) = 1 'Value available
120  NEXT J
196 :
198  REM ** Select five random values
```

```
   200  FOR J = 1 TO 5
-->210   RANDOM = INT( RND * 10 + 1 )
-->250   IF A(RANDOM) = 0 THEN 210
   260   PRINT RANDOM;
   270   A(RANDOM) = 0 'Value unavailable
   280  NEXT J
```

Program 6-2. Drawing five random numbers from among ten.

```
Random number seed (-32768 to 32767)? 8
 3  8  6  2  10
```

Figure 6-2. Execution of Program 6-2.

From all appearances our program works just fine. But look at lines 210 and 250. If the value selected by line 210 has already been used, then line 250 requires the computer to draw another random value. Inevitably this is a trial-and-error process. It might be interesting to evaluate how well it does work. One measure of the quality of the program will be the number of unusable random numbers generated: the fewer the better. We can easily insert a counting variable to determine this. This is left as an exercise.

Considering the problem set before us, the trial-and-error method of the above program is not really a serious flaw in design. Drawing five numbers from among ten, or even drawing ten from among ten, does not require major computer resources. However, what happens when we increase the numbers? Suppose we want to draw one hundred from among one hundred? When we draw for the last number, we have a one in a hundred chance of getting it. That could take a while. It is worth investing some effort to eliminate the trial-and-error entirely.

Here is a plan that allows us to use every random number selected. First initialize the elements of the array as follows:

```
100  FOR J = 1 TO 10
110   A(J) = J
120  NEXT J
```

This means that each element stores one of the numbers in the range 1 to 10. Next, select a random number in the range 1 to 10 and use that value as the subscript, S. Now display A(S) and replace A(S) with A(10). We have our first random number. Now select a random number in the range 1 to 9. Since we have moved A(10) into a lower-numbered element, we may select from among fewer elements and still include all of the remaining numbers in the next random selection. The second time through we move A(9) into the selected element. We simply repeat the select-display-replace sequence until the desired number of random draws have occurred.

We need to calculate the number of elements remaining. As the draw number (J) goes from 1 to 5, the number of elements remaining goes from

10 to 6. Thus, we can calculate the last element with

```
210   LAST = 10 - J + 1
```

Of course we could just as well use LAST = 11 − J, but the form in line 210 tells us more about where the numbers are coming from. This makes the program easier to read. See Program 6-3.

```
   90   REM ** Random values without replacement
   92   '       and without trial-and-error
  100  FOR J = 1 TO 10
  110   A(J) = J
  120  NEXT J
  196 :
  200  FOR J = 1 TO 5
  210   LAST = 10 - J + 1
  230   S = INT( RND * LAST + 1 )
-->240   PRINT A(S);
-->250   A(S) = A(LAST) 'Move last value
  270  NEXT J
  300  END
```

Program 6-3. Drawing without replacement efficiently.

Notice that the element is printed in line 240 and then replaced by the current LAST element in line 250. LAST is always the number of active elements in the array. Even if we happen to select the LAST element this method continues to function properly. The LAST element will be assigned to itself. No harm done.

```
3  10  9  4  1
```

Figure 6-3. Execution of Program 6-3.

....SUMMARY

So now we have a variable that allows us to include several values in a single variable name. X(J) is the Jth element in the array variable X. We may use subscripts from 0 to 10. We often use arrays to store data values that belong in a group.

Problems for Section 6-1 .....

It is a good idea to simply experiment with arrays to get the feel of how they work. Some of these problems are suggested to provide "fingertip learning."

1. Modify the temperature program (6-1) to determine how many times the temperature increased, decreased, and remained unchanged.
2. Modify the temperature program (6-1) to display the day on which the highest and lowest temperatures occurred.

3. Modify the first random-number drawing program (6-2) to draw ten numbers from among ten.
4. Change the first random-number-drawing program (6-2) to count the number of random values that are duplicates. Run it with several seeds to get a range of values.
5. Enter 3, 5, 6, and 17 in one array and 6, −9, 11, −13, and 3 in another. Display all possible pairs by selecting one element from each array. There are 20 pairs.
6. Fill two arrays as in Problem 5. Fill a third array with all elements in either array with no duplicates. Display the resulting array.
7. Fill an 11-element array with random values. Display the largest value and its position in the array.

6-2...A Simple Sort

Computers do a lot of sorting and arranging of data. Whole books are devoted to sorting and searching. In this section we are going to look at a very simple sort and write a program to implement it. Arrays are ideal for jobs like this. We will load an array with ten numbers and arrange them in increasing order. To do this we will check pairs of elements in the array one pair at a time. If they are in the correct order, we simply go to the next pair. If they are not in the correct order, then we want to exchange them. One way to exchange two values requires an intermediate variable. To exchange A and B we need three BASIC statements as follows:

```
520  TEMP = A
530  A    = B
540  B    = TEMP
```

The variable TEMP is used to save the value of A while we copy the value of B into it. Then the value that we saved in TEMP can be copied into B.

BASIC lets us do that in a special statement.

```
530  SWAP A, B
```

does exactly the same thing. If we check out pairs of numbers that are next to each other in an array named A, the heart of the sort will be the following statement:

```
240   IF A(J) > A(J+1) THEN SWAP A(J), A(J+1)
```

We need a routine to READ the values into the array, a routine to perform the test of line 240 above on all necessary pairs, and a routine to display the results.

The values can be read in with a loop that looks for a special value to signal the end of DATA. Let's use −999999.

The routine that checks all necessary pairs simply scans the array from the beginning to one less than the end, looking at the Jth and J + 1st

as in line 240 above. At the end of each scan, we can look at the one less pair because we have moved the next largest value to the correct location in the array. Once we have put the correct value in the element numbered 2, the process is guaranteed to be complete. It is time to display the result.

```
50   REM **    A simple sort
98   REM ** Load numbers to be sorted in A array
100  N = 0
110  READ X : IF X = -999999! THEN 200
120   N = N + 1
130   A(N) = X
140   GOTO 110
196 :
198  REM ** Here is the sort
200  FOR LAST = N - 1 TO 2 STEP -1
230   FOR J = 1 TO LAST
240    IF A(J) > A(J+1) THEN SWAP A(J), A(J+1)
250   NEXT J
280  NEXT LAST
296 :
298  REM ** Sort complete - display
300  FOR J = 1 TO N
310   PRINT A(J);
320  NEXT J
890  END
896 :
898  REM ** Test data
900  DATA 102, 32, -91, 982, 87
902  DATA 73, 23, -981, 234, 21
990  DATA -999999
```

Program 6-4. A simple sort.

This sort is deceptively easy. It is also very slow. If we have much data to be sorted, we must turn to more sophisticated methods.

```
-981 -91  21  23  32  73  87  102  234  982
```

Figure 6-4. Execution of Program 6-4.

Problems for Section 6-2

1. Notice that the 200 routine of Program 6-4 that does the actual sorting would take the same time for a list that is already in order as for any other. Put in a variable that switches on whenever a SWAP is done. At the end of the inner loop have the program test to see if any SWAP has been done. If no SWAP has been done then the sort is finished. (This will improve execution for some lists, but very little can be done to improve the inherent inefficiency of this type of sort.)
2. Change Program 6-4 to arrange in decreasing order.

6-3...Array Sizes and Shapes (DIM)

Suppose we want to deal with data for the 12 months of the year. We would like to have an array with subscripts up to 12. It is easy with DIM.

```
85  DIM MONTHS(12)
```

does the trick. The DIM or DIMension statement is executed only once to create the desired effect. We may also use DIM to declare smaller arrays. As before, the zero subscript is available, so we really have 13 storage cells in the MONTHS array. For our little problem dealing with the days of the week we could use

```
8Ø  DIM WEEK(7)
```

It is always good programming practice to include every array in a DIM statement. Ideally this statement is among the early statements in the program. This provides important information to anyone reading our program. It is disconcerting to find a statement referring to X(6), or worse yet X(J9), without any clue as to how large the array might be. Even if we want to allow subscripts up to ten, we should state that in a DIM statement. Several arrays may be mentioned in a single DIM statement by separating them with commas.

```
95  DIM WEEK(7), MONTH(12)
```

takes care of two arrays for us. An attempt to DIM an array a second time will evoke the

```
Duplicate Definition
```

error message. Versions of BASIC prior to 5.0 say "Redimensioned array".

....Multiple Dimensions

Suppose we want to work with population figures over a period of years. Let's look at a table of values for Spokane, Washington.

YEAR	POPULATION
1950	161,271
1960	181,608
1970	170,516
1980	171,300

Table 6-1. Population of Spokane, Washington.

With such a small table we could actually do a lot of analysis by eye. However, the principles we learn here may be applied to larger amounts of data. We need an array with four rows and two columns. We can easily provide such an array with

```
100  DIM CENSUS(4,2)
```

Up to 255 dimensions are theoretically possible, but there are numerous practical limits that we will reach long before 255. Three or four dimensions quickly gobble up computer memory. If we ever access CENSUS with a first subscript larger than 4 or second subscript larger than 2, we will get

```
Subscript out of range
```

On the other hand, if a subscript goes negative, we get

```
Illegal function call
```

In the CENSUS situation we are actually providing an extra row and an extra column because of the zero subscripts. In the interest of simplicity, let's not use the zero subscripts. Later, when we are more comfortable with arrays, we can look at this situation in more detail. Sidelight 6 tells us how to eliminate zero subscripts.

So what do we do with this data? We might want to know the year of the largest and smallest population. Or we might want to know about percentage increases and decreases. Perhaps it would be useful to arrange the years in order of population. But first we must get the data into the array. Perhaps the easiest way is to READ DATA. We will be careful to leave the extra commas out when typing the DATA statements for our program. Commas are used to separate DATA items. Once we have the data in our array we can put together a program to provide answers to all of our questions. It will be a good idea to develop the program using subroutines for the various functions. See Program 6-5.

```
100 DIM CENSUS(4,2)
110  GOSUB 800 'Read census data
120  GOSUB 900 'Display census data
190  END
796 :
798  REM ** Read census data
800  FOR ROW = 1 TO 4
810   FOR COLUMN = 1 TO 2
830    READ CENSUS( ROW, COLUMN )
870   NEXT COLUMN
880  NEXT ROW
890  RETURN
896 :
898  REM ** Display census data
900  FOR ROW = 1 TO 4
910   FOR COLUMN = 1 TO 2
930    PRINT CENSUS( ROW, COLUMN ),
970   NEXT COLUMN
975   PRINT
980  NEXT ROW
990  RETURN
```

```
996 :
1000  DATA 1950,161271, 1960,181608
1010  DATA 1970,170516, 1980,171300
```

Program 6-5. Read and display census data.

Note how convenient it will be to have the display code isolated as a subroutine. Later, when we only want to know which year produced the largest census for Spokane, we may simply leave out GOSUB 900. In this way the display routine will be unaffected. We could also include the conventional comma in the population figures with PRINT USING.

```
1950          161271
1960          181608
1970          170516
1980          171300
```

Figure 6-5. Execution of Program 6-5.

Now we are in a position to begin asking questions about the data. Let's find out which census tabulated the greatest population. We need a little routine that scans the array looking for the largest value of CENSUS(J,2). We also need to keep track of the year. We just set YEAR to the first year and set LARGE to the population for the first year. Then we scan the rest of the array to see if any years have a higher population. Program 6-6 lists the relevant changes to Program 6-5.

```
  100  DIM CENSUS(4,2)
  110  GOSUB 800 'Read census data
  120  GOSUB 700 'Find largest population
  190  END
  696 :
  698  REM ** Find largest population
  700  YEAR = CENSUS(1,1) : LARGE = CENSUS(1,2)
  710  FOR J = 2 TO 4
-->720   IF CENSUS( J, 2 ) <= LARGE THEN 770
  730   YEAR  = CENSUS( J, 1 )
  740   LARGE = CENSUS( J, 2 )
  770  NEXT J
  780  PRINT YEAR, LARGE
  790  RETURN
```

Program 6-6. Change Program 6-5 to find largest population.

Look at line 720. We said that if any value in column 2 is greater than the current value of LARGE we will save the new higher value. We say if the value in column 2 is less or equal to the current value of LARGE then proceed directly to the next value in column 2. Otherwise, save the new value of population and the year. We could also code that process with

```
720   IF CENSUS( J, 2 ) > LARGE THEN YEAR  = CENSUS( J, 1 )
                                : LARGE = CENSUS( J, 2 )
```

If we use this version of line 720, we will, of course, eliminate lines 730 and 740. This new line 720 may be easier to read. This program will inform us that Spokane had a population of 181,608 in 1960.

Problems for Section 6-3

Again, it is good to experiment with arrays. DIMension a two-dimensional array and try things. Try three dimensions.

1. Write a program to fill a five-by-seven array with values of your choice. Display the totals column by column. Display the totals row by row.
2. Write a program to fill a five-by-seven array with values of your choice. Display the largest value in each row. Display the largest value in each column.
3. Fill a four-by-eight array with random values in the range from 1 to 100. Display the array. Then multiply each element by −5 and display the result.
4. Draw one hundred numbers from among one hundred using the method of Program 6-2 and Program 6-3. Compare the time required.

6-4...Words, Words, and More Words (String Arrays)

If we were going to store the high temperature for each day of the week we might also want to store the names of the days of the week. We want Sunday, Monday, . . . , Friday, and Saturday as data. This is easy to do with a string array. Let's look at a little program to display the names of the days of the week.

```
90   REM ** Display the days of the week
96  :
100  DIM DAY$(7)
110  FOR J = 1 TO 7
120   READ DAY$(J)
130  NEXT J
196 :
200  FOR J = 1 TO 7
210   PRINT DAY$(J)
220  NEXT J
896 :
900  DATA Sunday, Monday, Tuesday, Wednesday
910  DATA Thursday, Friday, Saturday
```

Program 6-7. Display the days of the week.

Now it is a simple step to combine the ideas of Programs 6-7 and 6-1 to label the results with the day name.

Once we have the day names in a string array some nice things begin to happen. We have the labels available at all times for display. We also have the flexibility of using the full day name where that is important or using abbreviations where there is little space. We may use

```
LEFT$(DAY$(J),3)
```

to display just the first three letters.

String values are listed in DATA statements in the same manner as numeric values. Strings and numerics may be intermixed at will. A string variable may READ a numeric value, but a numeric variable cannot READ a string value. If you try to do that you will get

```
Syntax error in 900
```

where the line number names the DATA statement. Now you know. If it becomes necessary to include a comma or a colon as part of a data item, then surround the entire data item with quotes. Quotes are also required for important leading or trailing spaces.

Problems for Section 6-4

1. Modify Program 6-7 to display only the three-letter day name abbreviation in common use.
2. Write a program that stores the months of the year in a string array and displays them as column headers as follows:

```
J F M A M J J A S O N D
a e a p a u u u e c o e
n b r r y n l g p t v c
```

3. Write a program that stores the months of the year in a string array and displays them as column headers as follows:

```
J F M A M J J A S O N D
 a e a p a u u u e c o e
  n b r r y n l g p t v c
```

6-5...An Alphabet Game

You're driving along and someone says, "Let's play Alphabet." Everybody in the car tries to find every letter of the alphabet in order on signs along the roadway. Whoever gets the "Z" first wins. We can develop a computer program to do a fair job of simulating that game. We'll write a program to play this game for a single player. Then you may want to expand on it. The

program will involve many of the things we have been doing recently. There will be a string array to store the signs. We can use the RND function to select signs. The sign should appear on the screen only briefly to simulate highway driving. During that time the player should have the opportunity to strike a letter key to signify that he or she has spotted the next letter. The computer needs to do some checking and display messages according to the outcome. For now we may enter the signs in DATA statements. Later we may want to use data files. Since much of what we will be doing in this program has to do with single letters, and many signs have both upper- and lowercase letters, we will have an opportunity to work with the ASC function a bit here.

This is a big job we have laid out for ourselves. We can easily trim it down to size by spending a little extra time organizing before we generate any BASIC program statements. Think about the steps in the game in programming terms.

....Load the Signs Array

It can be said that, once we select our route, the stream of signs has been determined. We may easily simulate this by storing a sequence of signs in a string array. By later selecting signs at random, we may offer the game player different "routes." We may arbitrarily DIMension a string array. Let's use 50 for now. We can just think up a few signs and put them in DATA statements. Later we can put in additional DATA statements if we prefer to.

....Establish Game Beginning

It is easy to get a game going. We simply say, "Now I begin with 'A' or 'a'." For simplicity let's begin with a capital A. Since we will be scanning the alphabet, it will be convenient to think in terms of numeric or ASCII codes. The ASCII code for a capital A is easy to find with

```
PRINT ASC("A")
```

We get 65.

....Simulate Random Signs along the Road

Once we have the first letter established all we need is to generate some random signs. Of course, when we are riding in a car the signs come flashing by. So let's flash a sign on the screen and then make it disappear. We need to provide a FOR . . . NEXT loop to delay the sign image on the screen just long enough to be seen. We can clear any screen by coding 24 PRINT statements. This way, the sign moves up the screen. Thus, simulating the way we drive along the highway.

....Did the Player Spot the Next Letter?

Here we create a routine that allows the player to either enter or not a letter of the alphabet. The easiest way to do that is to use the

INKEY$

statement. This is a new statement that reads a single character from the keyboard. BASIC does not display the character entered. If we want to display it, we do that with PRINT. The RETURN key is not required. If no key has been pressed, then the resulting string is null—that is, it has an ASCII value of zero. And program execution continues on, unlike INPUT, which stops to wait for a response at the keyboard no matter what. So what we need here is to "look" at the keyboard and "see" if a letter has been pressed. If a letter has been pressed, let's arbitrarily convert it to uppercase. This is an ideal place for the case converting functions from Section 5-4. If nothing is pressed, then no action is required. If any key has been pressed, then this routine should determine that the player, in fact, pressed the next letter in the alphabet. If no key has been pressed then the sign selecting routine should be repeated.

....Is the Next Letter Really on the Sign?

Having displayed a sign and received the next letter in alphabetic sequence the program should check to be sure that the letter is on the sign. Here again, we need to deal with uppercase and lowercase. If the current letter is on the sign, and the player has not completed the alphabet, then we repeat the sign-generating routine. Otherwise we move on to terminate the game.

Each of the processes described here is an ideal candidate for a subroutine. This means that we have partitioned the complex problem into manageable tasks. We may concentrate on each smaller task and do the job efficiently. The benefits don't stop there, however. Once the program has been written and it performs to our satisfaction, we may easily make important modifications by concentrating on a single subroutine rather than poring through one long stream of code. It is going to be a simple job to convert this program so that it stores the signs in a data file on disk. Then we can request each new player to enter a favorite sign. Thus the game will become more and more interesting as more people play it.

To summarize then, here is a description of the final program.

1. Load the signs array
2. Establish game beginning
3. Simulate random signs along the road
4. Did the player spot the next letter? If not then repeat step 3
5. Is the next letter really on the sign? If not "Z" yet repeat step 3 or else wind up this game

We will code each of the numbered steps as a subroutine. This summary will serve as the control routine. By retaining each line of the summary as a remark in the program itself we can provide good documentation without further effort. All that remains is for us to make a few decisions about line numbers and variables, and the control routine is complete. Let's just

place the subroutines at 1000, 2000, and so on. Now for the variables. Let's use SIGNS$ as the signs array, N as the total number of signs, ALPHA1 as the ASCII value of the current letter, CAPA as the ASCII value of the letter entered at the keyboard during the play of the game (remember it will be zero if no key is pressed), and R as the randomly selected position of the current sign in the SIGNS$ array. So SIGNS$(R) contains the current sign. We won't forget to DIMension SIGNS$ and include the DEF for converting to uppercase. All of this transforms easily into the BASIC code for a routine to control our game program. See Program 6-8a.

```
100  DIM SIGNS$(50)
102  DEF FNU$(A$) =
          CHR$(ASC(A$) + 32 * (ASC(A$)>96 AND ASC(A$)<123))
110  GOSUB 1000 'Load the signs array
120  GOSUB 2000 'Establish game beginning
130  GOSUB 3000 'Simulate random signs along the road
140  GOSUB 4000 'Did the player spot the next letter?
145    IF CAPA = 0 THEN 130 'If not then repeat step 3
150  GOSUB 5000 'Is the next letter really on the sign?
155    IF ALPHA1 < 91 THEN 130 'If not "Z" yet repeat step 3
160  PRINT "Congratulations, you have made it through the alphabet"
190  END
```

Program 6-8a. Control routine to play Alphabet.

Now all we have to do is write each of the subroutines. The program will practically write itself. Loading the SIGNS$ array consists of READing DATA. We provide for a counter and a final data value as a signal that all data has been read. The number of signs read here is returned in "N".

```
 998   REM ** Load the signs array
1000   N = 0
1010   READ A$ : IF A$ = "Done" THEN 1080
1020    N = N + 1
1030    SIGNS$(N) = A$
1040    GOTO 1010
1080   PRINT "There are:"; N; "signs in this game."
-->1085 GOSUB 1100
1090   RETURN
```

Program 6-8b. Load the Alphabet game road signs.

Note the GOSUB 1100 in line 1085. We need a delay so that the player has time to read the message in the program. We chose to do this in a subroutine at line 1100. We can add the DATA statements at any time.

Now let's establish the game beginning. This simply consists of initializing ALPHA1 to the ASCII value for capital "A".

```
1998  REM ** Establish game beginning
2000  ALPHA1 = 65 'Get ready to look for 'A'
2090  RETURN
```

Program 6-8c. Start with capital "A".

To simulate the signs along the roadside we need to generate random values from one to the number of signs in the array. Next we display the sign and provide a delay. Since part of the idea of this game is to have the signs go flying past, we use a separate delay here. This one should be shorter than the one we use for displaying messages. Finally we PRINT 24 blank lines and leave this subroutine. After you see this program RUN you might want to make some changes. You might vary the length of time the signs stays on the screen from one sign to the next. This could be done with the RND function. You might want to display the sign at different places on the screen. The length of time a sign is on the screen might vary with the length of the sign. With tasks handled as subroutines, the resulting program will be easy to fine-tune.

```
2998  REM ** Simulate random signs along the road
3000  R = INT( RND * N + 1 )
3020  PRINT SIGNS$(R)
3030  FOR J = 1 TO 800 : NEXT J
3040  FOR J = 1 TO 24 : PRINT : NEXT J
3090  RETURN
```

Program 6-8d. Display a sign.

Next we process the player keyboard input. As we said a little earlier, we will use INKEY$. Since the INKEY$ function does not display keyboard input we will include a PRINT statement to do so. It is important to know that the ASC function cannot handle a null string—that is, one containing no characters. When the INKEY$ function finds that no key has been pressed, it returns a null string. Therefore we will have to test this condition separately. If we find that a letter that has been keyed in is not the next one in the alphabet, we need to display a message and keep it on the screen long enough for the player to read. Here is another use for the subroutine at line 1100.

```
3998  REM ** Did the player spot the next letter?
4000  A$ = INKEY$
4005  IF LEN(A$) = 0 THEN CAPA = 0 : GOTO 4090
4010   PRINT A$; " ";
4020   A$ = FNU$(A$) : CAPA = ASC(A$)
4030   IF A$ < "A" OR A$ > "Z" THEN 4000
4050   IF CAPA = ALPHA1 THEN 4090
4060    PRINT "Not the next letter in the alphabet" : GOSUB 1100
4070    GOTO 4000
4090  RETURN
```

Program 6-8e. Check keyboard input.

To check if a letter is on a sign we need to check both upper- and lowercase. We need a message for "not found" and one for "found". Here is yet another use for the delay routine at line 1100. If the letter is found we need to increment ALPHA1 to move to the next letter in the alphabet.

```
4998  REM ** Is the next letter really on the sign?
5000  FOR J = 1 TO LEN(SIGNS$(R))
5010   B$ = FNU$(MID$(SIGNS$(R),J,1))
5020   IF A$ = B$ THEN 5050
5030  NEXT J
5040  PRINT "Your letter is not on the sign" : GOSUB 1100
5045  GOTO 5090
5050  PRINT "Good" : GOSUB 1100
5060  ALPHA1 = ALPHA1 + 1
5090  RETURN
```

Program 6-8f. Check if a letter is on a sign.

And now we come to the delay routine. It is simply a FOR . . . NEXT loop that does nothing. Here we have set the upper limit at 1500. You might want to change that to suit your own taste.

```
1098  REM ** Time delay for messages
1100  FOR J = 1 TO 1500 : NEXT J
1190  RETURN
```

Program 6-8g. Time-delay routine.

Finally, we have included a few signs for data.

```
1498  REM ** The signs
1500  DATA Stop, Al's Pizza, Dairy Queen, Burger King
1502  DATA Yield, One Way, This Way Out, Detour
1504  DATA One Show Only Tonight, Exit Only, Entrance Only Please
1506  DATA Florida 2138 mi., Fly United, Jet Set Diner
1508  DATA Give Her a Valentine, Give Him a Valentine
1510  DATA First Avenue, North Side
1598  DATA Done
```

Program 6-8h. Data for the Alphabet game.

Here is the complete Alphabet game program.

```
100  DIM SIGNS$(50)
102  DEF FNU$(A$) =
          CHR$(ASC(A$) + 32 * (ASC(A$)>96 AND ASC(A$)<123))
110  GOSUB 1000 'Load the signs array
120  GOSUB 2000 'Establish game beginning
130  GOSUB 3000 'Simulate random signs along the road
140  GOSUB 4000 'Did the player spot the next letter?
145    IF CAPA = 0 THEN 130 'If not then repeat step 3
150  GOSUB 5000 'Is the next letter really on the sign?
155    IF ALPHA1 < 91 THEN 130 'If not "Z" yet repeat step 3
160  PRINT "Congratulations, you have made it through the alphabet"
190  END
996 :
998   REM ** Load the signs array
1000  N = 0
1010  READ A$ : IF A$ = "Done" THEN 1080
1020   N = N + 1
1030   SIGNS$(N) = A$
```

```
1040  GOTO 1010
1080  PRINT "There are:"; N; "signs in this game."
1085  GOSUB 1100
1090  RETURN
1096 :
1098  REM ** Time delay for messages
1100  FOR J = 1 TO 1500 : NEXT J
1190  RETURN
1496 :
1498  REM ** The signs
1500  DATA Stop, Al's Pizza, Dairy Queen, Burger King
1502  DATA Yield, One Way, This Way Out, Detour
1504  DATA One Show Only Tonight, Exit Only, Entrance Only Please
1506  DATA Florida 2138 mi., Fly United, Jet Set Diner
1508  DATA Give Her a Valentine, Give Him a Valentine
1510  DATA First Avenue, North Side
1598  DATA Done
1996 :
1998  REM ** Establish game beginning
2000  ALPHA1 = 65 'Get ready to look for 'A'
2090  RETURN
2996 :
2998  REM ** Simulate random signs along the road
3000  R = INT( RND * N + 1 )
3020  PRINT SIGNS$(R)
3030  FOR J = 1 TO 800 : NEXT J
3040  FOR J = 1 TO 24 : PRINT : NEXT J
3090  RETURN
3996 :
3998  REM ** Did the player spot the next letter?
4000  A$ = INKEY$
4005  IF LEN(A$) = 0 THEN CAPA = 0 : GOTO 4090
4010   PRINT A$; " ";
4020   A$ = FNU$(A$) : CAPA = ASC(A$)
4030   IF A$ < "A" OR A$ > "Z" THEN 4000
4050   IF CAPA = ALPHA1 THEN 4090
4060    PRINT "Not the next letter in the alphabet" : GOSUB 1100
4070    GOTO 4000
4090  RETURN
4996 :
4998  REM ** Is the next letter really on the sign?
5000  FOR J = 1 TO LEN(SIGNS$(R))
5010   B$ = FNU$(MID$(SIGNS$(R),J,1))
5020   IF A$ = B$ THEN 5050
5030  NEXT J
5040  PRINT "Your letter is not on the sign" : GOSUB 1100
5045  GOTO 5090
5050  PRINT "Good" : GOSUB 1100
5060  ALPHA1 = ALPHA1 + 1
5090  RETURN
```

Program 6-8. The Alphabet game.

Now, there are lots of things that you could do to improve this program. You could convert it to accommodate several players. You could

make changes so that the program prods the player for missing signs that have the next letter on them. The program could easily be made to go faster or slower and to change speed at random. It could stop at a traffic light or move onto an Interstate. You could do lots of things with graphics. The signs could be flashed at random about the screen. They could be made to flash on and off. We have only scratched the surface.

Problems for Section 6-5 .....

Problems 1 through 7 refer to the Alphabet game.

1. Write a little routine to seed the random-number generator. Request the player's name. Then use the length of the name.
2. Write a little program to seed the random-number generator. Request a name. Sum up the ASCII values of the characters of the name. Use the sum as the seed. Using this scheme Ann and Bill will produce different random sequences.
3. Change the Alphabet game to simulate changing speed in the car. Stop at a traffic light. Move onto the Interstate.
4. Make the time a sign stays on the screen proportional to the length of the sign.
5. Give the program the ability to make signs appear with differing probability. "Yield" could appear often, but "Rudy's Diner" should appear only once in any one game (unless we get lost).
6. Write a program to tabulate the frequency of occurrence of the letters in the signs. This information could be used to decide on additional signs to include in the DATA.
7. Arrange the results in problem 6 in order of frequency of occurrence.
8. Write a program to play Geography. In this game two or more players take turns thinking of place names. Each player must name a place whose first letter matches the last letter of the previous player's place. Have the program add new place names to the array and offer to play additional games. Make the computer a player in a two-player game. Names may not be repeated in any one game.

SIDELIGHT 6

Array Goodies

....OPTION BASE

BASIC automatically allows for zero subscripts. Programmers may ignore the zero subscript if that is desirable. Sometimes zero subscripts are just

the ticket, and sometimes we don't need them. We are not required to use zero subscripts. Usually this is not an important consideration. But when we work with large arrays, the amount of memory taken up by all those unused zero elements can be a real problem.

Suppose we look at an array DIMensioned ten by ten by ten. Such an array contains 1,331 elements. Of those, 331 are referenced by at least one zero subscript. If we have no logical use for them, we can save all that space with

```
OPTION BASE 1
```

This statement eliminates all elements with a zero subscript. So for our ten-by-ten-by-ten array we need only enough memory for 1,000 elements. That is quite a saving.

....Variable DIM

Suppose we have a program in which the dimensions of our arrays might change depending on data handled during execution. We can provide for all situations by using a variable DIM statement. We may use a program segment such as the following:

```
120  INPUT "          Number of weeks:"; WEEKS
130  INPUT "Number of values per week:"; VALUES
140  DIM MATRIX( WEEKS, VALUES )
```

If we call for too large an array BASIC will deliver the

```
Out of memory in 140
```

error. If we try to dimension the same array again, regardless of the amount of memory required, we will see

```
Duplicate Definition in 140
```

Older versions of BASIC will report "Redimensioned array in 140". There is a way around the "Duplicate Definition" problem.

....ERASE

We can eliminate an array and recover the memory it occupies with the ERASE statement.

```
945  ERASE TEST, TEST1
```

does this for arrays TEST and TEST1. This frees us to redimension any array mentioned in the ERASE statement. Or we might just want the space for some other purpose. An attempt to ERASE a nonexistent array brings forth

```
Illegal function call
```

....Variable Typing and Memory

If we have a situation in which we need a very large array, and values in the range −32768 to 32767 are sufficient, then we have the option of declaring our array as an integer array just as we did for simple variables.

```
110  DIM ARRAY%(400,5)
```

gives us 2000 or 2406 values depending on our use of zero subscripts. Integers require two bytes in memory, while our conventional six-digit precision values require four.

On the other hand, we might require up to 16 digit precision. In this case, we declare a double precision array.

```
100  DIM NUMBER#(10,20)
```

does the job. Of course, now our numbers occupy eight bytes each.

In a program where several numeric data types are in use, we might want to explicitly declare single precision with a statement such as

```
90   DIM TIDBIT!(12,31,2)
```

Such a move will serve to more clearly document what is going on in your program.

Chapter 7

Miscellaneous Applications

7-1...A Calendar Program

Let's write a program to display one month of a calendar given the month and year. The workings of our calendar are well known. The days of the week have a seven-day rotation. The number of days in each month is fixed. The calendar follows a strange pattern, but it is a fixed pattern. The four-year rotation for leap year is clear. If we limit ourselves to the twentieth century, we don't have to worry about the 400-year cycle. It is easy to develop this program by going from the big tasks down to the smaller ones.

Given the month and year, we are going to produce the calendar display of Figure 7-1.

```
      Dec        1929

Sun Mon Tue Wed Thu Fri Sat

  1   2   3   4   5   6   7
  8   9  10  11  12  13  14
 15  16  17  18  19  20  21
 22  23  24  25  26  27  28
 29  30  31
```

Figure 7-1. One page of a calendar.

The control routine has only two segments: request data from the keyboard and display the calendar for that month. We start right in with Program 7-1a.

```
20   GOSUB 100 'Get Month, Year
30   GOSUB 200 'Display calendar
90   END
```

Program 7-1a. Control the calendar.

We want a number from 1 to 12 for the month and from 0 to 99 for the year. Let's make sure that values entered are within that range. We might just as well ensure that the values passed to the main program are integers, while we are at it. This is easy to do with the INT( ) function. See Program 7-1b.

```
96  :
98   REM ** Get Month, Year
100  INPUT "Month, Year"; MONTH, YEAR
130   MONTH = INT(MONTH) : YEAR = INT(YEAR)
140   IF MONTH < 1 OR MONTH > 12 THEN 100
160   IF YEAR  < 0 OR YEAR  > 99 THEN 100
190  RETURN
```

Program 7-1b. The INPUT subroutine.

WOW, we are already half done—but not quite. We need to break up the calendar display into several smaller tasks. A few calculations are needed and we might split the display itself into two parts. Let's display the title and calendar itself separately. That sounds like more subroutines. We just keep on breaking the task into manageable pieces. Study Program 7-1c.

```
196 :
198  REM ** Display calendar
200  GOSUB 300 'Calculate
240  GOSUB 400 'Display title
260  GOSUB 500 'Display days
290  RETURN
```

Program 7-1c. Control printing the calendar.

The subroutine at line 300 will make all the necessary calculations from the month and year entered at the keyboard. As long as we keep the month and year variables intact, the subroutine at 400 can easily display the title. In order to display the familiar number grid for a month, we need to know the number of days in the month and the day of the week for the first day. Given those two things, the subroutine at line 500 can do its job.

We need to tackle the calculations now. The leap year calculation is modular in nature. If the year number is divisible by 4 then it is a leap year (ignoring the 400-year cycle here). If YEAR MOD 4 is 0, then we have a leap year. Remember MOD from Chapter 2?

```
300  LEAP = 0 : IF YEAR MOD 4 = 0 THEN LEAP = 1
```

Using line 300, LEAP is 1 for leap year and 0 if not.

The days follow a seven-day cycle. This is another modular process. Let's assign 0 to 6 to the day names Sunday through Saturday. If we can just work out the day name for January 1 of any year, then we will be able to work our way through the months. If we know what day of the week January 1 of any year falls on, we can calculate all the rest. For 365-day years, the day of the week of Jan. 1 advances one day from year to year. Leap years advance one extra day. It turns out that Jan. 1, 1900, fell on a Sunday. So the day of the week of Jan. 1 is the YEAR number adjusted for leap years taken MOD 7. The leap year adjustment should add an extra day every four years beginning with 1901. Therefore, we add 3 to the year number before dividing by 4. Remember, we are accepting year numbers in the range 0 to 99.

```
310  DAY = (YEAR + INT((YEAR + 3) / 4)) MOD 7
```

Upon execution of line 310, DAY will be in the range 0 to 6 for the day of the week of Jan. 1 of the current year.

Now we want to know the day name for the first day of the current month. For this we need to know the pattern governing the number of days in each month. Here it is:

```
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
  1   2   3   4   5   6   7   8   9  10  11  12
 31  28  31  30  31  30  31  31  30  31  30  31
```

See that the alternation changes after July? The number of days in a month is 30 plus 1 for odd-numbered months up to the seventh month. For the later months, we add 1 for the even-numbered months. This is a perfect place for another MOD calculation. M MOD 2 is 1 for odd-numbered months and 0 for even-numbered months. If we just add 1 for M > 7 we get back on the track. (Subtracting 1 is fine, too.) In BASIC, M > 7 evaluates to −1, but it doesn't matter whether we add or subtract. All this is done to assign the number of days in the current month to the variable N in the following statement:

```
350   N = 30 + ((M + (M > 7)) MOD 2)
```

We need to add this to the day-of-the-week value for the next month as we cycle through the months of the current year to get to the requested month. Note that for January, the value for N must be 0 as we begin to cycle through the months of the year, since there is no previous month. And finally, when we come to February, as we cycle through the months of the requested year, we must set N to either 28 or 29 as this is a leap year or not.

```
360   IF M = 2 THEN N = 28 + LEAP
```

All this goes to make up the calculations of Program 7-1d.

```
292 :
294  REM ** Calculate
296  '      DAY - weekDAY, 1st of Month
298  '      N - No. of days in Month
300  LEAP = 0 : IF YEAR MOD 4 = 0 THEN LEAP = 1
310  DAY = (YEAR + INT((YEAR + 3) / 4)) MOD 7
320  N = 0
330  FOR M = 1 TO MONTH
340   DAY = (DAY + N) MOD 7
350   N = 30 + ((M + (M > 7)) MOD 2)
360   IF M = 2 THEN N = 28 + LEAP
380  NEXT M
390  RETURN
```

Program 7-1d. Calendar calculations.

The calendar title at line 400 comes next. We simply set up a string with all the month names and PRINT the three characters corresponding to the month selected. See Program 7-1e.

```
396 :
398  REM ** Display calendar title
400  M$ = "JanFebMarAprMayJunJulAugSepOctNovDec"
410  PRINT
430  PRINT TAB(8); MID$(M$, MONTH*3-2, 3), 1900 + YEAR
440  PRINT : PRINT "Sun Mon Tue Wed Thu Fri Sat" : PRINT
490  RETURN
```

Program 7-1e. Display calendar title.

Finally, we need to work out the display of the number grid for the month. We are going to display 28, 29, 30, or 31 days as determined by the value of N. That is easy with a FOR loop running from 1 to N. The first day of the month needs to be positioned under the corresponding day name as displayed by Program 7-1e. The program needs to output a blank PRINT whenever it has just displayed a Saturday. See Program 7-1f.

```
496 :
498  REM ** Display calendar days
500  K = DAY
530  FOR J = 1 TO N
-->540   PRINT TAB(K*4 + 1 - (J < 10)); J;
550   K = (K + 1) MOD 7 : IF K = 0 THEN PRINT
570  NEXT J
580  PRINT : PRINT
590  RETURN
```

Program 7-1f. Display calendar days.

There is just one thing in line 540 that bears further comment. We are using TAB( ) to position the display in the correct column. To make it come out right, we are subtracting (J < 10) there. Since the expression

(J < 10) is −1 when true, we are TABing an extra space for one digit values of J. Thus the subtraction. This lines up the columns nicely for us.

This program has been designed to be flexible. The control routines of Program 7-1a and 7-1c can be changed to achieve different goals. By breaking up the task into small subroutines, we can easily use them unchanged for other purposes. The problems for this section are intended to clearly demonstrate this concept.

Now we list the program in its entirety.

```
20   GOSUB 100 'Get Month, Year
30   GOSUB 200 'Display calendar
90   END
96  :
98   REM ** Get Month, Year
100  INPUT "Month, Year"; MONTH, YEAR
130   MONTH = INT(MONTH) : YEAR = INT(YEAR)
140   IF MONTH < 1 OR MONTH > 12 THEN 100
160   IF YEAR  < 0 OR YEAR  > 99 THEN 100
190  RETURN
196 :
198  REM ** Display calendar
200  GOSUB 300 'Calculate
240  GOSUB 400 'Display title
260  GOSUB 500 'Display days
290  RETURN
292 :
294  REM ** Calculate
296  '     DAY - weekDAY, 1st of Month
298  '     N - No. of days in Month
300  LEAP = 0 : IF YEAR MOD 4 = 0 THEN LEAP = 1
310  DAY = (YEAR + INT((YEAR + 3) / 4)) MOD 7
320  N = 0
330  FOR M = 1 TO MONTH
340   DAY = (DAY + N) MOD 7
350   N = 30 + ((M + (M > 7)) MOD 2)
360   IF M = 2 THEN N = 28 + LEAP
380  NEXT M
390  RETURN
396 :
398  REM ** Display calendar title
400  M$ = "JanFebMarAprMayJunJulAugSepOctNovDec"
410  PRINT
430  PRINT TAB(8); MID$(M$, MONTH*3-2, 3), 1900 + YEAR
440  PRINT : PRINT "Sun Mon Tue Wed Thu Fri Sat" : PRINT
490  RETURN
496 :
498  REM ** Display calendar days
500  K = DAY
530  FOR J = 1 TO N
540   PRINT TAB(K*4 + 1 - (J < 10)); J;
550   K = (K + 1) MOD 7 : IF K = 0 THEN PRINT
570  NEXT J
580  PRINT : PRINT
590  RETURN
```

Program 7-1. The calendar program.

Problems for Section 7-1 .....

1. Modify Program 7-1 so that it displays the calendar for an entire year.
2. Modify Program 7-1 so that it displays all calendars for a given month for a range of years.
3. Modify Program 7-1 to display the month and year for every month that has a Friday the thirteenth. Note: if Friday falls on the thirteenth, then what day of the week does the first of the month fall on?
4. Modify Program 7-1 to request a date in the form YYMMDD and display the day of the week for that date.
5. With the year 2000 close at hand, modify Program 7-1 to handle the twenty-first century.
6. Write a subroutine to verify keyboard input for being a valid date. Accept numbers in the form YYMMDD. Get right down to February and leap years.

7-2...The Sieve of Eratosthenes

Eratosthenes, who lived around 240 B.C., worked out a way of detecting prime numbers by eliminating all composite numbers. The steps used are: Write down all the integers. Now, beginning with 2, cross out all multiples of 2 up to the upper limit. Go back to the next un-crossed-out integer and cross out all multiples of it. Repeat this until there are no more numbers to cross out. The remaining numbers are primes.

This is nicely implemented on the computer using an array to "write down" the integers we want. Set all array values equal to 1. Then begin with 2, leave it, and proceed by crossing out 4, 6, and so on. Next go back to the 3, leave it, and begin by crossing out 6, 9, and so on. The crossing-out process may be simulated by setting the array value equal to 0.

This method is surprisingly easy to program. We first try this with just 100 elements in our array. If we start out with thousands and we make a programming error, it may be several minutes before the results are displayed. Let's perfect it with the smaller task. After we are sure that it works then we can try larger and larger values until we use all of memory. See Program 7-2.

```
100  PRINT "Finding primes using the Sieve of Eratosthenes"
110  PRINT
120  UPPER.LIMIT = 100
130  DIM SIEVE(UPPER.LIMIT)
196 :
198  REM ** Load the array with 1's
200  FOR J = 1 TO UPPER.LIMIT
```

```
  210   SIEVE(J) = 1
  220  NEXT J
  296 :
  298  REM ** "Work the sieve"
-->300  FOR J1 = 2 TO UPPER.LIMIT
  320   FOR J2 = 2*J1 TO UPPER.LIMIT STEP J1
  340    SIEVE( J2 ) = 0
  360   NEXT J2
  380  NEXT J1
  396 :
  398  REM ** Display primes only
  400  FOR J1 = 2 TO UPPER.LIMIT
  420   IF SIEVE( J1 ) = 1 THEN PRINT J1;
  440  NEXT J1
```

Program 7-2. Primes using the sieve of Eratosthenes.

Since we may want to run this program with a large value for UPPER.LIMIT, it makes sense to consider efficiency in this program.

Look at line 300. There is no need to have J1 go past the square root of the UPPER.LIMIT. Any larger values that must be crossed out already have been.

Think about what happens in the line 300 routine when J1 gets to 4. We might go to 8, 16, and so on. But any multiple of 4 is also a multiple of 2. And all multiples of 2 have already been crossed out, including 4. Therefore, we do not need to cross out values if the first element we come to already has been. We can carry out this test with an IF statement. The combination of these two things will result in a saving of execution time.

If we want a very large number of primes, then we can learn about random-access files and come back to this problem. We could store each integer in one record of a file and think of the file as one giant array on disk.

Problems for Section 7-2

1. Set the upper limit on line 120 of Program 7-2 at 1000 and RUN the program. Change the upper limit in line 300 to SQR(UPPER. LIMIT) and insert a line 310 to test if SIEVE(J1) has already been crossed out. RUN the new version. Compare the execution time for the two versions.
2. We are limited by available memory in Program 7-2. As written, the program uses 4 bytes for each integer in array SIEVE. First increase UPPER.LIMIT to the maximum possible size, then change the program to use an integer array SIEVE%. You should be able to nearly double the array size.
3. BASIC handles integers faster than single-precision numbers. Speed up Program 7-2 by declaring all variables as integer with

```
10  DEFINT A-Z
```

Compare execution time for single-precision and integer modes.

7-3...Number Bases

....Binary Numbering

Computers are not very good at reckoning in our familiar base-ten number system. Computers digest everything they do in terms of electrical states. Things are either in a charged state or in an uncharged state. With just two states, it makes sense to represent them with 1 and 0. This leads us to the binary, or base-two, numbering system. The computer doesn't work with different numbers, it just represents them differently. We are very used to working with the decimal, or base-ten, numbering system. In base ten, each place represents a power of ten. In binary, each place represents a power of two. When we write 10 in decimal, we mean ten. When we write 10 in binary, we mean two. To write ten in binary, we use 1010. And 1010 is $2^3 + 2^1$. In base ten, 1010 is one thousand ten.

Binary arithmetic in base two is very easy. For addition, the result of adding two digits is either 0, 1, or 10.

```
0 + 0 =  0
1 + 0 =  1
1 + 1 = 10
```

When adding 1 and 1 there will be a carry into the next place to the left.

Multiplication is also straightforward. When multiplying by 1 the digits shift according to the position of the 1, and when multiplying by 0 the result is 0. The shift for 1 in the first column on the right is 0 places, for 1 in the second column it is 1 place, and so on. Numbering the columns beginning with 0 makes the shift equal to the column number. This agrees nicely with making the column number the exponent on two represented by the digit in that column. Some examples are:

```
    1 * 101=     101    (shift of 0)
   10 * 101=    1010    (shift of 1)
 1000 * 101= 101000    (shift of 3)
```

To multiply two by two in binary looks like this:

```
  10
* 10
-----
 100
```

And to multiply 27 by 5 we would write the following:

```
   11011
 *   101
-------
   11011
  00000
 11011
--------
10000111
```

Note that the carry from adding 1 and 1 in column 3 created a "domino effect" by pushing the carry across several columns.

In any number system each digit of any integer represents an integer power of the base. So the digits in base two represent 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and so on in base ten corresponding to the bit positions 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and so on in binary.

One disadvantage of the binary number system is that it takes so many digits to represent numbers. For instance, 15 base ten is written as 1111 in binary, and 127 base ten is written 1111111 in binary. Only humans notice how cumbersome this is. Computers are well suited to accessing individual digits to determine their state, or to change their state.

A Binary digIT is called a BIT. Since a single bit isn't always useful, it makes sense to group them into packages. One byte is 8 bits. This works out very nicely to accommodate base-ten values in the range 0 to 255.

In BASIC the largest true integer value is 32767, and the smallest is −32768. That comes out to 65536 numbers. Zero base ten is 0 in binary. And 65535 base ten is 1111111111111111 in binary notation. That is 16 bits. We get 16 bits by grouping two bytes together. But, in practice, that number range is utilized as numbers in the range -32768 to 32767. Values from 0 to 32767 are stored as we would expect. Values from 32768 to 65535 are translated into values in the range −1 to −32768. The 16th bit is used to determine the sign of the number. We'll come back to this later.

It is instructive to write a program to convert base-ten numbers to binary notation. A very easy way to start is to use MOD to decide whether the given base-ten number is odd or even. If it is odd, then the units digit in the binary number is a 1. If it is even, then the units binary digit is 0. Next, we "peel off" the binary digit by dividing by 2 using integer division and do the odd/even test on the result. We repeat this until the result is zero.

```
   100  PRINT "Convert base ten numbers to binary format"
   110  PRINT
   196 :
   200  INPUT "Enter a value"; DECIMAL
   210  X = DECIMAL MOD 2
-->220  A$ = STR$(X) + A$
   230  DECIMAL = DECIMAL \ 2
```

```
240  IF DECIMAL THEN 210
250  PRINT A$
```

Program 7-3. Convert base ten to binary.

MOD and integer division are limited to integers. Values entered outside the range −32768 to 32767 will bring forth the following message:

```
Overflow in 210
```

Let's run Program 7-3.

```
Convert base ten numbers to binary format

Enter a value? 8466
 1 0 0 0 0 1 0 0 0 1 0 0 1 0
```

Figure 7-2. Execution of Program 7-3.

Program 7-3 does not handle negative numbers. BASIC uses "twos-complement" form to store negative integers in the range −1 to −32768. Once we have the binary form of the absolute value of our negative number, the rule to get twos complement is: change every 0 to a 1, change every 1 to a 0, and add 1 to the result. Let's look at an example. Running Program 7-3 for 32000 gives us

0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0

According to the rule we change 1's to 0's, 0's to 1's, and add 1, thus:

```
1 0 0 0 0 0 1 0 1 1 1 1 1 1 1 1
                            + 1
-----------------------------------
1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
```

This last 16-bit binary display represents −32000.

....Hexadecimal Numbering

It doesn't take long working with binary numbers for us to wish there could be some shorthand. Hexadecimal numbering comes to the rescue. This new system has 16 digits. Since our familiar base-ten system has only 10 digits, it is necessary to invent 6 new digits. The characters A through F have been selected. Thus, the hexadecimal digits are

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F

Now each place represents an integer power of 16. Here are some example values:

11	hex	17	base ten	10001	binary
1A	hex	26	base ten	11010	binary
4FFF	hex	16383	base ten	11111111111111	binary
8FFF	hex	32767	base ten	111111111111111	binary
FFFE	hex	65534	base ten	1111110000010110	binary

Clearly the hex form is very compact. It turns out that hex values from 0 to FF exactly correspond to base-ten values from 0 to 255. And that exactly corresponds to the range of values available in one 8-bit byte.

Fortunately, BASIC handles hex numbers with ease. If we want to know the base-ten equivalent for 4F, we just

```
PRINT &H4F
```

and quick as a flash we get 79. We may express hex numbers in BASIC by using the &H prefix. This is very handy for utilizing information we might find in magazine articles (or anywhere else for that matter) in which the author includes values in hex. One of the primary uses for hex numbering is for addresses in memory. Suppose we want the base-ten representation for FEFE. BASIC replies with −258. This assumes that we are interested in strict integer values in the range −32768 to 32767. We can get a positive result by adding 65536. We get 65278. Values greater than FFFF cause an overflow.

Converting values the other way is just as easy. The HEX$ function is the way to go. To find a hex value for 65535 we simply

```
PRINT HEX$(65535)
```

BASIC unselfishly reports FFFF. Trying to get a hex value for numbers greater than 65535 will evoke the

```
Overflow
```

error message.

....Octal Numbering

For completeness here, we introduce the OCT$ function and the &0 prefix for octal values. The base-eight system is referred to as OCTAL. The digits run from 0 through 7. The use of OCT$ and &O parallels, in every way, the use of HEX$ and &H. The largest octal value allowed in BASIC is 177777, which is equivalent to 65535 in base ten.

Problems for Section 7-3 .....

1. Program 7-3 displays the results with the binary digits separated. Fix this.
2. Program 7-3 uses the convenience of the integer MOD operator and integer division. We might like to see a binary representation of integers greater than 32767. Use ordinary division and the INT function to do this.
3. Write a program to convert binary numbers to base-ten form. Ac-

cept binary values from the keyboard into a string and process into a BASIC numeric variable.

4. Rewrite Program 7-3 to displays twos complement for integers in the range −1 to −32768.

Chapter 8

Files

For all but the most elementary of computer uses we require some mechanism for saving the results of our work for future use. Most computer work is done on a disk-based system. In recent years there has been an explosion in the use of the floppy disk. The appearance of the floppy disk has enabled schools, businesses, and individuals to use computers for a very low cost. For a little more money it is possible to dramatically increase storage capacity by obtaining a "hard disk." Very large computers have used disks for many years.

It is the disk system that allows us to begin a project and save it using the computer itself. Disks are used to save computer data just as cassette tapes are used to save sounds. We save our BASIC programs on disk with the SAVE command. We later retrieve our programs with the LOAD command, as discussed in Appendix C. All data saved on a disk is organized into "disk files." MBASIC.COM is a disk file. Even our programs saved on disk are called files. It happens that some files are programs that execute by themselves, some files can be executed by using BASIC first, and some files are useful only as data. These last are usually called data files. In the absence of data files we have been saving data in DATA statements of programs. Such data is hard to edit and not practical to update during program execution. With data files we can write a program to manage the data of the file itself and use other programs to obtain information from the file as needed. We can use data from several data files to prepare reports of all kinds.

8-1...Introduction to Data Files

A data file is just some area of the disk where we may save data. The applications for data files are truly unlimited. We can maintain inventory

information, payroll, all kinds of financial information, production records, personnel records, and an endless array of business-related facts and figures. We use data files for word processing, such as writing letters or writing books (this one is an example). Legal documents of all kinds are now prepared using word processing.

In this chapter we will write a few programs that will serve as an introduction to data files. We will use relatively small files. The theory and practice of very large files are beyond the scope of this book. As files are required to contain thousands and even millions of data items it becomes necessary to develop special techniques for organizing the data and finding it later.

BASIC data files come in two distinct varieties: sequential access and random access. Data stored in sequential files is similar to data stored in DATA statements of a program. To get to any particular data entry we must READ all other data entries that precede the one we want. Random-access files are organized into segments that allow us to read any single data entry anywhere in the file directly.

....Sequential-Access Files

Sequential-access files are relatively easy to work with from a programming viewpoint. We simply learn a few new BASIC keywords and think about the continuous stream of data in the file. The catch is that sequential files tend to produce slow-running programs. To make a single change in a data entry of a sequential-access file requires that we read and rewrite the entire file. Thus we avoid sequential access for large files.

....Random-Access Files

Random-access files are slightly more complex to program. But they tend to produce faster results and readily facilitate large files and ones that require frequent updating. Random access does not mean that we go about in an erratic fashion accessing data in the file. It means that we may access data at random anywhere in the file. We may read the 31st entry, change it, and rewrite it without reading any other entry in the file. We simply need to organize our file so that we know where our data are in the file.

8-2...Sequential Files

OPEN, PRINT#, INPUT#, and CLOSE are the four BASIC keywords required to perform any useful work with data files.

....OPEN

OPEN establishes the communications channel between our program and the data file on the disk. OPEN creates the link between the file and

the program and points to the beginning of the file. Further, OPEN creates an area in memory where data are temporarily held. This is called a file buffer. The file buffer is used to manage the flow of data between the program and the disk file. We need to open a file for either Output or Input. Our program may either Output data to the file or Input data from the file. A file opened for sequential access cannot do both on a single channel. In addition we need to select a channel number for our file and give it a name.

```
11Ø  OPEN "O", #1, "SAMPLE.DAT"
```

Opens a file named SAMPLE.DAT for output on channel number 1. We may select any channel number from 1 to 15. Microsoft BASIC automatically allows for three file channels. If we want some other number, we simply declare that when we invoke MBASIC or GBASIC. See Sidelight 9.

If the file already exists, OPEN simply forms the necessary linkage. If the file does not exist then OPEN creates it. Once opened in "O" mode, a file is ready to receive data. This is done with a PRINT # statement.

....PRINT

PRINT #C outputs data to the file buffer numbered C. Whenever the file buffer fills up, it is written out to the disk file itself. That is, a copy of the information in the buffer is written out to the disk. And PRINT #C again fills the buffer. Since disk access is slow and memory access is fast, this buffer scheme is used to reduce the time it takes to move data back and forth between a program and a file.

PRINT # performs very much like PRINT. When data items are separated by semicolons, little space is wasted. When data items are separated by commas, spaces are included to provide for 14 character columns. Since this takes up space on the disk, we usually avoid comma spacing. A data item in a file is much like a display line on our screen or our printer. The entire line is treated as a whole. So, if we use

```
2ØØ  NUMBER = 456
21Ø  PRINT #1, "This is a test"; NUMBER
```

to output data to a file, then the data item consists of the entire expression as follows:

```
This is a test 456
```

just as though we displayed it on our screen. And the cr-lf (carriage return–line feed) goes out, too.

....INPUT

INPUT #C accepts data from the file on the channel numbered C. This is just a form of the INPUT statement used for accepting data from the

keyboard. A list of variables must name the values to be entered. To make a file available for INPUT #, it must first be OPENed in input mode.

```
110  OPEN "I", #1, "SAMPLE.DAT"
```

does the job. To read the data from our PRINT # example, we simply use a single string variable in an INPUT # statement.

```
INPUT #1, A$
```

will cause the string

```
This is a test 456
```

to be transferred from the file buffer to the string variable A$. It's that simple.

....CLOSE

The communications channel between the file and the program is severed with the CLOSE statement.

```
CLOSE #1
```

disconnects the program from the file opened on channel 1. Part of this process involves transferring data from the file buffer in memory to the file itself on disk. CLOSE without any file number disconnects all files. It is especially important to close files that have been written to. Failure to do this could result in losing data. Some other actions cause an automatic CLOSE, but it is not good practice to count on this.

This problem is of special concern when program execution terminates due to an error condition. In this situation, the file buffer is left hanging. We must issue a CLOSE from the keyboard to ensure that any data written to the buffer is copied out to the file. Executing an END statement does perform the functions of a CLOSE, but it is good practice to include an explicit CLOSE anyway.

....STOP, CTRL-C, CONT

STOP and CTRL-C leave things in the same condition as an error. CLOSE is not automatic with STOP or CTRL-C. We may use CTRL-C to interrupt our program during testing. This gives us the opportunity to display the values of variables and decide whether our program is performing satisfactorily. If all is well, then we may resume execution with CONT. We may insert STOP statements at key points in a program for the same purpose. As with CTRL-C, files are still OPEN so we may proceed with CONT. Since STOP displays the current line number, this is an important technique for finding trouble spots in a program.

If you get

```
Can't continue
```

it could be that you have edited a line of the program. This message is also displayed if the program was interrupted by an error instead of STOP, CTRL-C, or END.

We have four of the key sequential-files statements: OPEN, PRINT #, INPUT #, and CLOSE #. With these four statements we can create and use significant sequential files. Now we are ready to convert our little Alphabet program from Chapter 6 so that it is file-based. We need an initialization program to set up the file in the first place, and we need a program to play the game. All we need to do is take the DATA statements from Program 6-8h, write a new control routine, and create a new subroutine that READs DATA and PRINT #'s it to a data file. See Program 8-1.

```
90   REM ** Write some signs to a file for Alphabet game.
96   :
98   REM ** Control routine
120  OPEN "O", #1, "SIGNS.DAT"
140  GOSUB 7000 'Write the signs to the SIGNS file
150  CLOSE
190  END
1496 :
1498  REM ** The signs
1500  DATA Stop, Al's Pizza, Dairy Queen, Burger King
1502  DATA Yield, One Way, This Way Out, Detour
1504  DATA One Show Only Tonight, Exit Only, Entrance Only Please
1506  DATA Florida 2138 mi., Fly United, Jet Set Diner
1508  DATA Give Her a Valentine, Give Him a Valentine
1510  DATA First Avenue, North Side
1598  DATA Done
6996 :
6998  REM ** Write the signs to the SIGNS file
7000  READ A$
7020  PRINT #1, A$
7030  IF A$ <> "Done" THEN 7000
7090  RETURN
```

Program 8-1. Initialize the signs file for Alphabet game.

The control routine for initializing the signs file couldn't be simpler. Just OPEN the file, use a subroutine to read the signs from DATA statements and write the signs to the SIGNS file, and CLOSE it. That's all.

Look at the subroutine at line 7000. Notice that the end-of-data signal word "Done" is written to the file. This means that we can use the subroutine at line 1000 of our original program with very slight changes. When we play the game we will get the data from the file instead of from DATA statements in our program. Instead of READing DATA we want to INPUT # the file data. The conversion is astonishingly straightforward. See Program 8-2a.

```
   998   REM ** Load the signs array
   1000  N = 0
-->1010  INPUT #1, A$ : IF A$ = "Done" THEN 1080
   1020   N = N + 1
   1030   SIGNS$(N) = A$
   1040   GOTO 1010
   1080  PRINT "There are:"; N; "signs in this game."
   1085  GOSUB 1100
   1090  RETURN
```

Program 8-2a. Load the Alphabet game road signs.

The only change in this subroutine is in line 1010. We replaced READ with INPUT #1,. It is that simple.

Next we need only provide an OPEN before GOSUB 1000 and CLOSE after. See Program 8-2b.

```
   100  DIM SIGNS$(50)
   102  DEF FNU$(A$) =
             CHR$(ASC(A$) + 32 * (ASC(A$)>96 AND ASC(A$)<123))
-->105  OPEN "I", #1, "SIGNS.DAT"
   110  GOSUB 1000 'Load the signs array
-->115  CLOSE
   120  GOSUB 2000 'Establish game beginning
   130  GOSUB 3000 'Simulate random signs along the road
   140  GOSUB 4000 'Did the player spot the next letter?
   145    IF CAPA = 0 THEN 130 'If not then repeat step 3
   150  GOSUB 5000 'Is the next letter really on the sign?
   155    IF ALPHA1 < 91 THEN 130 'If not "Z" yet repeat step 3
   160  PRINT "Congratulations, you have made it through the alphabet"
   190  END
```

Program 8-2b. Changed control routine in Alphabet game for files.

All we had to do was insert line 105 to OPEN the file and line 115 to CLOSE it. Nothing more is needed in the control routine.

Finally, there is the matter of DELETEing the unwanted DATA statements in the range 1500 to 1590. It is remarkable that we have made such a major conversion with so little programming effort. It pays to organize our programs carefully.

```
   100  DIM SIGNS$(50)
   102  DEF FNU$(A$) =
             CHR$(ASC(A$) + 32 * (ASC(A$)>96 AND ASC(A$)<123))
   105  OPEN "I", #1, "SIGNS.DAT"
   110  GOSUB 1000 'Load the signs array
   115  CLOSE
   120  GOSUB 2000 'Establish game beginning
   130  GOSUB 3000 'Simulate random signs along the road
   140  GOSUB 4000 'Did the player spot the next letter?
   145    IF CAPA = 0 THEN 130 'If not then repeat step 3
   150  GOSUB 5000 'Is the next letter really on the sign?
   155    IF ALPHA1 < 91 THEN 130 'If not "Z" yet repeat step 3
   160  PRINT "Congratulations, you have made it through the alphabet"
```

```
190  END
996 :
998   REM ** Load the signs array
1000  N = 0
1010  INPUT #1, A$ : IF A$ = "Done" THEN 1080
1020   N = N + 1
1030   SIGNS$(N) = A$
1040   GOTO 1010
1080  PRINT "There are:"; N; "signs in this game."
1085  GOSUB 1100
1090  RETURN
1096 :
1098  REM ** Time delay for messages
1100  FOR J = 1 TO 1500 : NEXT J
1190  RETURN
1996 :
1998  REM ** Establish game beginning
2000  ALPHA1 = 65 'Get ready to look for 'A'
2090  RETURN
2996 :
2998  REM ** Simulate random signs along the road
3000  R = INT( RND * N + 1 )
3020  PRINT SIGNS$(R)
3030  FOR J = 1 TO 800 : NEXT J
3040  FOR J = 1 TO 24 : PRINT : NEXT J
3090  RETURN
3996 :
3998  REM ** Did the player spot the next letter?
4000  A$ = INKEY$
4005  IF LEN(A$) = 0 THEN CAPA = 0 : GOTO 4090
4010   PRINT A$; " ";
4020   A$ = FNU$(A$) : CAPA = ASC(A$)
4030   IF A$ < "A" OR A$ > "Z" THEN 4000
4050   IF CAPA = ALPHA1 THEN 4090
4060    PRINT "Not the next letter in the alphabet" : GOSUB 1100
4070    GOTO 4000
4090  RETURN
4996 :
4998  REM ** Is the next letter really on the sign?
5000  FOR J = 1 TO LEN(SIGNS$(R))
5010   B$ = FNU$(MID$(SIGNS$(R),J,1))
5020   IF A$ = B$ THEN 5050
5030  NEXT J
5040  PRINT "Your letter is not on the sign" : GOSUB 1100
5045  GOTO 5090
5050  PRINT "Good" : GOSUB 1100
5060  ALPHA1 = ALPHA1 + 1
5090  RETURN
```

Program 8-2. File-based Alphabet game.

Since we have changed none of the game-playing features of this program, it will behave exactly as the last version did.

There are lots of things we could do now to add interest to the program. We could make it tell the player when a letter was on a sign but not identified. Now that the program is file-based we could ask each player to enter his or her favorite sign. If the entry is not in the file, then the program could add it and rewrite the file. The program could use the number of letters in the sign to seed the random-number generator instead of asking for a name.

Problems for Section 8-2

There are lots of games that could be computerized using files to store information. Use your imagination.

1. Modify the Alphabet game program to seed the random-number generator by requesting a new sign from the keyboard. Use the number of characters in the sign as the seed. Compare the new sign with those in the array. If it is a new one, add it to the end of the array, OPEN the file for output, and write the array out to the file. Then CLOSE the file. Just use the subroutine at line 7000 in Program 8-1.
2. Modify Program 8-2 to report each time the player missed the next letter twice.
3. Write a program to tabulate the number of times each letter of the alphabet occurs in the signs file.
4. Add a routine to your program in Problem 3 to arrange the results in order of frequency of occurrence.
5. Write a program to play Geography. In this game two or more players take turns thinking of place names. Each player must name a place such that the first letter matches the last letter of the previous player's place. Have the program save all new place names in a disk file. No name may be used twice in the same game. Make the computer one player in a two-player game.

8-3...A Program Is a File, Too!

When we write useful programs, we save them on disk. Now the program is a file. If we use the ",A" option to save the program, it is accessible to other programs. (See Appendix C.) This is exactly how a text editor works.

Let's write a little program to simply display a program stored on disk. Then let's think about expanding it to pretty up program listings. Some programmers use lots of colons to include several statements on the same line. This does save computer memory and disk space. The drawback is that too much of this makes programs hard to read. We can break each of those lines up into individual statements and display them separately. But

first we just display the program as is. Note that the program we are writing can display any file stored in ASCII format.

We simply OPEN for input and repeatedly INPUT data from the file, displaying as we go. Let's use our program to display Program 8-2a as the first example. See Program 8-3.

```
   100  REM ** Display a program from disk
-->200  OPEN "I", #1, "LOADSIGN.BAS"
-->210  INPUT #1, A$
   240   GOSUB 9100
   250   GOTO 210
   9096 :
   9098  REM ** Straight PRINT
   9100  PRINT A$ : RETURN
```

Program 8-3. Display a program from disk.

We have named Program 8-2a as "LOADSIGN.BAS" in line 200. Let's do it. See Figure 8-1.

```
   998   REM ** Load the signs array
   1000  N = 0
-->1010  INPUT #1
   A$ : IF A$ = "Done" THEN 1080
   1020   N = N + 1
   1030   SIGNS$(N) = A$
   1040   GOTO 1010
   1080  PRINT "There are:"; N; "signs in this game."
   1085  GOSUB 1100
   1090  RETURN
   Input past end in 210
```

Figure 8-1. Execution of Program 8-3.

Figure 8-1 reveals two serious problems. We can solve both of them easily with new BASIC features. Notice that line 1010 is displayed as two lines. That is because there was a comma in the program line. A comma here means the same thing that a comma entered at the keyboard in response to INPUT means. It separates data items from each other. To cure this we need a statement that reads data from the file until a carriage return–line feed is encountered. LINE INPUT is made to solve just this problem.

....LINE INPUT

LINE INPUT accepts all data on a line up to the cr-lf pair. LINE INPUT #F does it for file #F. All we have to do is change line 210 to

```
LINE INPUT #1, A$
```

But what about the other problem? "Input past end" is similar to "Out of DATA". We can use EOF here.

....EOF (End of File)

We can check for the end of a sequential file with the EOF function. We have used a special data item to signal the end of data in our programs thus far, but a program stored as a file by BASIC has no such easy signal. EOF(F) returns 0 if there is more data in the file on channel number F. We get −1 if the end of the file has been reached. So all we need is to move line 210 to 220 and code

```
IF EOF(1) THEN END
```

at line 210. See Program 8-4.

```
  100  REM ** Display a program from disk
  200  OPEN "I", #1, "LOADSIGN.BAS"
-->210  IF EOF(1) THEN END
-->220   LINE INPUT #1, A$
  240    GOSUB 9100
  250   GOTO 210
 9096 :
 9098  REM ** Straight PRINT
 9100  PRINT A$ : RETURN
```

Program 8-4. Fix Program 8-3.

Executing this program will produce the display of Program 8-2a.

Now it's time to work on breaking out multiple statements on a line. We can just add a subroutine at 9200 to do this and change line 240 of Program 8-4. Let's look for space-colon-space. This means that we will have to write our programs with that sequence to separate statements on the same line. We can just add a subroutine at line 9200 and change line 240 to read GOSUB 9200. See Program 8-5.

```
  100  REM ** Display a program from disk
  200  OPEN "I", #1, "DISPLAY.BAS"
  210  IF EOF(1) THEN END
  220   LINE INPUT #1, A$
-->240   GOSUB 9200
  250   GOTO 210
 9096 :
 9098  REM ** Straight PRINT
 9100  PRINT A$ : RETURN
 9196 :
 9198  REM ** Format lines here
 9200  FOR J = 1 TO LEN(A$)
 9210   IF MID$(A$,J,3) <> " : " THEN 9250
 9220    PRINT LEFT$(A$,J-1)
 9230    PRINT "     ";
 9240    A$ = MID$(A$,J+1) : GOTO 9200
 9250  NEXT J
 9260  PRINT A$
 9290  RETURN
```

Program 8-5. Format multiple statements in a program.

Now, just for fun, let's save Program 8-5 as "DISPLAY.BAS" and use it to display itself. See Figure 8-2.

```
   100  REM ** Display a program from disk
   200  OPEN "I", #1, "DISPLAY.BAS"
   210  IF EOF(1) THEN END
   220   LINE INPUT #1, A$
   240   GOSUB 9200
   250   GOTO 210
   9096 :
   9098  REM ** Straight PRINT
-->9100  PRINT A$
        : RETURN
   9196 :
   9198  REM ** Format lines here
   9200  FOR J = 1 TO LEN(A$)
-->9210   IF MID$(A$,J,3) <> "
        : " THEN 9250
   9220    PRINT LEFT$(A$,J-1)
   9230    PRINT "     ";
-->9240    A$ = MID$(A$,J+1)
        : GOTO 9200
   9250  NEXT J
   9260  PRINT A$
   9290  RETURN
```

Figure 8-2. Execution of Program 8-5.

Our program has very nicely rearranged lines 9100 and 9240. But look at line 9210. The very statement that decides to break statements up has the sequence of characters we are looking for in it. We could easily put in another check to see if the J + 3rd character is a quotation mark by comparing to CHR$(34). (See the ASCII chart in Appendix D.) This is left as a problem.

Problems for Section 8-3

1. Add a check in Program 8-5 to fix the improper display of line 9210.
2. Modify Program 8-5 to display only lines containing FOR or NEXT statements. Hint: INSTR will be handy here.

8-4...Updating a Sequential File

We have seen how to create a sequential file. If we can read the file contents into an array in memory, we can easily rewrite the file with updated information. Now let's see a more general method for updating a file. We simply OPEN a second file temporarily. Once the changes are made and

the new file complete, we KILL the old file and give its name to the new file.

Let's maintain a list of names in a file. We will write a program that allows us to add a name at the beginning of the file. First, we need to create the file. That is easy with Program 8-6.

```
100  OPEN "O", #1, "DEMO01.DAT"
110  READ N$
120   PRINT #1, N$
140   IF N$ = "End" THEN CLOSE : END
160   GOTO 110
896 :
900  DATA Tom, Dick, Harry
999  DATA End
```

Program 8-6. Put some names in a file.

Now we can work on the program to do the actual update. Let's name our temporary file "DEMO01.TMP". See Program 8-7.

```
98   REM ** Add a name to a sequential file
100  OPEN "I", #1, "DEMO01.DAT"
110  OPEN "O", #2, "DEMO01.TMP"
120  INPUT "Add a name"; N1$
130  PRINT #2, N1$
140  INPUT #1, N$ : PRINT #2, N$
150  IF N$ <> "End" THEN 140
160  CLOSE
170  KILL "DEMO01.DAT"
180  NAME "DEMO01.TMP" AS "DEMO01.DAT"
190  END
```

Program 8-7. Add a name to a sequential file.

This program could be made to do a number of other things. We might want to prevent duplicates from getting into our names file. We might want to keep it alphabetized. We might want to add the ability to delete names. All of these are relatively straightforward tasks. We leave them as problems.

Problems for Section 8-4

1. Write a program to simply display the names in the names file.
2. Modify Program 8-7 to prevent duplicate names from ever getting into the file. This can be done without an extra pass through the file.
3. Modify Program 8-7 to keep the file alphabetized. Of course, the DATA in Program 8-6 must be in the right order to begin with.
4. Rewrite Program 8-7 to delete a name.

SIDELIGHT 8

Double Buffer

Double what?? In Chapter 8 we updated a sequential file by two different commonly used methods. For jobs that allow us to add all new entries at the end of the file, here's another scheme.

OPEN the same file on two different file channels. Open it for input on one channel and for output on another. Then transfer the entire file by INPUT #'ing from the input channel and PRINT #'ing to the output channel. Then add the new item at the end. This method also works for deletions anywhere in the file.

One advantage over creating a second file, killing the old file, and finally renaming the scratch file is that the double-buffer method requires no extra disk space. A disadvantage is that additions must come at the end. Let's just use Program 8-6 from Chapter 8 to write a few names in the file to get started.

Now we want a program that lets us add names. We can easily transfer all names up to the "End" marker. Next we want to request a name from the keyboard, PRINT # it to the file, and PRINT # "End" out there. Finally, we CLOSE the file. See Program 8-8.

```
-->100  OPEN "I", #1, "DEMO01.DAT"
-->105  OPEN "O", #2, "DEMO01.DAT"
   110  INPUT #1, N$
   115  IF N$ = "End" THEN 135
   120   PRINT #2, N$
   125   PRINT N$
   130   GOTO 110
-->135  INPUT "Enter a new name"; N1$
   140  PRINT #2, N1$
   150  PRINT #2, N$ 'Be sure End goes out
   160  CLOSE #1, #2
   190  END
```

Program 8-8. Double-buffer sequential-file update.

We have to be a little careful with this double-buffer business. There are some pitfalls to be avoided. Whoever runs this program must never exit the INPUT at line 135 with CTRL-C. This would result in a file without the "End" marker. The next time we run this program we will get

```
Input past end in 110
```

One other word of caution. Look carefully at lines 100 and 105. It is no accident that we opened for input before we opened for output. When a file is opened for output, BASIC points to the beginning of the file and clears out the first record. By opening for input first, we have caused the first data in the file to be copied to a buffer area in the memory of our computer. Now, no harm is done by clearing it out in line 105.

Chapter 9

Random-Access Files

Since entries in a file may vary in length, they may require varying amounts of space. Therefore, there is no way of predicting just where the 5th or the 50th entry might begin. So, for sequential files, we must always read from the beginning of the file. When we write to such a file we must write the entire file. As the file becomes larger and larger this all takes more and more time.

It takes a little more programming effort to work with random-access files than it does for sequential access. But they tend to produce faster results and readily facilitate large amounts of data. Random access is essential for applications that require frequent updating. Again, random access does not mean that we go about in an erratic fashion accessing data in the file; it means that we may access data at random anywhere in the file. We may read the 31st entry, change it, and rewrite it without reading any other entry in the file. We may do the same for any entry in the file. We simply need to organize our file so that we know where our data are. And that takes a little planning.

9-1...An Introduction

We use random-access files for all kinds of record keeping. The ability to access any data entry at will is ideal for applications where we will not be processing every entry every time we access the file. Contrast this with the Alphabet game, in which we must read every entry in the file with

every use of the program. Random-access files are used for name-and-address mailing lists, every conceivable financial accounting function, and stock portfolio management. Recipes, home-management data, and magazine-article reference material are all appropriate for random-access files.

In many applications several files are linked together to form a system of files. An order entry might "point" off to a mailing-list file and an inventory file.

In order to access data entries in a file at random, BASIC must be able to calculate the exact location of every entry. This can be done by allocating a fixed amount of disk space to each entry. Thus, if we allow 25 bytes for each entry, the tenth entry ends with the 250th byte and the 11th entry begins with the 251st byte. In practice, if we specify which entry, BASIC does the rest.

With sequential files the fundamental unit of storage is the character or byte. With random-access files the fundamental unit of storage is the record. A record is simply a collection of bytes. We think of a record as containing one entry. An entry consists of items that belong together. We might have an inventory file in which an entry contains the part number, price, number on hand, reorder point, and date last received. Those five items make up one entry. If 32 bytes is enough for the items in our entries, then we may organize our file into records that contain just 32 bytes. The record size is entirely up to us—well, as long as our record calls for 128 or fewer bytes it is up to us. (For larger records see Sidelight 9.) We decide record size according to our application. It is important to study each application thoroughly and plan effectively how we will organize files to manage the data required.

Often a group of programs will be used to handle a file or system of files—one program to enter and delete entries, another to edit entries, and perhaps a third to print a nicely formatted report to display all of the data in the file. Additional programs may be used to prepare reports of all kinds.

A new set of tools is needed to work with random-access files.

9-2...Some Tools

We have a set of new keywords and conditions that enable us to work with random-access files. We can get started with FIELD, LSET and RSET, PUT, and GET. In addition, we will learn new ways to use OPEN. CLOSE works just the way it did in the last chapter.

....OPEN

We open for random access with the "R" mode indicator in an OPEN system. Further, we may declare the record length.

```
100  OPEN "R", #1, "SAMPLE.DAT", 40
```

sets up a file named SAMPLE.DAT for random access on channel 1. A file channel forms the communications linkage between our program and the file itself on disk. The record size is 40 bytes. If we omit the record-length option, BASIC automatically makes it 128 bytes. However, when we invoke BASIC, we may change that with the /S: option. See Sidelight 9.

....FIELD

Once a file is opened for random access, the file buffer is established in memory. A FIELD statement is required to describe the layout of the buffer. That is, we need to tell BASIC just how each item in an entry will be placed in the record. All data in a random-access file must be in string form. Later we will see how to convert numeric values to string values and vice versa.

```
110  FIELD #1, 22 AS X$, 10 AS Y$
```

defines our records as having two strings. One string is allocated 22 characters and the other 10 characters. The string values may be assigned with LSET and RSET.

....LSET and RSET

Once the buffer is established in memory, we use a FIELD statement to partition the buffer according to our needs. In order to move data into the file itself, we need to first place it in the file buffer. This is done with an LSET or RSET statement.

```
370  LSET Y$ = "TEST"
```

loads the string value into the space in the buffer designated by Y$ in the FIELD statement.

LSET differs from LET in two regards. LSET assigns a string value in the file buffer. LET may not be used for this purpose. LET assigns a string value in an area of memory restricted to variable usage. LSET moves the string value into the left end of the string variable and fills the right end with spaces, while LET simply assigns the string value to a string variable. LSET would move "TEST" into Y$ of our FIELD statement above as "TEST ". Thus, LSET always creates a string having as many characters as specified in the FIELD statement. It is important to note that we cannot place data in a file buffer with a LET statement.

RSET is the same as LSET except that the string value is loaded into the right end of the space allocated in the buffer. The left end is filled with spaces. So,

```
210  RSET Y$ = "TEST"
```

will cause Y$ to contain " TEST".

LSET and RSET have no effect when the variable has not been named in a FIELD statement.

....PUT

PUT is the statement we use to copy data from the file buffer in memory to the disk file on the disk. Once we have finished working on a record in memory, we want it written out to the disk.

```
210  PUT #1, 5
```

will write out the buffer to record number 5 of the file OPENed on channel number 1. If we omit the record number, PUT simply writes out to the next record.

....GET

GET is the statement we use to copy data from the disk file on the disk to the file buffer in memory.

```
305  GET #6, REC
```

copies the contents of record number REC of the file OPENed on channel 6 to the associated file buffer. Note that it really is a copy of the data. The data is not removed from the disk file. If we omit the record number, GET simply accesses the next record.

....CLOSE

The communications established by an OPEN statement are severed with a CLOSE statement.

```
490  CLOSE #3
```

terminates any activity on channel number 3. If we have entered any data into the file on that channel and executed a PUT statement, CLOSE will cause the current buffer contents to be written out to the disk.

We may also close several channels.

```
392  CLOSE #1, #2, #8
```

CLOSEs the three file channels designated in the statement. CLOSE by itself CLOSEs all active file channels.

....SUMMARY

Once we organize files in records of a fixed size we may get at any data entry in the file as long as we know where it is. Whether our file contains 50 or 1000 records, a program can access any data entry directly and quickly.

A few easy-to-remember statements are available to manipulate data in a file to solve problems of our choice. OPEN, FIELD, LSET and RSET,

PUT, GET, and CLOSE are all that we need to get started with random-access files. In the next section we will develop an example, then go into a little more detail and introduce some more tools.

9-3...A Sample Random-Access File

Suppose we are working on an accounting system. We have been assigned the task of creating a file to contain the labels for a chart of accounts. For example, we might designate account number 1 as real estate taxes, number 2 as personal property taxes, number 9 as medical expenses, and 99 as miscellaneous.

With just a little thought we can do the job. We might call the file "ACNAMES.DAT", for account names. Now we need to consider the record size. "Personal property taxes" contains 23 letters. So we need at least 23 bytes per record. We do not need to include the carriage return and line-feed characters in the byte count, as we do with sequential files. Let's just allow 30 characters for good measure.

It is a simple matter to OPEN a file and FIELD the corresponding buffer.

```
100  OPEN "R", #1, "ACNAMES.DAT", 30
110  FIELD #1, 30 AS X$
```

Program 9-1a. OPEN and FIELD the accounts-label file.

Since we have only one data item in each file record, it turns out that the record size is the same as our single data item. This is a special situation. The record size is usually the sum of the number of bytes required for all items in an entry. (It could be larger.)

If we limit account numbers to the range from 1 to 99 and we don't need them all, what do we do about the "holes"? Let's label them "Unassigned". We can accomplish this by first doing an LSET to store "Unassigned" in X$ and then doing a PUT in a loop that runs from 1 to 99. This is a routine we do only once in the life of the file.

```
200  LSET X$ = "Unassigned"
210  FOR REC = 1 TO 99
220   PUT #1, REC
230  NEXT REC
```

Program 9-1b. File accounts-label file with "Unassigned".

Notice that we only perform the LSET to load the buffer once. The PUT operation creates a copy in the file. The buffer remains intact. Thus, we can copy the buffer contents over and over again. Now we have a file with "Unassigned" written to all 99 records.

Finally we need a routine to write the real account labels to the file. This can be done by READing the labels from DATA in our program.

When each label is written to the file, the "Unassigned" label previously written there will be replaced.

```
   300  READ N, N$
   310  IF N$ = "Done" THEN 390
-->320   IF N < 1 OR N > 99 THEN 380
   340    LSET X$ = N$
   350    PUT #1, N
   360    GOTO 300
   380   PRINT N; "Out of range"
   390  CLOSE #1
   395  END
```

Program 9-1c. Write actual account labels to the file.

We are providing for "Done" as the signal for end of data. In line 320 we check to see if the account number is within the agreed-upon range. If a value is out of range we get a little message. We would fix the incorrect data and run the program again. Little checks like this save untold grief later on. Working with files increases the complexity of programming. An error in the data written to a file by one program may later look like a programming error in some other program. A little extra care along the way is worth the effort.

We put this all together with sample DATA as Program 9-1.

```
   80   REM ** Initialize account label file
   96  :
   100  OPEN "R", #1, "ACNAMES.DAT", 30
   110  FIELD #1, 30 AS X$
   196 :
   198  REM ** Fill file with "Unassigned"
   200  LSET X$ = "Unassigned"
   210  FOR REC = 1 TO 99
   220   PUT #1, REC
   230  NEXT REC
   296 :
   298  REM ** Write out actual labels
   300  READ N, N$
   310  IF N$ = "Done" THEN 390
   320   IF N < 1 OR N > 99 THEN 380
   340    LSET X$ = N$
   350    PUT #1, N
   360    GOTO 300
   380   PRINT N; "Out of range"
   390  CLOSE #1
   395  END
   896 :
   900  DATA 1,  Real estate taxes
   901  DATA 2,  Personal property taxes
   902  DATA 9,  Medical expenses
   903  DATA 99, Miscellaneous
   904  DATA 22, Sewer and water
```

```
9Ø5  DATA 38, Cleaning and maintenance
9Ø6  DATA 44, Mortgage interest
99Ø  DATA Ø,  Done
```

Program 9-1. Initialize an accounts-label file.

It's a good thing that we didn't settle for 23 characters in a record, since "Cleaning and maintenance" requires 24.

....SUMMARY

We have seen most of the basic tools we need for random-access files. With OPEN, FIELD, LSET, RSET, PUT, GET, and CLOSE we can perform all of the operations required to create and maintain a simple file. It is important to analyze our space requirements so that we allow enough space in each record for the largest entry we will encounter. It is important to execute a CLOSE statement to copy the final buffer contents to the disk file itself.

Problems for Section 9-3 .....

1. Write a program to print chart-of-account labels. Simply scan the file and print the number and label for all assigned records.
2. Write a program to allow for adding account labels. Your program should first determine that the account number is actually unassigned.
3. Write a program to allow renaming account labels. This would be useful when a label is incorrectly spelled due to a typing error or a more accurate label has been suggested. In practice, accountants don't arbitrarily change account labels.
4. Sometimes it is desirable to have shorter labels for reports that have little space. Change Program 9-1 so that two labels are stored in each record. One label will be the full description and the other will be an abbreviation. Limit abbreviations to eight letters.

9-4...Some More Tools

We have worked with a random-access file using string values only. Obviously there must be some way to handle numeric values. A special set of functions is provided to represent numeric values in string form. These functions provide for compact storage of numeric data. We have MaKe functions and ConVert functions. The MaKe functions make strings out of numeric values. The ConVert functions convert string values into numeric values. These are very different from the STR$ and VAL functions.

....MKS$

The MKS$ function makes a string out of a single-precision numeric value. The string formed requires four bytes.

```
410  LSET Y$ = MKS$(Y9)
```

does the whole job of loading the string representation of the numeric value of Y9 into the buffer for Y$. We need a companion function to go the other way.

....CVS

CVS converts a four-byte string to a single-precision numeric value.

```
520  Z8 = CVS(Z$)
```

does it.

Let's create a file to store the names of the ten largest U.S. cities, their rank, and the percentage of growth from 1970 to 1980. Then we can write programs to prepare various reports.

Table 9-1 was prepared from information found in an almanac.

CITY	RANK	% GROWTH
Baltimore	9	−13.1
Chicago	2	−10.8
Dallas	7	7.1
Detroit	6	−20.5
Houston	5	29.2
Los Angeles	3	5.5
New York	1	−10.4
Philadelphia	4	−13.4
San Antonio	10	20.1
San Diego	8	25.5

Table 9-1. Ten largest U.S. cities in 1980.

The program will simply OPEN a file and write each data set to a different record. We can easily use DATA statements for this. In practice, for larger applications we would have a system of programs. One of those programs would be used to enter data into the file and edit incorrect data already there. For a file with a thousand entries we would not have a thousand DATA statements in a program. The file would be managed directly from the keyboard. See Program 9-2.

```
100  OPEN "R", #1, "CITIES.DAT", 20
110  FIELD #1, 12 AS CITY$, 4 AS RANK$, 4 AS PERCENT$
196 :
200  FOR K = 1 TO 10
210   READ X$, R, G
```

```
220   LSET CITY$     = X$
230   LSET RANK$     = MKS$(R)
240   LSET PERCENT$ = MKS$(G)
250   PUT #1, K
280  NEXT K
290  CLOSE #1
890  END
896 :
900  DATA Baltimore, 9, -13.1
902  DATA Chicago, 2, -10.8
904  DATA Dallas, 7, 7.1
906  DATA Detroit, 6, -20.5
908  DATA Houston, 5, 29.2
910  DATA Los Angeles, 3, 5.5
912  DATA New York, 1, -10.4
914  DATA Philadelphia, 4, -13.4
916  DATA San Antonio, 10, 20.1
918  DATA San Diego, 8, 25.5
```

Program 9-2. Write ten-largest-cities data to random-access file.

We have simplified this project by stating that we will have ten cities. In the next chapter we will develop ways to manage files that have no preset or fixed number of records.

Now that we have the file, one of the easiest tasks we might perform is to simply display the data in a neatly arranged format in the same order in which it appears in the file. This is left as an exercise.

We might want to see the data in the file arranged by rank. An easy scheme will be to form a ten-element array that contains the record positions of the appropriate data. Array element 1 will contain the record number of the data for New York, and array element 10 will contain the record number of the data for San Antonio. So our program will first have to scan the file building the array and then access the records in order according to the array just formed for display. See Program 9-3.

```
   50   REM ** Display cities in rank order
   80   DIM ARRAY(10)
   96  :
  100   OPEN "R", #1, "CITIES.DAT", 20
  110   FIELD #1, 12 AS CITY$, 4 AS RANK$, 4 AS PERCENT$
  196 :
  198   REM ** First load the array with record number
  200   FOR REC = 1 TO 10
  210    GET #1, REC
  220    R = CVS(RANK$)
-->230   ARRAY( R ) = REC
  250   NEXT REC
  296 :
  300   PRINT "City          Rank     % Growth"
  310   FOR K = 1 TO 10
  320    GET #1, ARRAY( K )
  330    R = CVS( RANK$ )
```

```
   340   G = CVS( PERCENT$ )
-->350   PRINT USING "& ##        ###.#"; CITY$, R, G
   380  NEXT K
   390  CLOSE #1
   890  END
```

Program 9-3. Display cities in rank order.

Look at line 230. That program statement loads the array with the record where the city with the appropriate rank will be found in the file. The position in the report is the position in the array, and the data value stored in the array is the number of the record in the file. This is easily done with a single-dimension array. We have here a very special situation. Most data does not include its own order position as an item.

Again we have used PRINT USING to good advantage in line 350. Notice the ampersand (&) there. We may include that character as a signal to BASIC to display a string found in the expression list following the USING string. In this case that is CITY$. See Figure 9-1.

```
City          Rank     % Growth
New York        1        -10.4
Chicago         2        -10.8
Los Angeles     3          5.5
Philadelphia    4        -13.4
Houston         5         29.2
Detroit         6        -20.5
Dallas          7          7.1
San Diego       8         25.5
Baltimore       9        -13.1
San Antonio    10         20.1
```

Figure 9-1. Execution of Program 9-3.

It is important to realize that nothing we have done in this program has changed the data in the file. The data has been rearranged on paper only. Writing report-generating programs that do not modify the data file makes all reporting programs totally independent from each other.

....MKI$, MKD$, CVI, and CVD

We also have functions to work with integer and double-precision numeric values. MKI$ makes a two-byte string out of an integer numeric. CVI converts its back. MKD$ makes an eight-byte string out of a double-precision numeric. The process is reversed with the CVD function.

Now we have full flexibility to work with all of the data formats available to us. When we are working with files that may grow to hundreds and thousands of records it becomes important to fit data as compactly as possible.

....SUMMARY

We have the ability to store numeric data as strings in random-access files using the MaKe (MKI$, MKS$, and MKD$) and ConVert (CVI, CVS, and

CVD) functions. Strings created with MaKe must also be LSET or RSET into the FIELDed buffer.

We have seen an example here of rearranging data stored in a file for the purpose of producing a report. This was done without changing the file itself. Thus, various reports need not interact.

Problems for Section 9-4

1. Write a program that simply displays the data in the census file in alphabetic order.
2. Write a program that uses a sorting technique from the chapter on arrays combined with the method used in Program 9-3 to display the cities of Table 9-1 in order of increasing growth.
3. Convert the sieve of Eratosthenes program (7-2) to use each record of a file to store one element of the array. Be sure to test your program with a small upper limit before you experiment with large values. Program errors will take longer to detect if you have to wait a long time before the computer displays the results.

SIDELIGHT 9

Initialization Options

While most of the time it will be adequate to access BASIC by simply typing MBASIC or GBASIC, occasionally we may have a special requirement. BASIC establishes the number of available file channels, the number of bytes per file record, and the amount of computer memory that will be used. BASIC arbitrarily provides 3 file channels, 128 bytes per record, and all of memory. These are the defaults—the values we get by doing nothing.

..../M:

If you are using machine-language subroutines, you can reserve memory with the /M option.

```
A>MBASIC /M:33791
```

would exclude all memory above 33K from use by BASIC. Of course, the more memory you reserve, the less there is for your BASIC program.

..../F:

Often programs need to use more than 3 files at one time.

```
A>MBASIC /F:7
```

will provide up to 7 file channels. Each channel requires 178 bytes for overhead. In addition, each channel is allocated 128 bytes for storage of one record of data. Changing the record size changes the number of bytes allocated.

We may call for 0 to 15 channels. Just don't try to work with any files after specifying 0 channels. Calling for 16 or more will be greeted by the

```
Illegal function call
```

error message. On the other hand, a request for a negative number of channels evokes the

```
Syntax error
```

message.

..../S:

We can change the record size with the /S option. Why /S? Because records are sometimes referred to as sectors.

```
A>MBASIC /S:64
```

sets the record size to 64 bytes. Any integer greater than 36 may be used. Values less than 37 evoke a

```
Syntax error
```

message. If we select a value such that all of memory is used, then we may expect the

```
Out of memory
```

message.

Any of the parameters supplied to MBASIC may be entered in decimal, octal, or hexadecimal notation. Thus 10, &012, and &H000A are equivalent and may be used interchangeably.

....MBASIC FILENAME

We can directly execute a BASIC program by naming it right in the initialization instruction. Of course, BASIC will default to the .BAS extension if none is specified. In the event that we name a nonexistent program, BASIC will load and issue the

```
File not found
```

error message and return to CP/M.

These options may be used in any combination desired. Thus we can incorporate BASIC programs in SUBMIT files of CP/M. In order to utilize the powerful SUBMIT capability, the exit from the BASIC program must be via the SYSTEM statement. SYSTEM will cause BASIC to exit to the CP/M system. If a SUBMIT file is active, then the next command in the file will be performed.

Chapter 10

Random-Access Address List

Let's develop a computerized name-and-address list, a common need for business and personal use. The idea here is to store all the name and addresses in a disk file. Then we may extract those we need for any particular situation. Names may be classified by a code. We might set up a personal family mailing-list file using F, H, W, or C to designate friends of family, husband, wife, or children. A business might use B and S for billing and shipping addresses.

In business it is common practice to arrange these names alphabetically or by zip code or business volume. In order to achieve this we would not rearrange the names file itself; instead we would create a file that contains just a list of the records in the desired order. We might maintain several such lists of record numbers. Then we can easily write a program that will read a list of record numbers to print the corresponding name-and-address data from the data file in the desired order.

10-1...Design the File

Let's organize a program to build the mailing-list data file. There are a number of major tasks involved. One part of the program needs to request all of the necessary data from the keyboard. Another will write the entry into the file. Another will have to determine where the new entry belongs.

We will have to organize the entry itself. We must decide what information belongs in an entry and how many characters to allow for each item. Then we must calculate the necessary record size. Our program

must include code to manage all these things. We need a routine that will write the entry into the data file. Probably the most important part of writing the program is deciding how to organize entries within the file.

When we sit down to enter the first name and address we know that the file is empty. After that we have no idea how many names are in the file. Therefore we have no idea where the next entry should go in the file. We could use a piece of paper to keep track of how many names there are. But then we might just as well keep the names on paper, too. The whole idea is to let the computer do the work. We need to develop a plan for keeping track of where things are. One scheme is to assign each entry its record number as an identification number and include that number as part of the data entry. Thus, the first name in the system will be number 1, the second will be number 2, and so on. Now we can have the next number to be assigned saved in the file itself. A good place to do this would be in record 0. But there is no record 0. We can easily create one, however. If we call our working file "NAMES.DAT", we can put this information in "NAMES.ZER". So a file with no names in it should have a 1 stored in our little ".ZER" file. We can easily write an initialization program to do this. Then after each new name is entered the program adds 1 to that value in our ".ZER" file.

Even though we are thinking about a program to enter names in a file, this is the time to think about how names are to be deleted. Deleting names from a mailing list can be handled in one of several ways. We could replace the name with the word "Deleted." Or we could set things up so that each deleted entry immediately frees a record for new data. We can make deleted records available for new entries by setting up a catalog of available space within the file itself. Including the record number as part of the data fits right in here. Thus, we are going to build a catalog of available record numbers threaded through the data file. Then when an entry is deleted we store the number of the last deleted record in the deleted record and then store the number of the currently deleted record in the ".ZER" file along with the number of the next highest record in the file. This will leave a trail of deleted record numbers beginning with the number stored in file ".ZER". Now we have two numbers there—the next record at the end of the file and the most recently deleted record. When we start up a new file, the most recently deleted record will be 0.

This scheme also provides a method for determining whether an entry has been deleted or not. Read the record. If the identification number equals the record number, then it is real data. If not, then the entry has been deleted and the number is the record number of the previously deleted record. Note that the first deleted record will contain a value of 0. As an example of a file with deleted records see Figure 10-1 on page 147. Let's trace the available-space catalog in Figure 10-1. The second number in "FILE.ZER" is 8. Look at record 8 of FILE.DAT. There we find a 4.

FILE.ZER

9{on the end}, 8{last deleted entry}

FILE.DAT

1	1 Jones John . . .
2	2 Smith William . . .
3	3 Hayes Mary . . .
4	6{deleted entry} . . .
5	5 Bradshaw Eleanore . . .
6	0{deleted entry (first one)}. . .
7	7 Hough Hugh . . .
8	4{deleted entry}. . .
9	{never used}

Figure 10-1. Layout of records in use and deleted.

Look at record 4. There we find a 6. Look at record 6. There we find a 0. Thus the deleted records are 8, 4, and 6. When we finally use record 6 for a new entry, the program should place a 0 in "FILE.ZER" where the 8 is now. Following this event, the next new entry will go to new space at the end of the file.

The entry program will have to look at the two record numbers stored in "FILE.ZER" and decide whether to place the new entry at the end of the file or on a record from which a name has been deleted. That is easy. If the deleted record number is 0 the new name goes on the end. Otherwise use the deleted record.

It is important to observe in all this that even though we are designing the program to enter data, it is necessary to thoroughly think through the deleting process. We must design the whole system before actually coding any part of it. Thus we avoid the mistake of having to redesign the system after programs have been written.

We have entering and deleting pretty well under control. Now how about changing an entry? As long as each name has an identification

number we can easily read the corresponding record and display each item as it appears, giving the opportunity to make changes in each case. We will also need to periodically print up a list of the names with the IDs. It should be relatively easy to write a program to scan the file from beginning to end, displaying the data in each undeleted record. That program can easily select various categories according to the code stored in the code item.

We seem to have thought through four functions of our mailing-list system: new, delete, change, and display. We have mentioned the need to initialize the data file once to prepare it for entering data. Let's do that first. Let's identify new space at the end of the file using the file variable NEWID$ and deleted old space embedded within the file with the file variable OLDID$. See Program 10-1.

```
98   REM ** Initialize .ZER file
100  FILENAME$ = "NAMES"
110  OPEN "R", #1, FILENAME$ + ".ZER", 8
120  FIELD #1, 4 AS NEWID$, 4 AS OLDID$
130  LSET NEWID$ = MKS$(1)
140  LSET OLDID$ = MKS$(0)
150  PUT #1, 1
160  CLOSE #1
190  END
```

Program 10-1. Initialize mailing-list file.

Once this program has been run, we may count on "FILE.ZER" containing a 1 and a 0. Of course, we must ensure that this program is never run again. Life can be quite complete without ever having to reconstruct a file system with a bad ".ZER" file. Of course, it is a good idea to maintain copies of any data system on extra disks. With good data backup it is easy to recover from such a catastrophe.

Let's now design the layout for a data record. See Table 10-1.

DATA ITEM	LABEL	MAXIMUM # OF CHARACTERS
Identification #	ID #	4
Code	CODE	2
Last name	LAST	20
First name	FRST	20
Address	ADDR	30
City	CITY	20
State	STAT	2
Zip	ZIP	5
Telephone	PHON	17
		120

Table 10-1. Record layout for mailing-list file.

Note the large value for the telephone number. It allows for an area code, an exchange, and a four-digit extension. The total comes to 120 characters. We might consider allowing for the four new digits in the zip code, too. If we let the program calculate the total number of characters in the routine that reads the label data in the first place we won't have to give any further thought to this.

If we are careful about listing all of the above considerations we will have the structure of the control routine for our name-and-address entry program. Once we have the control routine we may concentrate on a single subroutine at a time. The following shows the list of functions for the name-and-address entry program.

1. Read data labels and limits
2. Read available-space parameters (.ZER file)
3. OPEN the .DAT file
4. Display next available ID and request data.
   Terminate on null LAST name
5. Prepare available space
6. Write new entry in .DAT file
7. Write available-space info back to .ZER file.
   Do it again (repeat step 4)

Each of the numbered tasks listed above can be accomplished with a subroutine. Some of those subroutines will also be used by the other programs that we will be writing for our name-and-address system. To terminate on null LAST name we need to provide a way for the data-requesting routine to send back a signal to quit. "Do it again" will simply direct the program to repeat the functions again beginning with number 4.

We may arbitrarily select line numbers for the subroutines and for the control routine itself, and we will have the guts of our program completed. See Program 10-2a.

```
   198  REM ** Control routine
   200  GOSUB 2000 'Read data labels and limits
   210  GOSUB 1900 'Read available-space parameters .ZER file
   220  GOSUB 1800 'OPEN the .DAT file
   230  GOSUB 1700 'Display next available ID and request data
-->240  IF EXIT = 1 THEN CLOSE : END
                    'Terminate on null LAST name
   250  GOSUB 1600 'Prepare available space
   260  GOSUB 1500 'Write new entry in .DAT file
   270  GOSUB 1300 'Write available-space info back to .ZER file
-->280  GOTO 230   'Do it again (repeat step 4)
```

Program 10-2a. Control routine for mailing-list program.

We have seven subroutines and two control statements in our main routine of Program 10-2a. Line 240 requires that the value of EXIT be set to 1

if the operator desires to exit and set to any other value for any entry that is to be placed in the file. Line 280 simply uses a GOTO to repeat the request for another new entry. We will now write the subroutines, one at a time.

We read the data labels at 2000. If we give some more thought to how to design the routine to take data from the keyboard, we should be able to come up with a creative scheme. We could surely ask the eight questions in eight statements using INPUT with prompt. For each of the eight inputs we could have a statement that checks to see if the entry is too long. Any changes in the file design will require changing that routine. And when we write the editor program for the system we will have another routine to rewrite if we wish to use this system of programs for another mailing list. Wouldn't it be a good idea to put the prompt labels and the maximum field sizes in DATA and read them into two arrays? Of course it would. Then major changes in the program can be made with simple changes in the DATA statements. Our DATA statements will come directly from the labels and character limits in Table 10-1. We can read the DATA into arrays with a FOR . . . NEXT loop. Here is where we total up the number of characters and save that number in RLENGTH. See Program 10-2b.

```
   1998  REM ** Read data labels and limits
   2000  READ N0
   2010  RLENGTH = 0
   2020  FOR X9 = 1 TO N0
   2030   READ LABEL$(X9), L(X9) 'Item length
-->2040   RLENGTH = RLENGTH + L(X9)
   2050  NEXT X9
   2090  RETURN
   2096 :
   2098  REM ** DATA - labels and limits
   2100  DATA 9
   2102  DATA  ID #,     4
   2104  DATA  CODE,     2
   2106  DATA  LAST,    20
   2108  DATA  FRST,    20
   2110  DATA  ADDR,    30
   2112  DATA  CITY,    20
   2114  DATA  STAT,     2
   2116  DATA "ZIP ",    5
   2118  DATA  PHON,    17
```

Program 10-2b. Read data labels for mailing-list program.

In Program 10-2b, N0 is the number of data items in an entry. The labels are stored in the LABEL$ array and the maximum numbers of characters are stored in the L array. The completed program should include a DIMension statement to provide for the LABEL$( ) and L( ) arrays.

The subroutine to read the available-space parameters is very simple. It just reads the values placed there by the initialization program. See Program 10-2c.

```
  1898  REM ** Read available-space parameters
  1900  OPEN "R", #1, FILENAME$ + ".ZER", 8
  1910  FIELD #1, 4 AS NEWID$, 4 AS OLDID$
  1920  GET #1, 1
  1930  NS = CVS(NEWID$)
  1940  DS = CVS(OLDID$)
  1990  RETURN
```

Program 10-2c. Read available space in mailing-list program.

In Program 10-2c we have chosen to carry new space in the variable NS and deleted space in DS.

Before we may access any data in the ".DAT" file we must OPEN and FIELD it. Referring to Table 10-1 we see that the record size must be 120. We have already taken care of this in variable RLENGTH at line 2040 of Program 10-2b. Let's create a string array for the file to match the string array we will be using to accept data from the keyboard. This means that the FIELD statement must provide for all nine elements of the array. See Program 10-2d.

```
  1798  REM ** OPEN the .DAT file
  1800  OPEN "R", #2, FILENAME$ + ".DAT", RLENGTH
-->1810 FIELD #2, L(1) AS F$(1), L(2) AS F$(2), L(3) AS F$(3),
                  L(4) AS F$(4), L(5) AS F$(5), L(6) AS F$(6),
                  L(7) AS F$(7), L(8) AS F$(8), L(9) AS F$(9)
  1890  RETURN
```

Program 10-2d. OPEN and FIELD the mailing-list data file.

Line 1810 in Program 10-2d is a long one. We can make it easy to read by using CTRL-J or the line-feed character to arrange the various items of the field in neat columns on several lines. We write the subroutine once and forget about it. We have used the elements of the L( ) array in the FIELD statement so that any nine-element mailing list can be processed without having to EDIT line 1810.

Now it is time to display the next available ID and request data. We said we would do this at 1700. Since we have planned carefully, this will be very straightforward. The first job here is to determine the next actual available space. We choose to first make it new space. Then if there is any deleted space we reassign DS to the ID. We handle the label display and the data request with a FOR . . . NEXT loop. See Program 10-2e.

```
  1698  REM ** Process entry from keyboard
  1700  ID = NS : IF DS <> 0 THEN ID = DS
  1705  PRINT
  1710  PRINT LABEL$(1); ": "; ID
  1715  KDATA$(1) = STR$(ID)
  1720  FOR I9 = 2 TO N0
  1725   PRINT LABEL$(I9);
  1727   INPUT X$ : IF I9 <> 3 THEN 1735
-->1730  IF LEN( X$ ) = 0 THEN EXIT = 1 : GOTO 1790
  1735   IF LEN( X$ ) <= L(I9) THEN 1750
```

```
   1740    PRINT "Too long - Reenter"
   1745    PRINT "   : "; : GOTO 1727
   1750    KDATA$(I9) = X$
   1760  NEXT I9
-->1770  EXIT = 0
   1790  RETURN
```

Program 10-2e. Handle keyboard data entry for mailing-list program.

Note that in line 1730 we set EXIT to 1 if the response to the request for LAST name is of zero length. The length will be zero when the user responds with only the RETURN key. Otherwise EXIT is set to 0 in line 1770. We created a KDATA$ array to accept keyboard data. Later we will transfer it to file data in F$. We must include the KDATA$( ) and F$( ) arrays in the DIMension statement in the completed program.

Next we must prepare available space. What we do here depends on whether we are going to replace a deleted entry or write a new record. If we are going to use a new record we simply add 1 to the new-space variable and RETURN. If we are going to write this data to a previously deleted record then we must retrieve the record number that was written there when the deletion occurred. That number is essential for correctly maintaining the available-space catalog. Remember this from Figure 10-1?

```
   1598  REM ** Prepare available space
   1600  IF DS = 0 THEN NS = NS + 1
                   ELSE GET #2, DS : DS = CVS(F$(1))
   1690  RETURN
```

Program 10-2f. Prepare available space for mailing-list file.

Note that in this subroutine either new space changes or deleted space changes, but never both.

Once the available-space situation is taken care of, we may actually write the entry to the file. We need to LSET the ID value into F$(1) and then move all keyboard data from KDATA$( ) to F$( ) as well. Finally we PUT the data into record number ID.

```
   1498  REM ** Write new entry in .DAT file
   1500  LSET F$(1) = MKS$(ID)
   1510  FOR I9 = 2 TO N0
   1520   LSET F$(I9) = KDATA$(I9)
   1530  NEXT I9
   1540  PUT #2, ID
   1590  RETURN
```

Program 10-2g. Write a data entry in the mailing-list program.

And last but by no means least we must provide the subroutine that writes the available-space parameters to the ".ZER" file. This is exactly like the initialization program except that we must write NS and DS. See Program 10-2h.

```
1298  REM ** Write available-space info back to .ZER file
1300  LSET NEWID$ = MKS$(NS)
1310  LSET OLDID$ = MKS$(DS)
1330  PUT #1, 1
1390  RETURN
```

Program 10-2h. Write available-space parameters in mailing-list program.

Finally, in order for all of this to happen, we must include the file name in FILENAME$ and also include the appropriate dimensioning statement. See Program 10-2i.

```
98   REM ** mailing list program
100  FILENAME$ = "NAMES"
110  DIM LABEL$(9), L(9), F$(9), KDATA$(9)
```

Program 10-2i. Program parameters for mailing-list program.

This makes it very easy to work on a different mailing list with the same field lengths by simply changing line 100.

We list the complete program here for your convenience.

```
98   REM ** mailing list program
100  FILENAME$ = "NAMES"
110  DIM LABEL$(9), L(9), F$(9), KDATA$(9)
196  :
198  REM ** Control routine
200  GOSUB 2000 'Read data labels and limits
210  GOSUB 1900 'Read available-space parameters .ZER file
220  GOSUB 1800 'OPEN the .DAT file
230  GOSUB 1700 'Display next available ID and request data
240  IF EXIT = 1 THEN CLOSE : END
                 'Terminate on null LAST name
250  GOSUB 1600 'Prepare available space
260  GOSUB 1500 'Write new entry in .DAT file
270  GOSUB 1300 'Write available-space info back to .ZER file
280  GOTO 230   'Do it again (repeat step 4)
1296 :
1298  REM ** Write available-space info back to .ZER file
1300  LSET NEWID$ = MKS$(NS)
1310  LSET OLDID$ = MKS$(DS)
1330  PUT #1, 1
1390  RETURN
1496 :
1498  REM ** Write new entry in .DAT file
1500  LSET F$(1) = MKS$(ID)
1510  FOR I9 = 2 TO N0
1520   LSET F$(I9) = KDATA$(I9)
1530  NEXT I9
1540  PUT #2, ID
1590  RETURN
1596 :
1598  REM ** Prepare available space
1600  IF DS = 0 THEN NS = NS + 1
                ELSE GET #2, DS : DS = CVS(F$(1))
1690  RETURN
```

```
1696 :
1698  REM ** Process entry from keyboard
1700  ID = NS : IF DS <> Ø THEN ID = DS
1705  PRINT
1710  PRINT LABEL$(1); ": "; ID
1715  KDATA$(1) = STR$(ID)
1720  FOR I9 = 2 TO NØ
1725   PRINT LABEL$(I9);
1727   INPUT X$ : IF I9 <> 3 THEN 1735
1730   IF LEN( X$ ) = Ø THEN EXIT = 1 : GOTO 1790
1735   IF LEN( X$ ) <= L(I9) THEN 1750
1740    PRINT "Too long - Reenter"
1745    PRINT "  : "; : GOTO 1727
1750    KDATA$(I9) = X$
1760  NEXT I9
1770  EXIT = Ø
1790  RETURN
1796 :
1798  REM ** OPEN the .DAT file
1800  OPEN "R", #2, FILENAME$ + ".DAT", RLENGTH
1810  FIELD #2, L(1) AS F$(1), L(2) AS F$(2), L(3) AS F$(3),
                L(4) AS F$(4), L(5) AS F$(5), L(6) AS F$(6),
                L(7) AS F$(7), L(8) AS F$(8), L(9) AS F$(9)
1890  RETURN
1896 :
1898  REM ** Read available-space parameters
1900  OPEN "R", #1, FILENAME$ + ".ZER", 8
1910  FIELD #1, 4 AS NEWID$, 4 AS OLDID$
1920  GET #1, 1
1930  NS = CVS(NEWID$)
1940  DS = CVS(OLDID$)
1990  RETURN
1996 :
1998  REM ** Read data labels and limits
2000  READ NØ
2010  RLENGTH = Ø
2020  FOR X9 = 1 TO NØ
2030   READ LABEL$(X9), L(X9)
2040   RLENGTH = RLENGTH + L(X9)
2050  NEXT X9
2090  RETURN
2096 :
2098  REM ** DATA - labels and limits
2100  DATA 9
2102  DATA  ID #,     4
2104  DATA  CODE,     2
2106  DATA  LAST,    2Ø
2108  DATA  FRST,    2Ø
2110  DATA  ADDR,    3Ø
2112  DATA  CITY,    2Ø
2114  DATA  STAT,     2
2116  DATA "ZIP ",    5
2118  DATA  PHON,    17
```

Program 10-2. Entering names in a mailing-list file.

This program is intended to be a simple example of a workable mailing-list data entry program. Using the preceding discussion and some of the routines of this program you should be able to develop programs to delete entries, change entries, and print mailing labels.

There are many areas in which this program can be made more flexible. We might request the mailing-list file name from the program operator. We might eliminate the DATA statements from the program by placing that data in the ".ZER" file as well as the data already there. The benefits of doing things this way are tremendous. With all of the information about the mailing list stored in a file, our one program can be used to process many different mailing lists. We can handle different labels and different item lengths as the program stands. We have only to change line 1810 and line 110 to change the number of items in an entry. We will soon find that we have to write a program to manage the companion file that contains all of this nice information. That is a small price to pay. When we can change the behavior of a program by changing data in a file, we approach data-base-management capabilities.

Computers and programming have acquired an aura of mystery that puts many people off. We are working toward the day when people who use computers will not have to do any programming. We all use elevators without being elevator operators. Yet we still need people who are elevator experts. One goal for programmers is to create programs that can handle many tasks without changing the program itself.

Programming for the delete and change functions can be handled by either writing separate programs or by including the new subroutines necessary right in Program 10-2. We could provide a menu that lets the user select which function is desired.

....SUMMARY

Once we organize files in records of a fixed size we may get at any data entry in the file as long as we know where it is. In this chapter we have designed a mailing-list system. We have written the program to enter data into this file using keyboard interaction. Arrays have been used to good advantage to provide a flexible system. We need only change the name of the file in one line of our program to work with a different mailing list. By changing only a few DATA statements we can even dramatically changing the mailing-list file itself. This experience has pointed the way to concepts that will even allow us to store the characteristics of a mailing-list system in yet another file. The closer we come to this, the closer we come to a truly "user-friendly" system.

Problems for Section 10-1 .....

1. Incorporate a delete routine in the name-and-address entry program.

2. Write a program to edit data in the mailing-list file. Display each item and ask if the user wants to make a change.
3. Write a program to display all data from the file for names having a specified code.
4. Write a program that will print mailing-address labels. Set the program up so that it requests up to ten ID numbers from the keyboard and then prints all of the labels. (The next step would be to have the program READ the list of IDs from another file prepared by yet another program.)
5. Modify our mailing-list system by placing the labels and item limits in the ".ZER" file. Have the program request the file name from the keyboard. You will have to write a little program to write the data to the ".ZER" file in the first place. To do this you can FIELD the same file three ways. Thus, different records may be used for different purposes. Here is one possible set of FIELD statements.

```
1798  REM ** OPEN the .ZER file
18ØØ  OPEN "R", #1, FILENAME$, 8
1810  FIELD #1, 4 AS NEWID$, 4 AS OLDID$
1812  FIELD #1, 4 AS NØ$
1814  FIELD #1, 4 AS LA$, 4 AS LE$
189Ø  RETURN
```

SIDELIGHT 10

Mixed-Access Files

In some applications it may be desirable to design a file using a mixture of random and sequential access. We may use PUT and GET to position at the desired record and use PRINT # and INPUT # to manipulate the data items within the record. Remember that the use of FIELDed records results in all string values being filled with spaces to occupy the space allocated. This does not happen in sequentially written records.

With a sequential record we can design for efficient space use. We can calculate the record size from the true maximum space use. Suppose we have an application with several items in each record that fluctuate widely in size. Using a FIELDed record we are required to allocate space based on the maximum for each item. Suppose we have a situation in which the maximum for the first item is 35 characters and the maximum for the second item is also 35 characters, but the sum of the first and second items is never more than 50. We can save 16 characters per record by writing sequentially. Remember that a sequential PRINT # inserts a cr-1f at the end of the line.

With sequential access we are free to intermix strings and numerics as needed. The only catch is that we must calculate all the characters in the printed form of the numeric value. We could use the MaKe functions to work with numeric values in a file. This would allow us to compress our data into the record. Thus, a number like 9.71208E+20 could be stored in the space of just four characters plus the cr-1f delimiter. We would just use

```
220  X$ = MKS$(X9)
```

and then use PRINT # to get the contents of X$ into the file buffer. Of course, we would use the ConVert functions to recover the numeric values from the file later.

Chapter 11

Lo-Res Graphics

....Introduction

While many computer applications center on numeric manipulations and things like word processing, we find a great deal of interest in computer graphics. Some people are attracted by the ease with which data can be presented in chart form using a computer. Others will use graphics merely for the pleasing effects that are possible. Still others are attracted to the games aspect. Both high-resolution and low-resolution graphics are offered on Apple computers using a SoftCard. If you are already familiar with graphics in Applesoft, you will find few differences in MBASIC.

If you are using an Apple with an 80-column card, you will want to check on how it interacts with graphics. The TEXT command returns the computer to normal text mode after graphics work. If your screen "looks funny" in this situation, then you might want to disable the 80-column card for graphics work. You should have no such difficulty using an external terminal. With an external terminal, though, PRINT statements display on the terminal and graphics display appears on the Apple video screen.

11-1...Getting Started in Lo-Res

With just five statements we create a graphics screen and have full control over placement and coloring of 1920 blocks. GR, COLOR, PLOT, HLIN, and VLIN are all we need. In addition, the TEXT statement restores the computer to the conventional text screen. Let's examine them before we attempt our first program.

....The Graphics Screen (GR)

The statement

```
100 GR        or        100 GR 0
```

prepares the computer for graphics work—or graphics play. When this statement is executed, the screen is divided into two parts. The top part is organized into 40 columns and 40 rows. Thus, we have 1600 blocks at our disposal. This graphics portion of the screen is cleared to all black. (We'll get to the rest of the colors in a minute.) The remainder of the screen is reserved for four lines of regular text display. See Figure 11-1.

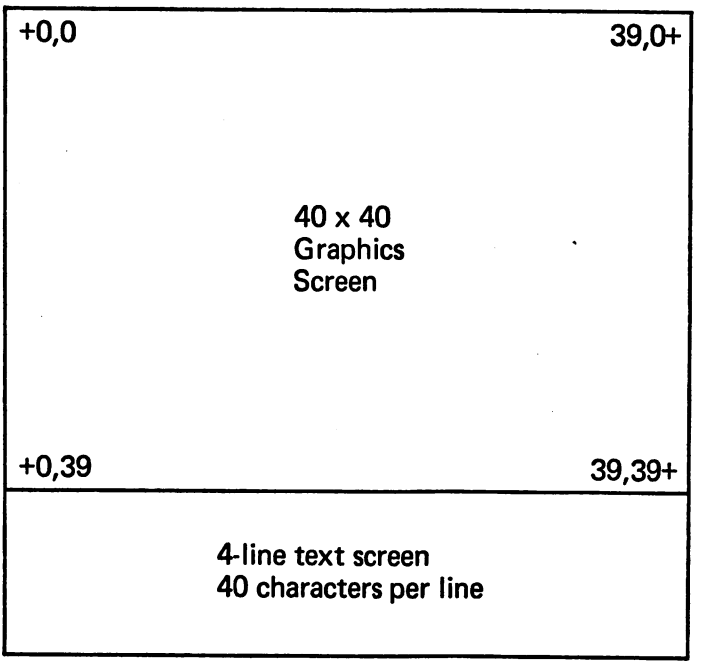

Figure 11-1. The graphics screen layout.

This arrangement is called mixed graphics and text. Each block is identified by its column and row. The block in the upper left corner is labeled 0,0. The block in the lower right corner is labeled 39,39. Columns are numbered from 0 to 39 from left to right and rows are numbered from 0 to 39 from top to bottom. This is not the same as the conventional rectangular coordinate system widely used in mathematics, but this difference presents no great obstacle. Many graphics applications are not related to mathematical pursuits anyway. When we want to represent a mathematical relationship we will find the translation easy enough. The plotted points are not exactly square, so we call them blocks rather than squares.

We may also use the GR statement to convert the entire screen for graphics work.

```
100  GR 1
```

does the job. Now the graphics screen is 40 blocks wide and 48 blocks high. The blocks are numbered 0 to 47 from top to bottom. We have replaced the four text lines at the bottom of the screen with eight rows for graphics blocks. Again, the graphics screen is cleared to all black. Note that any text display will appear as a mosaic of colored blocks in this area of the screen. So we will need to program any user interaction with the INKEY$ statement. Remember we used this in the Alphabet game in Section 6-4? INKEY$ takes a single character from the keyboard on the fly without displaying the character. GET is another statement that might be used here. See Sidelight 11 for this one.

Thus we have two Lo-Res graphics screens. They may be referred to as screen number 0 and number 1. Number 0 gives 1600 blocks in a grid 40 blocks wide and 40 blocks high. Number 1 provides 1920 blocks in a 40-by-48 grid. Of course, they aren't totally different screens; they are really variations of the same screen.

We may easily clear the screen to any Lo-Res color right in the GR statement.

```
100  GR 0, 10
```

will set up the mixed graphics-text screen and clear the graphics portion to color number 10. We'll do the colors next.

....Colors (COLOR)

Even if we are working with a noncolor monitor, we will have to pay attention to color. We will need to at least use white and black. There are 16 colors, numbered from 0 to 15 as shown in Table 11-1.

0	Black	8	Brown
1	Magenta	9	Orange
2	Dark blue	10	Gray
3	Purple	11	Pink
4	Dark green	12	Green
5	Gray	13	Yellow
6	Medium blue	14	Aqua
7	Light blue	15	White

Table 11-1. Lo-Res colors.

When the GR statement is executed, COLOR is set to black or the color specified in the GR statement. Once this happens we are free to assign the color of our choice with

```
110  COLOR = X
```

The COLOR statement may be used to establish any of the 16 colors listed above. Of course we may use a statement like COLOR = C1 to assign the desired color. Nothing visible happens when a COLOR statement is executed, just as nothing visible happens when a conventional assignment statement is executed. All plotting will appear in the most recently assigned color.

We may even use a statement such as

```
140  COLOR = COLOR + 1
```

COLOR must never be assigned outside the 0 to 15 range. Illegal values will evoke the

```
Illegal function call
```

error message. Decimal values will be rounded off.

And don't try

```
100  GR 0, COLOR
```

This will not clear the screen to the current value of COLOR. It will set COLOR to 0 and clear the screen to black. No harm is done. It's just that the behavior is a little confusing the first time one sees it. The results may be unexpected.

....Plotting Blocks (PLOT)

We plot blocks with the PLOT statement. The statement

```
500  PLOT 2, 3
```

will plot a block near the upper left corner of the graphics screen in the color that is active when line 500 is executed. Of course we may use PLOT X, Y so that values may be calculated to establish a position before executing the PLOT statement. Even

```
PLOT X + 3 * Y, 2 * Y - 1
```

may be coded. It's that simple.

....Drawing Lines (HLIN and VLIN)

We could plot blocks next to each other with several PLOT statements to draw lines. However, BASIC includes special statements to draw horizontal and vertical lines for us.

```
600  HLIN 0, 39 AT 0
```

will draw a horizontal line 40 blocks long at the very top of the screen.

```
700  VLIN A, B AT C
```

will draw a vertical line running from A to B in column C. We must ensure that the values of A, B, and C remain within the 0 to 39 range (or 0 to 47, as appropriate) to avoid the "Illegal function call" message.

VLIN is convenient for drawing bar graphs. We can incorporate some labeling in the four-line text screen at the bottom to make nicely readable charts.

....Restoring the Text Screen (TEXT)

The TEXT statement eliminates any graphics display and restores the text screen for us. The image on the graphics screen is gone forever. The cursor will sit at the bottom of the screen blinking at us awaiting our next command. We might just leave this statement out while we are writing and testing our programs. Then, when we are satisfied with the results, we insert the TEXT statement. This is also a good place for a HOME statement. Sometimes we will want to program in a little delay loop just to leave a graphics display on the screen long enough for the user to appreciate our handiwork.

We can learn a great deal about graphics using immediate mode. We can issue one of the GR commands, set a COLOR, and then PLOT points and draw lines directly from the keyboard. We will very quickly acquire a feel for the structure of the graphics screen and full-color Lo-Res graphics.

....Let's Experiment

One of the nice things about working with graphics programs is that we can produce dramatic changes in the results with minor changes in program code. We can demonstrate some pleasing effects with very short example programs. Even without a color monitor we get the idea. Consider Program 11-1.

```
5Ø   GR Ø
1ØØ  COLOR = 15
11Ø  FOR P = Ø TO 19
13Ø   PLOT P, P
19Ø  NEXT P
```

Program 11-1. A simple demonstration.

Can you tell what it does without running it? Program 11-1 simply plots a diagonal line from the upper left corner to the center of the screen. Type it in and RUN it. Now let's also draw a line from the upper right corner to the center. Just add

```
135   PLOT 39 - P, P
```

Note that when we think of measuring distance from the left edge we simply use the value and when we think of measuring distance from the

right edge we use 39 minus the value. The same thinking applies for the top and bottom edges. For full-screen graphics we subtract from 47 to measure from the bottom edge.

Now let's join pairs of points horizontally. To join the points plotted by

```
130   PLOT P, P
```

and

```
135   PLOT 39 - P, P
```

we could use a little FOR loop such as

```
130   FOR X = P TO 39 - P
132    PLOT X, P
135   NEXT X
```

But this is exactly what HLIN was designed to do.

```
130   HLIN P, 39 - P AT P
```

accomplishes the same result. Now we have a white triangle at the top of the screen. Let's add some color. Suppose we change the COLOR for each HLIN plotted. It is easy to insert statements to change the color and keep it in the 0 to 15 range. IF . . . THEN . . . ELSE is ideally suited to this situation.

```
100  COLOR = 0
180   IF COLOR = 15 THEN  COLOR = 0
                    ELSE  COLOR = COLOR + 1
```

Line 100 starts the color at black. Line 180 ensures that the COLOR value never exceeds 15. Now we have a triangle of many colors. Note that 4 colors appear twice since we have plotted 20 lines and used only 16 different colors. Let's keep going.

Let's add a triangle symmetrical to this one at the bottom. One BASIC statement ought to do it.

```
135   HLIN P, 39 - P AT 39 - P
```

That was easy. Finally, let's fill the triangles at the two sides.

```
140   VLIN P, 39 - P AT P
145   VLIN P, 39 - P AT 39 - P
```

Now we have a series of rectangles using the 16 colors available to us. See Program 11-2.

```
50   GR 0
100  COLOR = 0
110  FOR P = 0 TO 19
130   HLIN P, 39 - P AT P
135   HLIN P, 39 - P AT 39 - P
140   VLIN P, 39 - P AT P
```

```
145   VLIN P, 39 - P AT 39 - P
180   IF COLOR = 15 THEN  COLOR = Ø
                    ELSE  COLOR = COLOR + 1
190  NEXT P
```

Program 11-2. Drawing boxes of many colors.

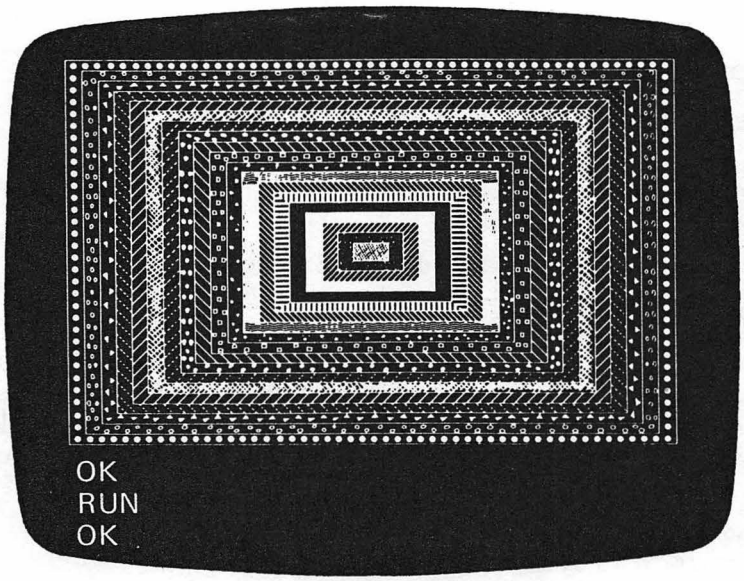

Figure 11-2. Execution of Program 11-2.

We can easily add motion to this demonstration by drawing the screen over and over again. We must be sure that the color keeps changing. We could just put in a new line 195 GOTO 110. Or we might enclose the whole routine in a FOR loop to repeat the pattern a fixed number of times.

Note that CTRL-S also stops graphics output. Sometimes it is useful to stop the figure as it progresses, giving us an opportunity to study our programs.

....SUMMARY

With five BASIC keywords, we control the Lo-Res graphics screen. GR prepares the screen for us. GR 0 gives us the 40-by-40 mixed graphics and text screen, while GR 1 invokes the full 40-by-48 graphics screen. Further, GR S, C may be used to clear the screen to color number C. We may set one of 16 colors in the range 0 to 15 by assigning the desired value to COLOR. We plot points with the PLOT statement. Lines are easy to draw with HLIN and VLIN. The rows are numbered from 0 to 39 beginning at the top of the screen. The columns are numbered from 0 to 39 beginning at the left edge of the screen. For full-screen graphics the rows are numbered 0 to 47. We restore the text screen with TEXT.

Problems for Section 11-1 .....

A few problems are offered here to get you going in your experimentation with graphics. Don't limit yourself. Try new things. You can't damage the computer with a BASIC program.

1. Change Program 11-2 so that the boxes get larger instead of smaller.
2. Change Program 11-2 so that a COLOR is selected at random instead of in sequence.
3. Change Program 11-2 so that the COLOR is selected at random for each line drawn.
4. Write a program to select a COLOR, an X-coordinate, and a Y-coordinate at random. Plot the selected point in the selected COLOR. Have the program repeat this without end. (CTRL-C gets you out.)
5. Write a program to simulate stars blinking in the sky. Randomly set COLOR to either 0 or 15. You'll want more 0's than 15's. Randomly select coordinates in a portion of the screen (try 10 by 10 in the upper left corner). It will be a little more realistic if you select only odd or even coordinate values.
6. Write a program to draw a bar graph picturing the following temperatures for a nine-day period:

DAY	TEMP
1	30
2	27
3	26
4	31
5	26
6	30
7	38
8	36
9	34

11-2...A Graphic Example

It is easy to program a computer to simulate the roll of a die and display a numeric result. Now that we know about Lo-Res graphics, let's also display a realistic picture of a die. It will be surprisingly easy to do. Remember that we can assign colors, plot small blocks, and draw lines. Meanwhile, let's concentrate on the nature of a picture of one face of a die.

Think of drawing the six possible faces of a die on ordinary graph paper. This can be done nicely if we use a rectangle five blocks wide and seven blocks high. Our drawing on graph paper will be distorted. But

when we get to the graphics screen the result will be more nearly square because the graphics blocks are wider than they are high. We come up with the sketch of Figure 11-3.

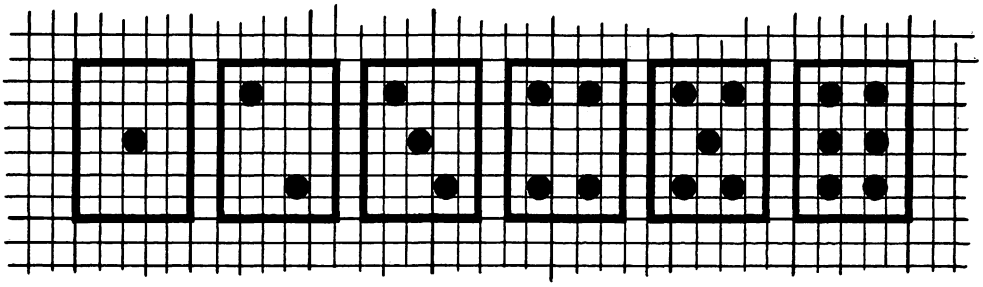

Figure 11-3. The six dice.

Now the computer problem separates into two parts. First, we need the die background. And second, we need six different configurations for the dots in some contrasting color.

The five BASIC keywords GR, COLOR, PLOT, HLIN, and VLIN are all we need to do wondrous things on the graphics screen. We can now plan how to apply them to draw a die. Let's first draw the "1" face of a white die. We need to turn on graphics, set COLOR to white, draw 5 vertical lines 7 blocks high, set COLOR to black, and PLOT a block in the middle of the 5-by-7 rectangle. Program 11-3 draws a "1" near the upper left corner of the screen:

```
98   REM ** The "1" face on a die
100  GR Ø
110  COLOR = 15
120  FOR X = 1 TO 5
130   VLIN 1, 7 AT X 'Plot background
150  NEXT X
160  COLOR = Ø
170  PLOT 3, 4        'Plot dot
180  END
```

Program 11-3. Draw the "1" face of a die.

Note that it would be equally correct to draw 7 HLINs 5 blocks wide. We simply chose the scheme that resulted in the fewer number of statements to execute. Program 11-3 executed line 130 only five times. Now see Figure 11-4. That is pretty nice. How do we get a "3"? Simply add the following two statements and run the new program:

```
165  PLOT 2, 2
175  PLOT 4, 6
```

After we have had a chance to study the graphics screen, we can type TEXT to clear it. Looking at Figure 11-3 we see that the seven positions on

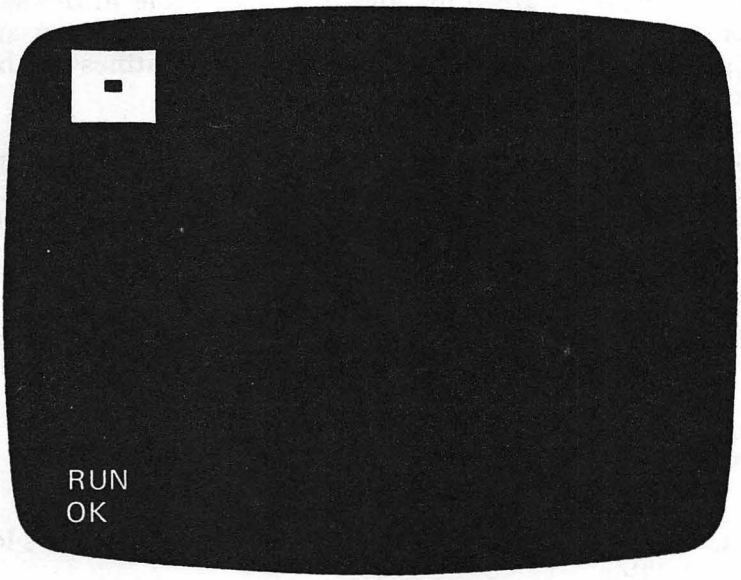

Figure 11-4. Execution of Program 11-3.

the face of the die where a dot may appear are 2,2; 2,4; 2,6; 3,4; 4,2; 4,4; and 4,6. By properly selecting from among these seven, we may draw any of the six faces.

Problems for Section 11-2

1. Write a program to display a die showing the "6" face in the upper right corner of the screen.
2. Write a program to display a pair of dice—one showing a "1" and the other showing a "3".

11-3...Divide and Conquer (More Dice)

Once we have written the code to display a die of a particular color having a particular face value in a particular place, it is hard to be inspired to write new code to display that same die in another location or another color. And it is even less exciting to consider displaying five dice this way. When we find ourselves writing routine after routine, each of which is only a slight variation of another routine, programming becomes tedious. It doesn't have to be that way. The more experience we gain in programming, the more opportunity we will have to utilize what we have already done. Often a current problem is only a slight variation of an old, already solved, one.

If we want to display a green die and then a pink die in the same location, the only thing that changes is the color. Clearly, it is a nuisance to duplicate the code that does the actual graphing. Subroutines will help us tremendously here.

For our green-die-followed-by-pink-die problem we need to have the program pause between the two displays. Otherwise, things will happen so quickly that we will not see the first die. This pause can be accomplished with a time-waster FOR . . . NEXT loop that does nothing else. The problem is solved in seven easy steps as follows:

1. Enable graphics mode.
2. Set green color.
3. Display the die.
4. Waste some time.
5. Set pink color.
6. Display the die.
7. End.

Once again, we have the control routine of a new program. Let's look at it. See Program 11-4a.

```
   100  GR 0, 0     'Enable graphics mode.
   110  COLOR = 12  'Set green color.
   120  GOSUB 1000  'Display the die.
-->150  FOR X = 1 TO 1500 : NEXT X
                    'Waste some time.
   160  COLOR = 11  'Set pink color.
   170  GOSUB 1000  'Display the die.
-->190  END         'End.
```

Program 11-4a. The control segment of a die-drawing program.

We think of

```
GOSUB 1000
```

as "display a die" without having to think about the actual BASIC statements required to do the display. Look at 150. That is our delay loop. For a longer delay, use a value larger than 1500. Without a delay, we would not even see the first die because it would be so quickly replaced with the second die.

The display routine is very easy. We may simply select those statements from our earlier die-drawing program and use appropriate line numbers. We may concentrate on the display without having to think about other parts of the program. See Program 11-4b.

```
998   REM ** Display a "1" die
1000  FOR X = 1 TO 5
1010   VLIN 1, 7 AT X 'Plot background
1020  NEXT X
1030  COLOR = 0
```

```
1040  PLOT 3, 4        'Plot dot
1090  RETURN
```

Program 11-4b. Subroutine to display a "1" die.

Programs 11-4a and 11-4b together make up a complete program to display the "1" die in two different colors with a brief delay in between. This is difficult to show on the printed page, so you will have to run it yourself.

Wouldn't it be nice to be able to display a die anywhere on the screen? This is easy with subroutines. All we need is to send values to our subroutine that specify where a corner of the die is to be. Using X and Y as the horizontal and vertical positions of the upper left corner of the die, we get the following subroutine to display the "1" anywhere on the screen.

```
998   REM ** Display a "1" die
1000  FOR I9 = 0 TO 4
1010   VLIN Y, Y + 6 AT X + I9
1020  NEXT I9
1030  COLOR = 0
1040  PLOT X + 2, Y + 3
1090  RETURN
```

Program 11-5. Drawing a "1" anywhere on the screen.

However, we must ensure that the values of X and Y place the entire die within the 40-by-40 graphics screen. That means that X may range from 0 to 35 and that Y is limited to values from 0 to 33 for our 5-by-7 die face.

Now the final piece of the puzzle will fit into place as soon as we write six subroutines, one for each of the six possible faces of a die. Numbering the first lines 1100, 1200, and so on to 1600 will help to identify the purpose of each subroutine.

```
1098  REM ** Plot one
1100  PLOT X + 2, Y + 3
1190  RETURN
1196 :
1198  REM ** Plot two
1200  PLOT X + 1, Y + 1
1210  PLOT X + 3, Y + 5
1290  RETURN
      .
      .
      .
1598  REM ** Plot six
1600  PLOT X + 1, Y + 1
1610  PLOT X + 1, Y + 3
1620  PLOT X + 1, Y + 5
1630  PLOT X + 3, Y + 1
1640  PLOT X + 3, Y + 3
1650  PLOT X + 3, Y + 5
1690  RETURN
```

Now we may remove lines 1030 and 1040 from our die-display subroutine. This leaves us with a very simple subroutine that will serve two functions for us: it will draw a die background and it may be used to erase a die from the screen.

```
998   REM ** Display a die background
1000  FOR I9 = 0 TO 4
1010   VLIN Y, Y + 6 AT X + I9
1020  NEXT I9
1090  RETURN
```

The display separates nicely into showing the background and plotting the spots. These two functions are now done with distinct subroutines. GOSUB 1000 displays the background. GOSUB 1100 through GOSUB 1600 may be used to display 1 through 6 spots on the die. The selection of one of these six subroutines is easily done with the ON . . . GOSUB statement.

```
200  ON R GOSUB 1100,1200,1300,1400,1500,1600
```

where *R* is the value for this roll of a die does it all for us. We can set the colors independently. Once a die has been drawn on the screen, we can set the color to 0 and call upon the background-display routine to erase the die, spots and all.

Problems for Section 11-3

1. Write a program to display a die face showing a "5" in the upper right corner of the graphics screen.
2. Write a program to display a random die face in the upper left corner of the screen.
3. Display a random die face, leave it for a few seconds, and then erase it.
4. Display two dice at random next to each other in the lower left corner.
5. Write a program to display a blinking die. Let it blink 10 times, then leave the display on the screen.
6. Display a few dice at random in random locations on the screen to simulate physically rolling the dice. Then display a pair of dice at random and leave them on the screen.

SIDELIGHT 11

Miscellaneous Aids to Graphics

....GET

GET may be used to accept a single character from the keyboard without any display on the screen. This is just what we need for full-screen Lo-Res graphics. GET waits until a character is struck before BASIC will proceed to the next program statement. This distinguishes GET from INKEY$. GET waits; INKEY$ does not wait. The only escape from a GET seems to be to hit the RESET key. If we enter CTRL-C, it is entered into our string variable. So, be a little careful about endless loops with GET.

INKEY$ is a function, but GET is a statement. It looks like this:

```
220  GET A$
```

If we use GET with a numeric variable, BASIC will think we are accessing a random-access file buffer.

....BEEP

With invisible keyboard interaction, it is sometimes nice to give the user some feedback in the form of sounds.

```
500  BEEP A, B
```

causes the Apple to emit a tone whose pitch is governed by A and that lasts for a time dependent on B. Both A and B range from 0 to 255 (zero is high pitch and short time). So a statement such as

```
250  BEEP ASC(A$), ASC(A$)
```

would change both according to the key pressed in our earlier GET statement.

Sometimes we need responses of more than one character. We can put INKEY$ or GET A$ in a loop and build up a string from individual characters. Another option exists in BASIC.

....INPUT$

INPUT$ can be used to request a specified number of characters from the keyboard without displaying the input.

```
520   A$ = INPUT$(2)
```

will wait until two characters have been entered and store them in A$. CTRL-C halts execution.

....External Terminal

Here is an option that helps us in a lot of situations. The SoftCard CP/M system provides for an external terminal. Now we can have upper/lowercase, 80-character lines, and graphics all in one operation. We do need a suitable communications interface card in SLOT #3 and a connecting cable to do this, but it is worthwhile for many applications.

With the external terminal we get both Lo- and Hi-Res graphics as well as a full screen of text at the same time. The SoftCard documentation provides full details for setting up a system in this way.

....SCRN

We might get into a situation where we would like our program to be able to distinguish the colors on the Lo-Res screen. We can easily do it with the SCRN function.

```
200  C = SCRN(X,Y)
```

returns the color of the block at position (X,Y) of the graphics screen. Values outside the range 0 to 39 for X and outside the range 0 to 47 for Y will evoke the

```
Illegal function call
```

error message. SCRN will return values for the text screen, but they are related to characters rather than to colors.

Chapter 12

Hi-Res Graphics

We saw in Chapter 11 that we could convert the screen into a graphics area containing up to 1920 little blocks. We could select from among 16 colors for each and every block individually.

Hi-Res graphics provides more dots and fewer colors. We can easily plot in a graphics area of 44,800 dots. We will also have four lines at the bottom of the screen for standard text display. If we do not require those four text lines, we can create a graphics screen of 53,760 dots. Instead of 16 colors we have 6 in Hi-Res.

12-1...Introduction to Hi-Res Graphics

There are just four commands for controlling the Hi-Res screen: HGR, HCOLOR, HPLOT, and TEXT. Hi-Res graphics is available in GBASIC only. A great deal of memory is required to work with Hi-Res graphics. So there is less memory available for program use in GBASIC. Any attempt to use HGR, HCOLOR, or HPLOT in MBASIC will be rewarded with the

```
Graphics statement not implemented
```

error message. Let's look at them all before we attempt to write our first program.

....The Hi-Res Graphics Screen (HGR)

The statement

```
100 HGR
```

prepares the computer for Hi-Res graphics work. When this statement is executed, the screen is divided into two parts. The top part is organized into 280 columns and 160 rows. This gives us the 44,800 dots mentioned earlier. The remainder of the screen is reserved for four lines of regular text display. Each dot in the graphics area is identified by its column and row. The columns are numbered from 0 to 279 going from left to right. The rows are numbered from 0 to 159 going from top to bottom. This is not the same as the conventional rectangular coordinate system widely used in mathematics, but this difference presents no great obstacle. The dot in the upper left corner is labeled (0,0). The dot in the lower right corner is labeled (279,159). The computer is restored to the conventional full text screen with the TEXT statement. Since TEXT immediately restores the text screen we will often code a little delay loop to allow time for the viewer to examine our handiwork.

There's more. Sometimes we may want to bring back something we drew earlier on the Hi-Res screen. We can do that with a value in the HGR statement.

```
100 HGR 2
```

does it for us. HGR 3 also restores any previous plotting. The difference between 2 and 3 is that 3 uses the four-line text window for graphics and 2 does not. This gives us the 53,760 dots mentioned earlier.

Think about that. We can restore a graphics screen. That means the screen was there all the time. TEXT simply allows us to look at the text screen. The text screen occupies a different portion of computer memory than the Hi-Res screen. Thus, when we use the TEXT command we see whatever was on the text screen before plus any text interaction that has taken place.

In fact, there are four values we may use to implement the Hi-Res screen. The values 0 through 3 represent different graphics modes. See Table 12-1.

SCREEN MODE	ACTION	
0	mixed graphics and 4-line text	clear the screen
1	full-screen graphics	clear the screen
2	mixed graphics and 4-line text	no clear
3	full-screen graphics	no clear

Table 12-1. Hi-Res screen values available in HGR.

Screens 1 and 3 use the full screen for graphics. This permits us to plot points in the range from 0 to 191 vertically. Thus the lower right corner becomes (279,191). In addition, for screens 0 and 1 we may specify a color value. The screen will be filled with the color we specify.

```
120 HGR 0, 5
```

clears the screen to orange and leaves four lines for text. Note that the cursor is not automatically moved into the text window as it is for Lo-Res. We need VTAB for that. VTAB A places the cursor at the beginning of the Ath line of the text screen. The lines are numbered 1 to 24. So we need VTAB 21.

....Hi-Res Colors (HCOLOR)

Even if we are working with a black-and-white monitor, we will have to pay attention to color. HGR presents us with the screen color of our choice. Further plotting is done in the same color as the screen. If we don't change the plotting color then our drawings will be invisible. We set the Hi-Res color with HCOLOR. We may use values in the range 0 to 12 with 12 causing the color to be the reverse of the color already on the screen. Thus, green and violet are exchanged, as are orange and blue. Similarly the whites and blacks replace each other. Drawing twice with HCOLOR set to 12 restores the original colors. The color names are shown in Table 12-2.

0	black	4	black	8	black1	12	reverse
1	green	5	orange	9	white1		
2	violet	6	blue	10	black2		
3	white	7	white	11	white2		

Table 12-2. Hi-Res color values.

The statement

```
12Ø  HCOLOR = 1
```

will set the high-resolution graphics color to green.

....Plotting Dots (HPLOT)

The statement

```
15Ø  HPLOT X, Y
```

will plot a dot at (X,Y) on the high-resolution graphics screen. The color used will be the last Hi-Res color set using HCOLOR. See Program 12-1.

```
1ØØ  HGR
11Ø  HCOLOR = 3
12Ø  HPLOT Ø, Ø
13Ø  HPLOT Ø, 159
14Ø  HPLOT 279, 159
15Ø  HPLOT 279, Ø
```

Program 12-1. Plot dots in the four corners.

Program 12-1 will place a white dot in each of the four corners of the graphics screen. At least that is what we would think. It turns out that there are some limits on what colors may be plotted where. A white dot

plotted in an odd-numbered column is really green (that is, if we select HCOLOR = 3) and a white dot plotted in an even-numbered column is really violet. For HCOLOR = 7 an odd column produces orange, while an even column plots as blue. Don't despair: we can easily produce white dots by plotting two dots next to each other, or by using colors 9 and 11. Now our dots will be wider, but they will be white. HCOLORs other than 3 and 7 do not exhibit this problem. We might want to change our program as shown in Program 12-2.

```
100  HGR
110  HCOLOR = 11
120  HPLOT 0, 0
130  HPLOT 0, 159
140  HPLOT 279, 159
150  HPLOT 279, 0
```

Program 12-2. Plot dots in the four corners (white this time).

....Lines in Hi-Res (HPLOT . . . TO)

There is no HLIN or VLIN statement in Hi-Res graphics. Instead we have a powerful extension of the HPLOT statement.

```
100  HPLOT X, Y TO X1, Y1
```

plots a line going from X,Y to X1,Y1. This is much more flexible than HLIN or VLIN. HPLOT . . . TO may be used for horizontal, vertical, and diagonal lines. We can easily extend Program 12-2 to place a nice border around the graphics screen. See Program 12-3.

```
100  HGR
110  HCOLOR = 11
120  HPLOT 0, 0 TO 0, 159
130  HPLOT 0, 159 TO 279, 159
140  HPLOT 279, 159 TO 279, 0
150  HPLOT 279, 0 TO 0, 0
```

Program 12-3. HPLOTting a border on the Hi-Res screen.

It is often desirable to have a border around a graphics display. So let's write a subroutine to do that right now. We could write the four statements from 120 to 150 from Program 12-3 as a single line by using three colons to create a multiple statement. However, HPLOT allows us to include multiple TOs. See Program 12-4.

```
598  REM ** Plot a border
600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
690  RETURN
```

Program 12-4. Subroutine to plot a border.

From now on we can use GOSUB 600 as calling for a Hi-Res border in the currently active HCOLOR. For HCOLORs 3 and 7 we would need to add a

vertical line at the left and right edges of the screen. The ability to continue plotting with multiple TOs is very nice.

HPLOT has one additional feature. Once a point has been plotted we can continue plotting in the same color with

```
HPLOT TO X, Y
```

This is useful for plotting in a "dot-to-dot" style.

So there we have it. HGR, HCOLOR, HPLOT, and TEXT give us tremendous power to draw figures on the Hi-Res graphics screen. When plotting white using HCOLOR 3 or 7 we must plot two horizontally adjacent dots to really get white.

For demonstration purposes let's write a program to display the Hi-Res colors. We need the usual HGR to prepare the graphics screen. Next we should label the colors. This can be done by displaying the color number just beneath each vertical color bar. In order to do this we have to prepare the four-line text window. In Lo-Res mode HOME clears only the bottom four lines. In Hi-Res graphics HOME clears the entire 24-line text screen. So now the cursor is hidden in the upper left corner of the text screen. Only the last four lines of the text screen are visible below the upper 160 lines of the Hi-Res graphics screen. As we noted earlier, VTAB 21 places the cursor at the first line of the window. We get a white border by setting HCOLOR to 11 and calling our border-plotting subroutine at 600. Next, for each color, we simply calculate some nice spacing and plot vertical bars. See Program 12-5.

```
   90   REM ** Display hi-res colors
   100  HGR
   106 :
   108  REM ** Prepare text window
   110  HOME : VTAB 21
   116 :
   118  REM ** White border
   120  HCOLOR = 11 : GOSUB 600
   166 :
   168  REM ** Colors 0 thru 11
   170  PRINT " ";
   180  FOR C = 0 TO 11
   185   HTAB 3*C + 2 : PRINT C;
   190   HCOLOR = C
-->200   B = 21*C + 12
-->210   FOR X = 1 TO 12
   220    HPLOT X + B, 5 TO X + B, 154
   230   NEXT X
   250  NEXT C
   300  PRINT
   320  HTAB 14 : PRINT "Hi-res colors"
   590  END
   596 :
   598  REM ** Plot a border
```

```
600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
690  RETURN
```

Program 12-5. Display Hi-Res colors.

Line 200 simply calculates a starting point for each color bar. Line 210 sets up a FOR loop to plot bars 12 dots wide.

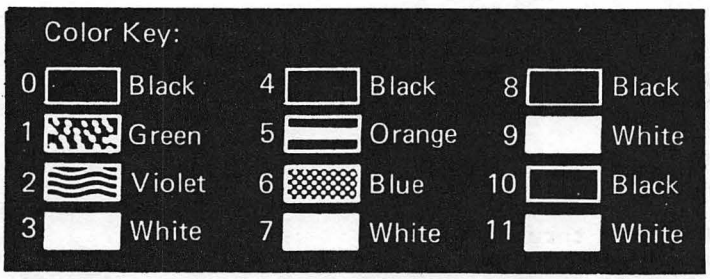

Figure 12-1. Execution of Program 12-5.

....SUMMARY

In Hi-Res graphics we have the tools to draw dots and lines in six colors. Values positioning points on the screen may range from 0 to 279 horizontally and 0 to 191 vertically.

We set up the screen with HGR. We may recall the most recent drawing or not, and we may access four text lines or not.

HCOLOR allows us to assign color values in the range 0 to 12. This gives us black, white, green, violet, orange, and blue.

With HPLOT we can plot dots or lines. HPLOT with TO is used to plot one line, plot several connected lines, or draw a line from the most recently plotted point.

Problems for Section 12-1

1. Modify the border-plotting subroutine of Program 12-4 so that it may also be used to plot a border around the full graphics screen. Require that the calling routine set the bottom edge by setting a variable to either 159 or 191.
2. Draw a bar graph picturing the following daily high temperatures for a one-week period.

DAY	TEMP
Sun	42
Mon	38
Tue	40
Wed	31
Thu	24
Fri	18
Sat	15

3. Draw a bar graph using two colors to represent the following daily high and low temperatures for a week.

DAY	HIGH	LOW
Sun	100	76
Mon	101	77
Tue	94	71
Wed	97	82
Thu	88	70
Fri	93	71
Sat	84	70

4. Draw a graph to show the following fluctuation in stock price for a five-day period.

DAY	PRICE
Mon	33-3/4
Tue	35-1/8
Wed	35
Thur	36-1/4
Fri	37-7/8

5. Use the data for problem 3 to draw a line graph with one line for high temperature and another for low temperature.

12-2...A Graphics Example

Now that we have the fundamentals we can work on making a drawing on the screen. We can simply code a series of HPLOT statements to draw lines and dots on the screen. Then, to add a line, we add an HPLOT statement. To remove a line, we remove an HPLOT statement. Using this method each new drawing is a new program.

A different approach is to write a little routine that HPLOTs lines using data stored in DATA statements. We can completely specify any line and any HCOLOR with five numbers—one for the color and two for each end of the line. To plot a single dot, simply make both ends of the line the same point. This makes the plotting routine very simple indeed. Once we perfect it, we may use it for any other drawing by simply changing the DATA. It is easy to terminate plotting by looking for a color value of −1. See Program 12-6.

```
198  REM ** Line plotting routine
200  READ C,X,Y,X1,Y1
210  IF C = -1 THEN 290
220   HCOLOR = C
230   HPLOT X, Y TO X1, Y1
240   GOTO 200
290  RETURN
```

Program 12-6. Plot drawings from DATA.

Program 12-6 is surprisingly short and simple. It is always very nice to come upon a short routine that does so much. This routine assumes that the Hi-Res graphics screen has been prepared. The real work in this drawing business is producing the data.

Just for fun let's draw a traffic light at an intersection of two roads. We should do the drawing on cross-section paper so that we can easily read the (X,Y) coordinates for each end of each straight line in the drawing. See Figure 12-2 on the bottom of the following page. The first three lines are numbered as examples in Figure 12-2. Line 1 is represented by the data 11,100,20,145,60. Line 2 is represented by the data 11,162,75,250,153. And line 3 is represented by the data 11,60,17,125,75. In a similar fashion we obtain the rest of the data shown in Program 12-7a. It is a good idea to insert REMs to separate the data into sensible groups.

```
100  HGR 1
110  GOSUB 200
190  END
196 :
198  REM ** Line plotting routine
200  READ C,X,Y,X1,Y1
210  IF C = -1 THEN 290
220   HCOLOR = C
```

```
230   HPLOT X, Y TO X1, Y1
240   GOTO 200
290  RETURN
996 :
998   REM ** Line data
999   REM ** The road
1000  DATA 11,100,20,145,60
1002  DATA 11,162,75,250,153
1004  DATA 11,60,17,125,75
1006  DATA 11,142,90,210,151
1008  DATA 11,200,19,145,60
1010  DATA 11,125,75,80,109
1012  DATA 11,245,14,162,76
1014  DATA 11,142,90,80,137
1038  REM ** The light standard
1040  DATA 11,140,10,150,10
1042  DATA 11,150,10,150,40
1044  DATA 11,150,40,140,40
1046  DATA 11,140,40,140,10
1048  DATA 11,145,40,145,55
1050  DATA 11,145,40,148,55
1990  DATA  -1,0,0,0,0
```

Program 12-7a. Draw a traffic light using data and Hi-Res.

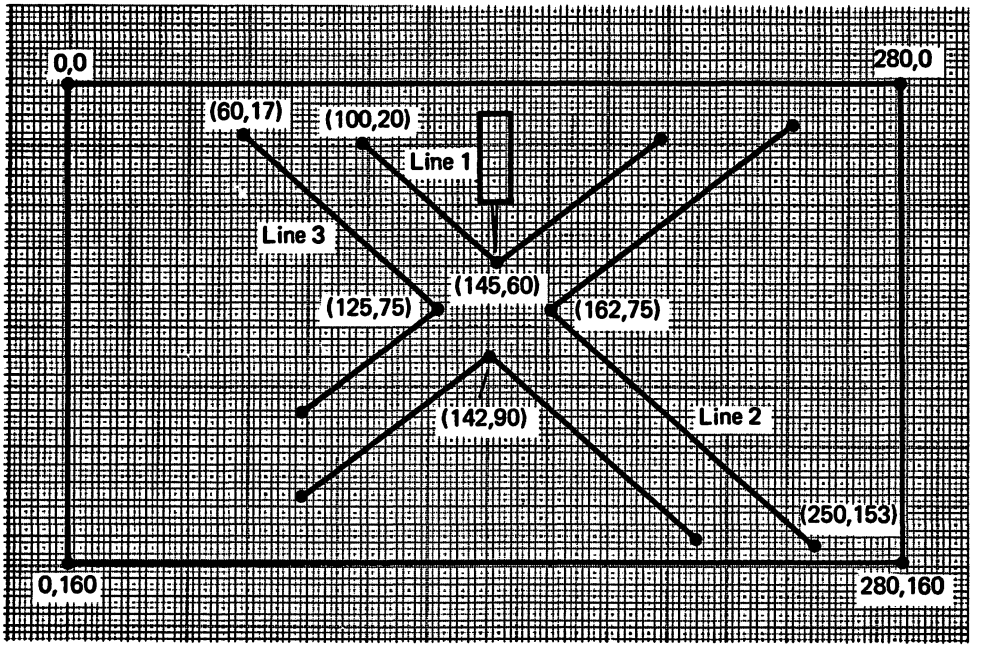

Figure 12-2. Drawing of a traffic light on cross-section paper.

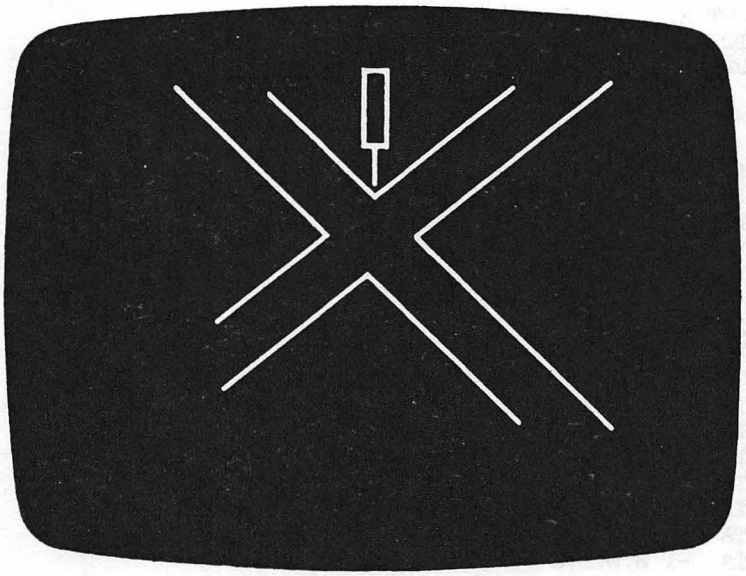

Figure 12-3. Execution of Program 12-7a.

Well, this is a good start. But we certainly ought to put in the three lights. Let's develop a routine to draw a circle of any size anywhere on the screen. Then we can place three of them in our traffic light. We can draw circles using the Pythagorean theorem or we might use sines and cosines. Let's use the Pythagorean theorem now.

The Pythagorean theorem says that for any point (X,Y) on a circle of radius R we have

$$X^2 + Y^2 = R^2$$

Refer to Figure 12-4 on the following page.

So we can get points on a circle by solving for Y and calculating values for a range of values of X.

$$Y = \sqrt{R^2 - X^2}$$

In order to graph the whole circle we should use both the positive and negative square roots for Y and use both positive and negative values for X. Let's draw a circle of radius 10 centered at the point (70,80). In order to make this as versatile as possible, let's use variables for the radius and the coordinates of the center of the circle. Consider Program 12-7b.

```
90   REM ** Draw a circle
100  HGR 1
170  HCOLOR = 3 : R = 10
180  XO = 70 : YO = 80
300  FOR X = -R TO R STEP .4
310   Y = SQR(R^2 - X^2)
```

```
320   HPLOT XO + X, YO + Y
330   HPLOT XO + X, YO - Y
340  NEXT X
```

Program 12-8. Draw a circle for the traffic light.

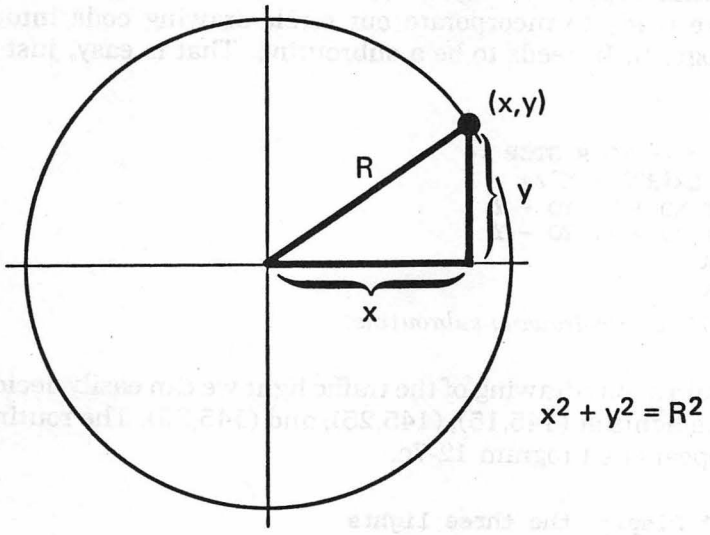

Figure 12-4. Coordinates on a circle.

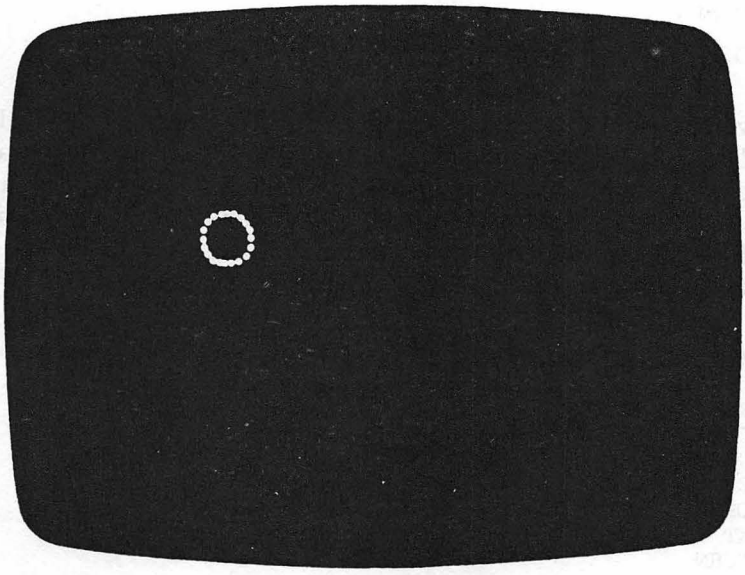

Figure 12-5. Our first circle in Hi-Res.

It would be a good idea to type this program in and experiment with different increments in the FOR loop and different sized circles at different places on the screen. The greater the increment for X the faster the drawing goes and vice versa. Of course, if we use too large an increment then we have large gaps in the figure. (See Figure 12-5 on page 183.)

Now we are ready to incorporate our circle-drawing code into our traffic light program. It needs to be a subroutine. That is easy, just add 390 RETURN.

```
300  FOR X = -R TO R STEP .4
310   Y = SQR(R^2 - X^2)
320   HPLOT XO + X, YO + Y
330   HPLOT XO + X, YO - Y
340  NEXT X
390  RETURN
```

Program 12-7b. Circle-drawing subroutine.

Looking again at our drawing of the traffic light we can easily decide to center the three lights at (145,15), (145,25), and (145,35). The routine to control this appears as Program 12-7c.

```
398  REM ** Display the three lights
400  HCOLOR = 11 : R = 3
410  XO = 145 : YO = 15 : GOSUB 300
420             YO = 25 : GOSUB 300
430             YO = 35 : GOSUB 300
490  RETURN
```

Program 12-7c. The three lights.

You will have to type this one in, too, to get a feel for the result. Now if we just had red, yellow, and green as Hi-Res colors we could go the distance. Let's just settle for a blinking "yellow" light. Using orange will be a reasonable compromise. We can put in little delay loops to make it realistic. Again we do this with subroutines. We control the blinking light with Program 12-7d, while the actual light appears in Program 12-7e.

```
498  REM ** Blinking "yellow"
500  XO = 145 : YO = 25
510  FOR T = 1 TO 30
520   HCOLOR = 10 : GOSUB 600
525   FOR I9 = 1 TO 100 : NEXT I9
530   HCOLOR =  5 : GOSUB 600
535   FOR I9 = 1 TO 500 : NEXT I9
580  NEXT T
590  RETURN
```

Program 12-7d. Control the blinking light.

```
598  REM ** Display the actual light
600  FOR I9 = -1 TO 1
610   HPLOT XO - 1, YO + I9 TO XO + 1, Y9 + I9
620  NEXT I9
690  RETURN
```

Program 12-7e. The blinking traffic light.

This completes a sketch of a traffic light at an intersection of two roads.

```
100  HGR 1
110  GOSUB 200
120  GOSUB 400
130  GOSUB 500
190  END
196 :
198  REM ** Line plotting routine
200  READ C,X,Y,X1,Y1
210  IF C = -1 THEN 290
220   HCOLOR = C
230   HPLOT X, Y TO X1, Y1
240   GOTO 200
290  RETURN
296 :
298  REM ** A circle
300  FOR X = - R TO R STEP .4
310   Y = SQR(R^2 - X^2)
320   HPLOT XO + X, YO + Y
330   HPLOT XO + X, YO - Y
340  NEXT X
390  RETURN
396 :
398  REM ** Display the three lights
400  HCOLOR = 11 : R = 3
410  XO = 145 : YO = 15 : GOSUB 300
420             YO = 25 : GOSUB 300
430             YO = 35 : GOSUB 300
490  RETURN
496 :
498  REM ** Blinking "yellow"
500  XO = 145 : YO = 25
510  FOR T = 1 TO 30
520   HCOLOR = 10 : GOSUB 600
525   FOR I9 = 1 TO 100 : NEXT I9
530   HCOLOR =  5 : GOSUB 600
535   FOR I9 = 1 TO 500 : NEXT I9
580  NEXT T
590  RETURN
596 :
598  REM ** Display the actual light
600  FOR I9 = -1 TO 1
610   HPLOT XO - 1, YO + I9 TO XO + 1, Y9 + I9
620  NEXT I9
690  RETURN
```

```
996 :
998   REM ** Line data
999   REM ** The road
1000  DATA 11,100,20,145,60
1002  DATA 11,162,75,250,153
1004  DATA 11,60,17,125,75
1006  DATA 11,142,90,210,151
1008  DATA 11,200,19,145,60
1010  DATA 11,125,75,80,109
1012  DATA 11,245,14,162,76
1014  DATA 11,142,90,80,137
1038  REM ** The light standard
1040  DATA 11,140,10,150,10
1042  DATA 11,150,10,150,40
1044  DATA 11,150,40,140,40
1046  DATA 11,140,40,140,10
1048  DATA 11,145,40,145,55
1050  DATA 11,145,40,148,55
1990  DATA  -1,0,0,0,0
```

Program 12-7. The completed traffic light program.

There is always room for improvement. Program 12-7 can draw only one traffic light of one size at one spot on the screen. We might convert the data so that every point is calculated in terms of a single starting point. Then we will be able to move the traffic light to any point that keeps the entire figure on the screen. Maybe we could draw cars racing around the screen. Our border-drawing subroutine could be used to nicely frame our picture. We could determine the data for many figures and save it in data files on disk. Then we will have a whole library of figures to use for later graphics applications. The possibilities are truly unlimited.

We have presented an introductory treatment of figures in Hi-Res. A great deal can be done with the tools available in BASIC. However, to produce high-speed action with collisions and explosions, programmers often resort to assembly language programming.

....SUMMARY

Just four BASIC keywords open the way to very powerful Hi-Res color graphics. HGR 0 gives us a screen with 280 columns and 160 rows. HGR 1 activates an additional 32 rows at the bottom of the screen. Colors in the range from 0 to 12 are available with HCOLOR. In order to get white we must plot the points (X,Y) and (X + 1,Y). Violet and blue appear only in even-numbered columns, while green and orange may be plotted only in odd-numbered columns. HPLOT . . . TO plots single points or line segments in any orientation. We restore the text screen with the TEXT statement. We have developed a routine that allows us to specify a drawing in terms of a collection of line segments. For each segment we need only supply the color and the endpoints.

Problems for Section 12-2 .....

The possibilities for drawing figures on the screen are literally unlimited. We can only begin to make some suggestions leading you into problems of interest. Let your imagination plunge you into exciting graphics demonstrations.

1. Adjust the data in the traffic-light-drawing program so that each set of data is calculated in terms of a fixed starting point. Using (X0,Y0) as (100,20), the first three data lines will be

```
1000  DATA  11,0,0,45,40
1005  DATA  11,62,55,150,133
1010  DATA  11,-40,-3,25,55
```

   Now the control routine can select a variety of starting points and draw the traffic light anywhere on the screen with just one plotting subroutine.
2. Supply data to draw a sailboat on the screen using the plotting routine of Program 12-7a.
3. Supply data to draw a simple TV set on the screen using the plotting routine of Program 12-7a.
4. Write a program that illustrates the raising of a flag on a flagpole. Plotting the flagpole is straightforward. By successively plotting a flag on ever-increasing heights of the pole, the flag will appear to be raised. Note that you must erase the previous flag as you plot each new one. This can be done by erasing only a section of the previous flag.

12-3...Hi-Res Graphs from Formulas

Figures that can be described using a formula are easy to graph. There are many examples from mathematics.

....Cartesian Coordinates

Let's develop a method for adjusting the X and Y values in the conventional Cartesian coordinate system for plotting on the screen. We would like to move the (0,0) point near the center of the screen and alter the orientation for Y values so that they are increasing up instead of down. Suppose we specify that the point (XO,YO) on the Hi-Res screen shall represent the point (0,0) in a Cartesian system. Typically we might place the origin of a graph near the center of the screen. So the point (XO,YO) may often be (140,80). The X conversion is easy. We simply want to move each plotted point to the right on the screen. The Y conversion requires

that we turn the graph "upside down." So the point

(X1,Y1)

in the conventional Cartesian coordinate system becomes

(XO + Xl, YO − Yl)

on the Hi-Res screen.

It would be nice to plot the X and Y axes right on the screen. A very simple subroutine will do this for us. Again, here we can plot the vertical line two dots wide.

Plotting points that fit a formula is straightforward enough. For our first graphs we might do just functions. This is a good application for a DEFined function. We can start with the simplest of all functions:

```
Y = X
```

We define this function with

```
160  DEF FNF(X) = X
```

We need a subroutine that scans all possible values for X and determines if the Y value is on the screen. If it is, then the routine should plot the point. If not, then the routine should simply try the next X value. All of this is done in Program 12-9.

```
 90   REM ** Plot a function
100   HGR Ø : HOME
116 :
118   REM ** White border
120   HCOLOR = 11 : GOSUB 600
126 :
128   REM ** Plot axes (still white)
130   GOSUB 700
146 :
148   REM ** Draw the graph
150   HCOLOR = 1      'Arbitrarily select green
-->160  DEF FNF(X) = X 'Define the function
180   GOSUB 200       'Plot the function
190   END
196 :
198   REM ** Plot a function
200   FOR X1 = -138 TO 138
220    Y1 = FNF(X1)   'Use the function
230    X  = 140 + X1
240    Y  =  80 - Y1
250    IF Y > 2 AND Y < 157 THEN HPLOT X, Y
270   NEXT X1
290   RETURN
596 :
598   REM ** Plot a border
-->600  HPLOT Ø,Ø TO Ø,159 TO 279,159 TO 279,Ø TO Ø,Ø
690   RETURN
696 :
```

```
698  REM ** Plot axes for graphing
700  HPLOT 3,80 TO 276,80
710  HPLOT 140,3 TO 140,156
790  RETURN
```

Program 12-9. Plot a function in Hi-Res.

This program is set up for the mixed graphics-text screen. We could easily convert the subroutines at lines 600 and 700 to plot for either full or part screen using an S0 value that could be 191 for full screen and 159 for part screen. In addition we might want to move the axes so that the point (0,0) is not in the exact center. As we discussed earlier this could be done by passing (XO,YO) to the axes-plotting subroutine as the Hi-Res coordinates of the (0,0) point for the Cartesian graph.

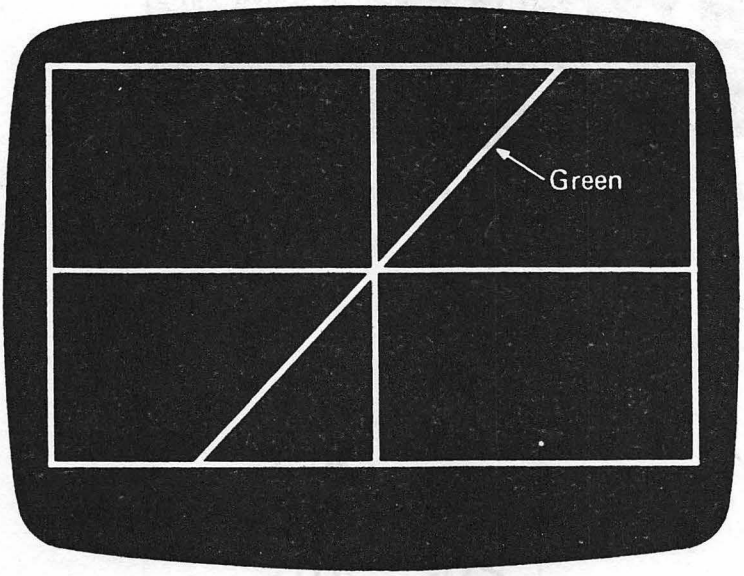

Figure 12-6. Execution of Program 12-9.

Now it is a very simple matter to replace line 160 of Program 12-9 with the function of our choice. With a little experimentation we can produce attractive displays without the tedium of arduous calculations. Values of sine are in the range from −1 to +1, so we need to scale up to get values that will show up nicely on the screen. We select a scale factor of 50 to get an idea of what it looks like. Let's demonstrate this with

```
160  DEF FNF(X) = 50 * SIN(X/10)
```

Now let's do a circle with sine and cosine.

Referring to Figure 12-8 we see that the X distance from the center of the circle is Rcos(G) and the Y distance is Rsin(G). Following our pattern

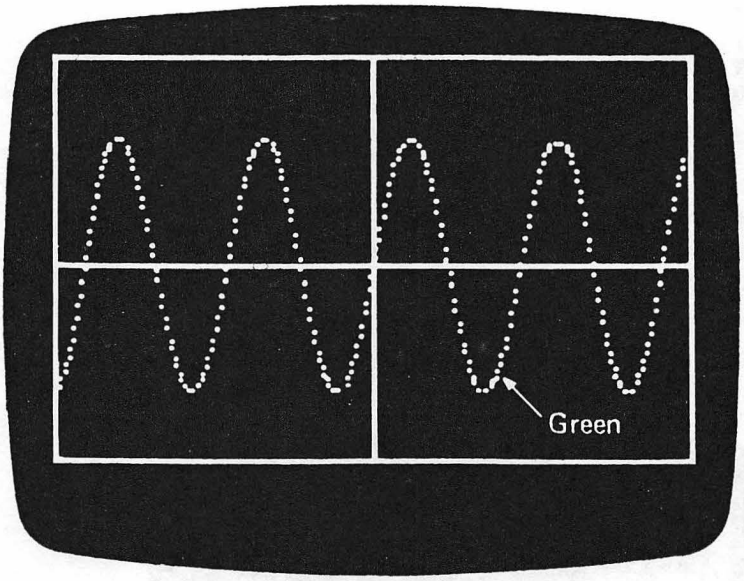

Figure 12-7. Program 12-9 with sine function.

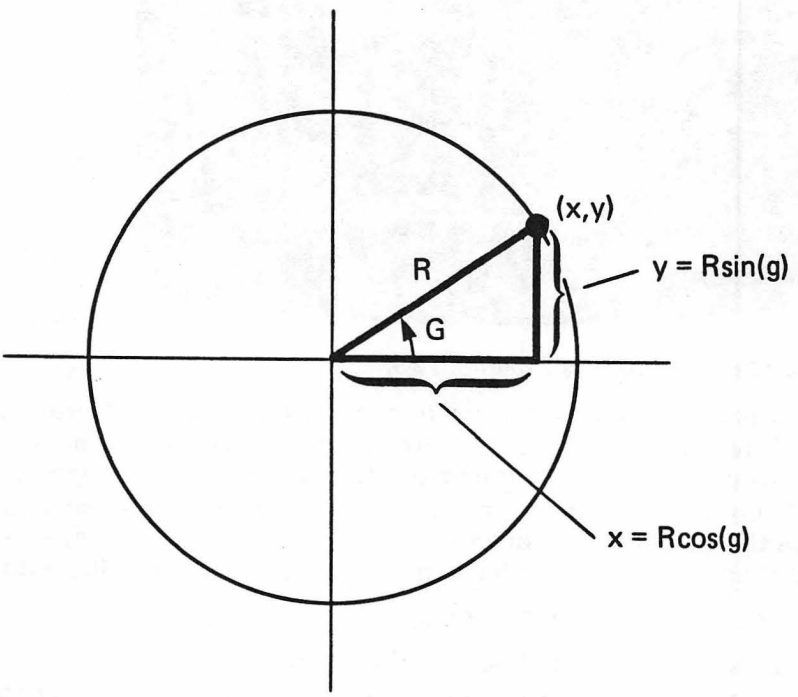

Figure 12-8. Coordinates on a circle using sine and cosine.

for Cartesian coordinates again we get the screen coordinates of the point (Rcos(G), Rsin(G)) as

```
(XO + R * COS(G), YO + R * SIN(G))
```

Finally, we obtain a range of points on a circle by rotating the angle G through 2π radians (about 6.29) or 360 degrees. This is best done with a FOR loop. We can experiment with the increment to get a smooth drawing in a reasonable amount of time.

Let's start with a circle of radius 10 centered at the point (70,70). Consider Program 12-10.

```
90   REM ** Draw a circle
100  HGR 0
150  R = 10
160  XO = 70 : YO = 70
170  HCOLOR = 11
300  FOR G = 0 TO 6.28 STEP .2
310   X1 = R * COS(G)
320   Y1 = R * SIN(G)
320   HPLOT XO + X1, YO - Y1
340  NEXT G
```

Program 12-10. Draw a circle.

....SUMMARY

We can plot a mathematical function by simply scanning the X-value range on the Hi-Res screen and calculating each Y value. The program needs to verify that each point is actually on the screen. By using a DEFined function we have been able to write a generalized program to display functions of our choice.

Problems for Section 12-3 .....

1. We sometimes need to experiment with a function. Try to plot $2X^3 - 2X^2 + 3X - 5$. It should be apparent that most of the Y values are off the screen. We can scale the Y dimension down by dividing the value of Y by a large number—say 100000. Try it.
2. Try $X^2 + 50X - 450$. Divide by 100. Remembering that the scale is distorted, we can gain a lot of insight into how a function performs.
3. Replace the circle-drawing routine in the traffic light of Program 12-7b with the circle-drawing routine of Program 12-10.

12-4...Polar Graphs

Polar equations often produce interesting graphs. One of the reasons we don't draw many polar graphs by hand is that they take too much tedious

calculation involving trigonometric functions. We can easily produce the graphs without the tedium by using Hi-Res graphics and letting BASIC do the calculations.

We may use

R = 1 − 2cos(G)

as an example equation. Using sines and cosines we get the X and Y coordinates as follows:

X = Rcos(G)

and

Y = Rsin(G)

where G is the central angle in radians. To obtain a full graph the central angle must sweep through a full 360 degrees or 2π radians, just as in Program 12-10. We can get about 60 points by using STEP .1 in a FOR . . . NEXT loop. Since the point (0,0) is in the corner of the Hi-Res screen we need to adjust the starting point to keep the figure in view.

To make our figures as large as possible we can use HGR 1 to obtain full-screen graphics. In this situation there is no text display, so after we have had a chance to examine the graph, we will need to type TEXT "in the blind" to get back the text screen and see our program. Now we have to think about adjusting the X and Y values on the conventional Cartesian coordinate system for plotting on the screen. This is exactly the same conversion we carried out in Section 12-3. So the point (X9,Y9) in the conventional Cartesian coordinate system becomes (XO + X9, YO − Y9) on the Hi-Res screen. Where the point (XO,YO) defines the point on the screen where we want the Cartesian point (0,0) to be located. Again we have shifted to the right and turned the graph upside down.

It would be nice to display a polar axis right on the screen with the graph. We can easily plot a line beginning at the point (0,0) and extending to the right edge of the screen. Placing the polar axis on the screen will clearly locate the graph for us.

Once we have a working program, it will be a simple matter to plug in other equations. In this way we can look at dozens of graphs in the time it would take to draw a single graph by hand. It is interesting to watch the figures as they are formed on the screen. Drawing a polar graph by hand, like typing a 100-page paper on a portable typewriter, is one of those things everybody ought to do once in a lifetime.

Our program separates nicely into three packages: the control routine, the polar-axis-plotting routine, and the graph-plotting routine. Let's work on them in that order.

In the control routine we set up the full graphics screen with HGR 1. Setting the color is easy. Next we define the X and Y axes and call the polar-axis-plotting subroutine. Polar graphs plotted true size are usually

very small. So we should provide a scaling factor to produce a larger graph. We define the radial scale in RS. In the actual plotting subroutine we will be arranging for the central angle to range through a full rotation of 2π radians. But we might like to control the step size in the control routine. Thus we set the value of ST here. Finally we call the plotting subroutine. That is all there is to it. See Program 12-11a.

```
   98   REM ** Control polar graphing
   100  HGR 1
   110  HCOLOR = 6
-->120  XO = 139 : YO = 95
   130  GOSUB 1000 'Plot polar axis
-->140  RS = 45 : ST = .1
   160  GOSUB 200  'Plot the graph
   190  END
```

Program 12-11a. Control routine for polar graphing.

In Program 12-11a line 120 sets the axes as close to the center of the screen as possible. Line 140 sets the radial scale at 45 and the step size at .1.

The easy one is the polar-axis-plotting routine. All we do is HPLOT a line from the point (XO,YO) to the right edge of the screen. That takes one statement. See Program 12-11b.

```
   998   REM ** Plot polar axis
   1000  HPLOT XO,YO TO 279,YO
   1090  RETURN
```

Program 12-11b. Draw a polar axis.

Now let's look at the actual plotting subroutine. We need to provide for the angle to sweep a full rotation. This is done with a FOR . . . NEXT loop ranging from 0 to 6.29. The number of points we want plotted may well depend on the size of the graph. We may want more points for larger graphs. So we let the calling routine establish the STep size. We can then experiment with each new equation until we get a nice graph. A large step size will not give enough points of the graph; too small a step size will take too long to plot. See Program 12-11c.

```
   198  REM ** Plot polar graph
   200  FOR G = 0 TO 6.29 STEP ST
-->210   R1 = 1 - 2 * COS(G)
-->220   R9 = RS * R1
-->230   X9 = R9 * COS(G) : Y9 = R9 * SIN(G)
   240   HPLOT XO + X9, YO - Y9
   250  NEXT G
   290  RETURN
```

Program 12-11c. Polar-graph-plotting subroutine.

In Program 12-11c, the polar equation is defined in line 210, the scaling factor is implemented in line 220, and the Cartesian X and Y values are calculated in line 230. It will be a simple matter to change the polar equation by changing line 210. We must be aware that other polar equations may contain points that are off the screen. We can test for out-of-range values and skip the plotting for those points. Further, we must be alert for equations that may cause BASIC to attempt to divide by zero. See Figure 12-9 for a trial run of this program.

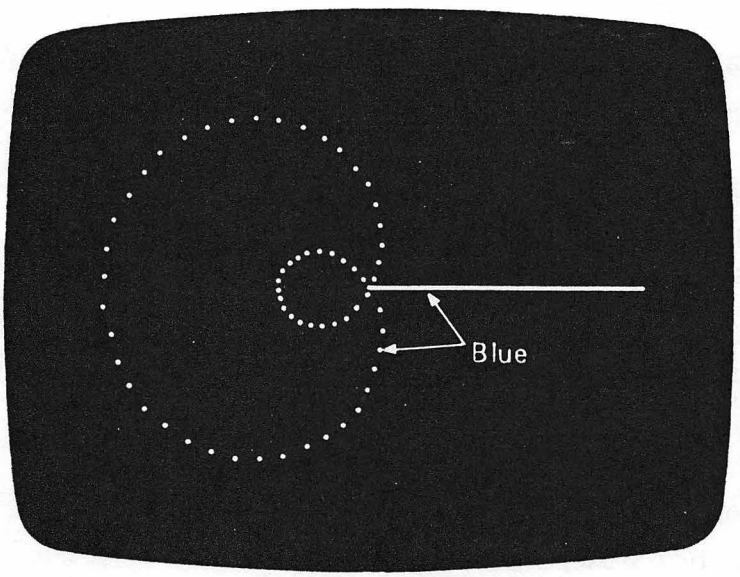

Figure 12-9. Execution of Program 12-11a,b,c.

Problems for Section 12-4

1. We can easily plot a circle with our polar-equation-plotting program using the polar equation R = 1. Do this.
2. There are lots of interesting polar graphs. Graph any of the following:

   (a) $R = 1 + 2\cos(G) - 3\sin(G)^2$
   (b) $R = 3 + \sin(3G)$
   (c) $R = 2 + \sin(2G)$
   (d) $R = \sin(G) + \cos(G)$

3. Many polar equations produce nice graphs, but they will cause our polar-plotting program to fail. Some points will lie off the graphics screen. Some values of G will cause division by zero. We can easily test whether a point is on the screen between lines 230 and 240 of

Program 12-11c. If a point is off the screen, don't plot it. If the formula we enter at line 210 has an indicated division then we can put in a test between lines 200 and 210. If the current value of G would cause such a zero division, don't even execute line 210. Adding these features will enable you to draw graphs for any of the following:

(a) Rcos(G) = 1
(b) R = 1 + Rcos(G)
(c) R = tan(G)
(d) R = 2G (make the scale 1 and make G range from −50 to 50)
(e) R = 2/G (scale 25 and G from −10 to 10)

SIDELIGHT 12

HSCRN

We may be interested in whether a point on the Hi-Res screen has been plotted.

```
210  P = HSCRN(X,Y)
```

will tell us. The value of P will be 0 if no point has been plotted and −1 if it has. Unlike SCRN with Lo-Res we cannot determine the color using HSCRN.

It can be very instructive to experiment with HSCRN. Issue an HGR command to paint the screen green. Then examine points with odd and even values for X. For X even we get HSCRN equal to 0 and for X odd we get HSCRN equal to −1. Green is plotted only in odd-numbered columns. Similarly, we will find orange in odd-numbered columns, while violet and blue are plotted in even-numbered columns. For HCOLOR 9 and 11 we find that both odd and even values of X are plotted. But for HCOLOR 3 and 7 only dots with even values for X are plotted.

Appendix A

Applesoft and SoftCard BASIC

....Applesoft Features Not Included

FLASH, IN#, PR#, HIMEM and LOMEM, DRAW and XDRAW, SCALE, and ROT are not available in SoftCard BASIC.

In addition (or subtraction), the ESC A, B, C, and D, and ESC I, J, K, and M editing features are replaced by the powerful EDIT Mode. EDIT Mode is discussed in Appendix B.

If you really want FLASH, just POKE −4046,127. (With some 80-column cards this will not work.) All characters with ASCII values in the range 64 to 127 and 192 to 255 will flash, while all others will appear as inverse characters. (See Appendix D for the ASCII chart.) This will cause all letters plus the characters @, [, \,], ^, and _ to flash. The way back is with the NORMAL statement. None of this matters much for applications that use an external terminal.

Several of the functions provided by IN# and PR# are standard features of the SoftCard. We may use a printer by plugging it into SLOT #1. In CP/M, CTRL-P acts as a toggle switch to turn the printer output on and off. If the printer output is off, CTRL-P turns it on. If the printer output is on, CTRL-P turns it off. In BASIC we send output to the printer with LPRINT, LPRINT USING, or LLIST.

We may use an external terminal by plugging the appropriate hardware into SLOT #3. We can tailor the CP/M software to our specific terminal using CONFIGIO in BASIC.

The SoftCard documentation includes a little routine that may be used to boot another disk. This takes the place of IN#6 in Applesoft,

assuming the disk is in SLOT #6. Of course, we can always shut the computer off, install the new disk, and turn the computer back on again.

We can achieve the purpose of HIMEM by using the /M: option upon invoking BASIC in the first place.

```
MBASIC /M:32767
```

limits BASIC to memory up to address 32767. Memory is set aside for special purposes.

Shape-table graphics are simply not provided.

....Features Included to Support Applesoft

HOME clears the text screen and places the cursor in the upper left corner.

The HTAB statement is used to provide absolute horizontal cursor positioning. HTAB X places the cursor at position X. The positions of the line are numbered beginning with 1. VTAB does the same thing for vertical positioning. The lines are numbered from 1 to 23. Note that HTAB and VTAB are statements and are not to be used in a PRINT statement.

INVERSE displays all further text as dark letters on a light background. NORMAL restores the familiar text display.

We may erase lines from a program with DEL or DELETE as we prefer. Both work, though DEL entered in a program statement will list as DELETE. To delete several lines we may use the comma as in Applesoft. DEL 1,120 will eliminate lines 1 through 120. LIST also allows the comma.

....Statements That Behave Differently

GR is used to determine whether we get four text lines at the bottom of the screen or not. GR 1 is just like GR in Applesoft. GR 2 invokes full-screen Lo-Res graphics. Further we may include a second parameter to clear the screen to a desired color.

```
210  GR 1, 8
```

will clear the full screen to brown.

COLOR in SoftCard BASIC may be treated as simply a special variable, rather than as a keyword. Thus, we may code statements such as

```
305  COLOR = COLOR + 1
```

We may not set COLOR values outside the range of 0 to 15.

Hi-Res graphics are available only in GBASIC. GBASIC allows far less user memory than MBASIC. There is no HGR1. The HGR statement may include a parameter to set mixed graphics and text or full-screen graphics and whether or not the screen is cleared. Further, a second parameter may be used to paint the screen the HCOLOR of our choice. HGR 0 is the same as HGR in Applesoft. HGR 1 (not to be confused with HGR1 in Applesoft)

clears the screen to black and enables us to use the bottom four text lines for graphics work. HGR 2 restores the graphics screen in mixed graphics and text, while HGR 3 restores the full graphics screen. HGR 0 and HGR 1 may be followed by a color number from 0 to 11 to clear the screen to the color of our choice. HCOLOR 12 reverses whatever color is already plotted.

HCOLORs 0 through 7 behave almost as in Applesoft. White HCOLORs 9 and 11 have been added and plot white whether the column is odd or even. Black HCOLORs 8 and 10 are paired up with 9 and 11.

FOR . . . NEXT in versions 5.0 and later of SoftCard BASIC will not execute if the initial value of the loop is outside the range specified by the last value and the STEP. In Applesoft FOR . . . NEXT always executes at least once.

IF . . . THEN in SoftCard BASIC permits an ELSE clause not allowed in Applesoft.

The logical operators are quite different. In Applesoft

```
2 OR 4
```

is evaluated as 1. So is

```
2 AND 10
```

On the other hand, SoftCard BASIC evaluates logical expressions bit by bit. Thus,

```
2 OR 4
```

evaluates as 6 because both the 2 bit and the 4 bit are set, while

```
2 AND 10
```

evaluates as 2 because the 2 bit is the only bit set in both 2 and 10.

NOT simply reverses all bits in the number. All ones go to zero and all zeros go to one. Thus,

```
NOT 127
```

becomes −128.

....Features in SoftCard BASIC Not Found in Applesoft

We can generate tones with BEEP.

```
210 BEEP 5, 15
```

will sound a tone whose pitch is related to 5 and whose duration is related to 15. These values may range from 0 to 255. These values are not mathematically coordinated with musical notes.

HSCRN (X, Y) returns a −1 or a 0 as the point (X,Y) is or is not plotted on the Hi-Res screen. It will not report the color number.

PRINT USING provides tremendous flexibility for formatting results in a display. We can easily specify neatly organized reports with right-

justified columns of figures all rounded to the same precision right in the PRINT statement. It is interesting to note that the pattern in a PRINT USING statement can be a string variable.

```
200  PRINT USING "  #####.## "; X
```

becomes

```
50   A$ = "  #####.## "
200  PRINT USING A$; X
```

This feature opens the way for tremendous flexibility for formatting results. The string pattern could be DATA in a program, or even stored in a file on disk.

WIDTH N sets the screen width to N characters.

We can change the line numbering sequence with RENUM.

```
RENUM  300, 125, 20
```

Will map the existing line numbers into new line numbers according to the three numbers given. In this case the old line number 125 will become 300 and succeeding line numbers will have intervals of 20. Lines cannot be moved out of sequence with RENUM.

A program line editor is available in EDIT Mode. See Appendix B for a complete description.

LINE INPUT reads an entire line of data without regard to commas. Everything up to cr-lf is accepted in a statement such as

```
220  LINE INPUT A$
```

No question mark is displayed.

We may work with double-precision numeric values with up to 16-digit precision. Such variables are designated by appending a number sign (#) to any legal variable name or any constant.

The WRITE statement displays values close packed and separated by commas. Strings are surrounded by quotes. We may also write data into a file with WRITE #.

```
100  X = 123456 : A$ = "The hour is late"
110  WRITE X, A$
RUN
123456,"The hour is late"
```

Files are managed with distinct statements designed for this purpose. Files are identified by channel number for easy organization of multiple file access in a single program.

One of the advantages of using Microsoft BASIC on any computer is that most of the features will be transportable to any other machine that supports the same language.

Appendix B

EDIT Mode

Touch typing is wonderful. EDIT Mode is even more wonderful. When we first "bring up" BASIC-80 the program area of the computer is a blank sheet of paper or an empty chalkboard. As soon as we type program statements in, it's not blank or empty anymore. There is a program that we might want to modify. We might have made a typing error, or we might want to change a statement to change its effect in the program. In many cases an error in a program line is simply a single mistyped character. Without EDIT Mode we are required to retype the entire line. And there is nothing to stop us from making more typing errors on the next try. With EDIT Mode, we simply use a few easy-to-remember commands to change only the offending character or characters.

In order to successfully master the world of EDIT Mode it is best to perform each exercise directly on a computer. This is fingertip learning. Read through this section to become familiar with the overall scheme of things and then go through it again following along on a keyboard. Go beyond the exercises; try each feature several times on your own program examples. After a bit of practice, many people find that they can simply tell their fingers what to do. They do the rest. The time spent mastering the editor now will be repaid many times in your programming experience. This point cannot be overemphasized.

....EDIT Mode Commands

BASIC-80 offers a set of commands that enable us to make changes in a single line. As we process a line making changes, the cursor follows along

to show us just where we are on the line. Our changes may be displayed for us to see, but the commands are not. At first this may seem a little difficult for the beginner, but soon even the timid user will be comfortable with the process. Let's begin by entering Program B-1.

```
100  PRINT "This is a EDIT example."
110  PRINT "We will bee makng a few changes."
120  PRINT "If we folllow alone with the,"
130  PRINT "then we will becom editting experts."
```

Program B-1. An EDITing example.

Type the program exactly as it appears so that you may folllow alone making exactly the changes outlined here. It is important for you to have access to a computer for this, as we cannot easily demonstrate the dynamic nature of the editing process on paper. The best we could do would be to insert clear plastic overlays in a book. But there really is no substitute for actual experience.

We get at line 100 by typing:

```
EDIT 100
```

BASIC goes into EDIT Mode by displaying the line number and waiting for us to make the next move. Probably the simplest move to make is to move the cursor.

....Move the Cursor (Press the Space Bar)

Pressing the space bar causes the editor to display the next character of the current line. Pressing it again produces another character. The keyboard repeat feature may be used for this. We step through the line until we expose the "a". At this point we want to insert the letter "n".

....Insert

The insert command is "I". As with BASIC, either "i" or "I" is valid. The character to insert is "n". We do that by keying "in" or "In". The character inserted will be upper- or lowercase exactly as we type it here. The "I" will not appear on the screen. The "n" will appear on the screen. At this point we may leave EDIT Mode for this line by pressing RETURN. The editor responds by displaying the rest of the line and returning to BASIC. Line 100 now reads

```
100  PRINT "This is an EDIT example."
```

Line 110 requires two changes. We want to fix "bee" and "makng". The double "e" problem is solved with "delete" and the missing letter is handled with "insert".

```
EDIT 110
```

Press the space bar repeatedly until the "e" in "bee" appears.

....Delete

Next, simply press "D" or "d" to delete a single character. The editor replies by displaying

```
11Ø  PRINT "We will be\e\
```

With the cursor just sitting there. The letter "e" displayed between the backslashes has been deleted. Now we may continue on the same line by pressing the space bar four times until the "k" appears. At this point "Ii" will insert an "i" to correct the word "making". The first "I" is for insert and the second "i" gets inserted. Press RETURN and the edit is complete for line 110. Your screen should look like this:

```
11Ø  PRINT "We will be\e\ making a few changes."
```

Now we are ready to make three changes in line 120. We need to fix the triple "l", change "alone" to "along", and insert the word "exercises" at the end of the line.

```
EDIT 12Ø
```

We could pass over 16 characters until the cursor gets to the first "l". Or we could use a new command to search for it.

....Search

Simply key in "S1" to tell the editor to search for the desired character. The line up to that point is displayed, and the editor awaits our delete command. The errant "l" is backslashed out of existence with delete and we may proceed to the next edit. Now we may search for the "e" in alone. What we want now is to for it to become a "g". We could use delete followed by insert "g". Or we might prefer to change the next character to a "g" with the change command.

....Change

We may enter "Cg" to change the next character to a "g" without regard to what was there. Following this we simply press the space bar enough to get to the point where we want to insert " exercises". The resulting display should match the following:

```
12Ø  PRINT "If we fo\l\llow along with the exercises,"
```

As you become more experienced with editing you will see several ways to achieve the same thing. For example, to insert the word "exercises" above, we might have preferred to search for the comma and then insert the desired word. Clearly the search method is faster. As you grasp the options that the editor offers you will also learn to quickly judge whether there is a faster alternative. Just don't use up a lot of time trying to think of a clever technique to save a millisecond. Each person develops his or her own techniques.

We have two changes to make in line 130. We need to fix "becom" and "editting".

```
EDIT 13Ø
```

Let's search for the "m" and then press the space bar once. At this point we may insert an "e" with "Ie". What next? We could press RETURN to signify the end of an edit and then reedit the line. But that seems cumbersome. And it is. It will be better for us to exit the insert mode with the ESCape key and then search for the first "t" in "editting".

....ESCape

Once we are in EDIT there are two submodes. We may be commanding the editor to do something or we may be inserting text. We insert with the insert command. Once in there we may exit in one of two ways. We have been using the RETURN key to get out. But that takes us all the way out of the editor entirely. We may use ESCape to simply exit insert without leaving the editor. This allows us to make numerous changes on a program line in a single edit. In our example we press ESCape and then "St" to search for the double "t". Next delete with "D" and press RETURN.

```
13Ø  PRINT "then we will become edi\t\ting experts."
```

Now we really ought to insert the word "Mode" in line 100. Type EDIT 100 again. Press the space bar until the cursor has passed over "EDIT". Or use a combination of the search command and the space bar. It will be necessary to search for "T" several times, or search for "D" and then press the space bar to locate the cursor properly. With practice you will learn to size up the best way to get to the point of the line where you want to be. Next, type "I Mode" and RETURN. Insert lets us insert as many characters as we like. We are limited to 255 characters on the line, though. It'll be a little while before that is a problem for us. It is the RETURN key that signals the end of the insertion.

And finally we may examine our edited program with the LIST command in BASIC.

```
LIST
1ØØ  PRINT "This is an EDIT Mode example."
11Ø  PRINT "We will be making a few changes."
12Ø  PRINT "If we follow along with the exercises,"
13Ø  PRINT "then we will become editing experts."
```

We have seen how to do six different things in EDIT Mode:

1. Move the cursor (press the space bar)
2. Insert
3. Delete
4. Search
5. Change
6. Leave the editor (RETURN)

It is best to practice each of the features we have demonstrated here. Work on this until things become reflex actions. Don't stop now. The rest of this appendix is about to unveil more powerful features of EDIT Mode.

....Beneath the Surface

After you have mastered the features presented thus far you will want to learn more. Once EDIT Mode becomes second nature to you, you may concentrate on your programs rather than on wrestling with the process of keying in BASIC program statements and getting them right. We present here a description of the additional capabilities that go with each of the six basic EDIT Mode functions.

1. Move the cursor (space bar and Rubout or left arrow)

Press the space bar once and the cursor moves one character to the right. Press it twice to move two characters. We may move the cursor any number of spaces by first entering that number. Thus if we press 5 followed by the space bar the cursor will move 5 characters to the right.

We can move the cursor to the left with the Rubout key or left arrow. Rubout and left arrow are two different symbols that have the same effect in this environment. (Rubout is CTRL-@ on Apple II and II Plus.) Press it once to move 1 character. To move 10 characters enter 10 and press Rubout. Just like magic.

2. Insert (I, #I, and X)

We insert characters on a line by typing "I" followed by our desired insertion. We insert x characters with "xi" followed by x-characters. We leave insert with either RETURN or ESCape.

A special command has been included for the sole purpose of extending a line. The "X" subcommand moves the cursor to the end of a line and goes into the insert submode.

We delete characters in this submode with Rubout or left arrow. Left arrow causes the cursor to move to the left over one character each time it is pressed. Rubout displays an underline character (_). Either way, a character is removed for each key press.

3. Delete (D, #D, and H)

Each time we use the delete command a character is deleted. We may delete any number of characters by entering that number before issuing the delete command. All characters are displayed between a pair of backslashes. If the number we enter is greater than the number of characters on the rest of the line, delete only removes the rest of the line. It does not extend to the next line.

The H command deletes the rest of the line to the right of the cursor and goes into insert mode.

4. Search (S, #S, K, and #K)

We may search for a character with the S command. We may also search for the ith occurrence of a character with iS.

The K search is both powerful and dangerous. The K search deletes all characters passed over in the search. If the character you are searching for is not found then the rest of the line is deleted. Use this one carefully.

5. Change (C and #C)

Change the next character with the C command. To change i characters, enter iC followed by the i characters you wish to insert. For example, "3cNEW" will replace the next three characters with "NEW".

6. Leave the editor (RETURN, E, Q, L, and A)

There are a number of commands grouped here that have to do with managing the entire edit. RETURN at any time ends the editing session for this line. In this case the remainder of the line is displayed. To avoid the display use the "E" command. The "Q" command allows us to quit the editor without making any of the changes we have entered. Thus the line remains as it was before we began to tinker. We may display the line in its current form with the "L" command. The line will be displayed and the line number will appear with the cursor just as it does at the beginning of an edit. The "A" command simply allows us to start all over by throwing away any changes we have made.

....Miscellaneous Additional EDIT Features

When BASIC encounters a syntax error during program execution it automatically enters EDIT Mode on the errant line. Normally we want to fix the line and RUN the program again. If we don't want to edit at this time, then we may simply leave the editor with the "Q" command or press RETURN.

We may enter EDIT Mode at any time in BASIC by typing Control-A. Hold down the CTRL key and press the letter A. BASIC-80 will move the cursor to a new line, display an exclamation point (!), and wait for your EDIT Mode commands. We may use this to fix a line we are currently typing or to fix the last line typed. Thus, if we issue a command at the keyboard and wish to do it again, we may type CTRL-A and press RETURN. Try it. Or suppose we type a command line and make a typing error. We may use CTRL-A to correct the error and reissue the command in one operation. If the command last issued was LIST, CTRL-A will enter EDIT Mode beginning at the last program line listed but without the line number. Now you know. This CTRL-A thing also works while responding to an INPUT statement. This makes it easy to edit data entered during program execution. Remember, not all 80-column cards support CTRL-A as a way to enter EDIT Mode.

Suppose you only want to change the line number. You could type it again with the correct line number. Or you could fool EDIT Mode into doing most of the work for you. A simple two-step edit will do the job. Let's change line 200 to line 500. First EDIT 200 and press RETURN. Second, press CTRL-A and insert 500 at the beginning of the line. You may now list the program to verify that the program line appears at both line 200 and line 500. To eliminate line 200 simply type 200 followed by the RETURN key.

Finally, the editor always remembers the most recent line number as "." (dot). So "EDIT ." will enter the editor with the most recent line number. The space is required between the "T" and the ".".

The beginner should spend some time at a keyboard trying all of the EDIT Mode features. A little time spent learning how to use it will pay off in much time saved as you learn to write programs.

Appendix C

Using the Disk

With a disk we can easily save programs for future use.

We are going to learn about SAVE, RESET, FILES, LOAD, RUN, MERGE, KILL, and NAME. All of these directives allow us to designate which disk drive to use.

....Program Names (and File Names, Too!)

We are allowed up to eight characters in program names. As we will see shortly, BASIC will add the three characters "BAS". So, if we name a program "TESTING", BASIC will call it "TESTING.BAS". There is the name and the three-character extension. They are separated by a period (.).

....Disk Drives

Under CP/M we simply use a letter followed by a colon to refer to a disk drive. If we want the current one, then we may omit the drive designation. To specify program "EGGS.BAS" on drive B, we simply attach the drive designation to the program name:

```
"B:EGGS.BAS"
```

It is that simple.

....SAVE

Suppose we have just put the finishing touches on one of our eggs programs. We can save our program on disk with the following:

```
SAVE "EGGS"
```

It's a good idea to use all uppercase letters for file names. After a little disk activity BASIC will return to await our next command. The program will be saved under the name "EGGS.BAS". The extension ".BAS" is added by BASIC. If you want some other extension, just include it when you type the SAVE command.

There are three formats for program files stored on disk by BASIC. The EGGS program mentioned earlier would be saved in compressed binary format by our first command. This format is usable only by BASIC, but it saves disk space for large programs.

There are a number of program editors available. These editors give us the ability to make wholesale changes in our BASIC programs. For example, we might want to change all variables named NUMBER to NEW-NUMBER. This would be somewhat tedious with EDIT Mode in BASIC, but it would go fast and easy with a good program editor. ED.COM is a program editor that comes with CP/M. A few simple commands are used to edit text in the computer. ED does not depend on our writing a BASIC program. We could write a letter to a friend. Instructions for the use of this program are included with the CP/M documentation. The catch is that most program editors require that the file being edited be stored in ASCII format. That is, each character must be stored as a single character using a standard coding method. Using this format the word PRINT is stored as five characters, whereas PRINT is stored in the space of a single character using compressed binary format. It is a simple matter to save our program in ASCII format.

```
SAVE "EGGS",A
```

does the job.

....Protected Programs

Sometimes we have a program that we want to let other people use, but we don't want them to be able to read the BASIC code. We may protect our program with

```
SAVE "EGGS",P
```

We have to be careful with this one. Even we cannot LIST the program. It is necessary to save such a program in an unprotected format as well. If we try to LIST a program saved in protected format, we will be greeted with

```
Illegal function call
```

Most things we might try to do to change the program are greeted with the same message. Note that if the program name we use in a SAVE statement is already on the disk, BASIC simply replaces it with the current

program. You are warned to be careful not to wipe out another old program by selecting its name for a new one. We go merrily along writing and saving programs, and someday we get the message

```
Too many files
```

It means just that. The disk has room for just so many files, and eventually we reach that point. To see the directory of files on the disk simply use the FILES command (see below). We may use KILL (see below) to eliminate junk programs and use SAVE again.

If we find that we cannot part with any programs on this disk, we switch disks and try again. But now we will get the message

```
Disk Read Only
```

Aha! BASIC remembers where programs are on each disk. If we remove one and insert another, BASIC will remember about the wrong disk. That message is for our own protection. We use the RESET command to tell BASIC to remember this disk now. Now we can SAVE our precious program. That was the RESET command—not the RESET key. What else can happen? Well, if our programs become very long or we work with files that contain a lot of data, we might see the following message:

```
Disk full
```

That means just what it says. There is no more room no matter what. Again, determine what can be erased from the disk to make room.

....FILES

The FILES command causes the name of the files on the current disk to be displayed. We may use the wild-card features of the CP/M operating system to view only selected files. Thus,

```
FILES "*.BAS"
```

will display only those files with the ".BAS" extension. And

```
FILES "CA??????.*"
```

will display a directory of all files that begin with CA.

....LOAD

Any program SAVEd is easily brought back with LOAD.

```
LOAD "EGGS"
```

loads a copy of our program into memory from disk. To execute the program we just issue the RUN command. Alternatively we could RUN our program directly with

```
LOAD "EGGS",R
```

The "R" option causes the program to execute as soon as it is loaded. It is

important to know that the "R" option keeps all files "OPEN". This command can be included within another program. So we may move from program to program under program control. LOAD replaces any program already in memory.

....RUN

This statement may also be used to directly execute a program on disk.

```
RUN "EGGS"
```

will replace any program in memory and execute "EGGS.BAS" stored on disk. Note that the "R" option to keep files open (described under LOAD) may also be used with RUN "program".

....MERGE

As we develop more and more programs, we will discover that routines written for one program exactly fit for another. We can use MERGE to incorporate BASIC statements stored in a file on disk with BASIC statements stored in computer memory. We simply need to make sure that the line numbers do not conflict.

```
MERGE "EGGS1"
```

will blend the code from EGGS1 on disk with any program residing in memory at the time. If there is a line-number conflict, then the statements coming from disk prevail. The program coming from disk must have been SAVEd with the ,A option for MERGE.

....KILL

We must have the ability to erase old, unwanted files from disk. The KILL command does it:

```
KILL "EGGS.BAS"
```

The KILL statement requires the extension even though SAVE, LOAD, and RUN don't. Needless to say, the KILL statement should be used with great care. There is no easy way to "UNKILL" a file. BASIC will KILL "*.*", but you probably wish it wouldn't.

....NAME

We may change the name of a file on disk with the NAME statement.

```
NAME "EGGS.BAS" AS "HAMNEGGS.BAS"
```

changes the name of EGGS program to HAMNEGGS. Note that the file name extension is also required. While SAVE can wipe out an old program, NAME will report

```
File already exists
```

to save us a lot of trouble. If we really want to replace the old one, we use KILL and NAME again.

Appendix D

ASCII Character Chart

This ASCII chart has been simplified. The control characters (codes 0 to 31) and the ASCII codes 96 to 127 have real meaning and are used by many computers. The codes 128 to 255 are essentially a repeat of codes 0 to 127.

Value	Character	Value	Character	Value	Character
00	CTRL-@	16	CTRL-P	32	SPACE
01	CTRL-A	17	CTRL-Q	33	!
02	CTRL-B	18	CTRL-R	34	"
03	CTRL-C	19	CTSL-S	35	#
04	CTRL-D	20	CTRL-T	36	$
05	CTRL-E	21	CTRL-U	37	%
06	CTRL-F	22	CTRL-V	38	&
07	CTRL-G	23	CTRL-W	39	'
08	CTRL-H	24	CTRL-X	40	(
09	CTRL-I	25	CTRL-Y	41	)
10	CTRL-J	26	CTRL-Z	42	*
11	CTRL-K	27	ESC	43	+
12	CTRL-L	28	FS	44	,
13	CTRL-M	29	CTRL-SHFT-M	45	-
14	CTRL-N	30	CTRL-SHFT-N	46	.
15	CTRL-O	31	US	47	/

Value	Character	Value	Character	Value	Character
48	0	75	K	102	f
49	1	76	L	103	g
50	2	77	M	104	h
51	3	78	N	105	i
52	4	79	O	106	j
53	5	80	P	107	k
54	6	81	Q	108	l
55	7	82	R	109	m
56	8	83	S	110	n
57	9	84	T	111	o
58	:	85	U	112	p
59	;	86	V	113	q
60	<	87	W	114	r
61	=	88	X	115	s
62	>	89	Y	116	t
63	?	90	Z	117	u
64	@	91	[†	118	v
65	A	92	\†	119	w
66	B	93	]†	120	x
67	C	94	^	121	y
68	D	95	_	122	z
69	E	96	`	123	{
70	F	97	a	124	|
71	G	98	b	125	}
72	H	99	c	126	
73	I	100	d	127	DEL
74	J	101	e		

† These characters are not labeled on the Apple II or the Apple II Plus keyboard. ASCII 91 ([) is generated by pressing CTRL-K, 92 (\) comes from CTRL-B, and we get 93 (]) from SHIFT-M. They are readily available on the Apple IIe keyboard and on most external terminals.

CTRL-@ is the Rubout character. CTRL-A summons up EDIT Mode. CTRL-B is used for the backslash (\). CTRL-C brings program execution to a halt. CTRL-G produces a bell-like sound. CTRL-H is the backspace character (left arrow). CTRL-I tabs to the next eight-character column. CTRL-J causes a line feed (lf). CTRL-K becomes a right square bracket (]). CTRL-M generates the RETURN character (cr). CTRL-O toggles program display while execution proceeds. CTRL-S suspends program execution. Any key resumes. CTRL-X cancels the current typed line.

The program CONFIGIO.BAS, supplied with the SoftCard disk, may be used to reassign keys for special purposes.

Appendix E

Index of Programs

Appendix F

Solution Programs for Even-Numbered Problems

Each two-page spread should be read from top to bottom as one individual page.

Chapter 1

Section 1-1

Problem No. 2

```
100  PRINT "Programming is fun.  ";
110  PRINT "The computer will solve many problems for us."

run
Programming is fun.  The computer will solve many problems for us.
Ok

run 110
The computer will solve many problems for us.
Ok
```

Section 1-2

Problem No. 2

```
100  PRINT (78 + 89 + 82) / 3
```

```
120  PRINT "Interest rate in percent";
130  INPUT RATE
140  PRINT "   Dollar amount of loan";
150  INPUT AMOUNT
160  PRINT
170  PRINT "        Interest is $"; RATE * AMOUNT / 100
180  PRINT "         Amount is $";
190  PRINT RATE * AMOUNT / 100 + AMOUNT
run
I will calculate simple interest.

Interest rate in percent? 11.98
   Dollar amount of loan? 1000

        Interest is $ 119.8
         Amount is $ 1119.8
```

Section 2-2

Problem No. 2

```
100  READ A, B, C, D, E
```

```
run
 83
Ok
```

Problem No. 4

```
100  PRINT "I am 15 years, 3 months, and 2 days old."
110  PRINT "That makes approximately:"; 5576 * 24; "Hours"
120  PRINT "We got 5576 days from the answer"
122  PRINT "to Problem 3."

run
I am 15 years, 3 months, and 2 days old.
That makes approximately: 133824 Hours
We got 5576 days from the answer
to Problem 3.
Ok
```

Problem No. 6

```
100  PRINT 283.4 + 658 + 385.8 + 17

run
 1344.2
```


Chapter 2

Section 2-1

Problem No. 2

```
90   PRINT "I will calculate the average of three numbers"
95   PRINT
100  PRINT "Enter your three numbers:";
110  INPUT A, B, C
120  PRINT "The average is:"; (A + B + C) / 3

run
I will calculate the average of three numbers

Enter your three numbers:? 43,56,12
The average is: 37
```

Problem No. 4

```
100  PRINT "I will calculate simple interest."
110  PRINT
110  LET N = A/B + B/C
120  LET D = D/E + A/B
130  PRINT N / D
900  DATA 2, 3, 4, 5, 6

run
 .944445
```

Problem No. 4

```
60   PRINT "Demonstrate the MOD operator"
70   PRINT
80   PRINT "Enter two numbers (A,B)";
100  INPUT A, B
120  PRINT "A MOD B ="; A MOD B

run
Demonstrate the MOD operator

Enter two numbers (A,B)? 5,7
A MOD B = 5

run
Demonstrate the MOD operator

Enter two numbers (A,B)? 7,5
A MOD B = 2
```

Section 2-3

Problem No. 2

```
100  READ A0$, A1$, A2$, A3$, A4$, A5$, A6$
120  PRINT A0$
130  PRINT A1$
140  PRINT A2$
150  PRINT A3$
160  PRINT A4$
170  PRINT A5$
180  PRINT A6$
900  DATA Sunday, Monday, Tuesday, Wednesday
910  DATA Thursday, Friday, Saturday

run
Sunday
Monday
```

Section 2-3 *Problem No. 2 (continued)*

```
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Problem No. 4

```
100  PRINT "Enter anything -";
120  INPUT A$
130  PRINT
140  PRINT "You entered <"; A$; ">"

run
Enter anything -? Green

You entered <Green>
```

Chapter 3

Section 3-1

Problem No. 2

```
100  COUNT = 1
120  IF COUNT > 19 THEN 190
130   PRINT COUNT
140   COUNT = COUNT + 1
160   GOTO 120
190  PRINT "Done"

run
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 11
 12
-4
-5
-6
-7
-8
-9
-10
Done
```

Section 3-2

Problem No. 2

```
80   PRINT "Bounce", "Height"
90   LET HEIGHT = 10
95   LET OLD.HEIGHT = HEIGHT
100  LET COUNT  = 1
105  LET DISTANCE = 0
115  LET DISTANCE = DISTANCE + HEIGHT
120  LET HEIGHT = HEIGHT * .9
125  LET DISTANCE = DISTANCE + HEIGHT
130  PRINT COUNT, HEIGHT
135  IF HEIGHT * .9 < OLD.HEIGHT / 2 THEN 190
140   LET COUNT = COUNT + 1
160   GOTO 115
190  PRINT COUNT; "Bounces"
200  PRINT DISTANCE; "Meters - total distance"

run
Bounce        Height
 1             9
 2             8.1
 3             7.29
 4             6.561
 5             5.9049
 6             5.31441
 6 Bounces
 89.0262 Meters - total distance
```

Problem No. 4

```
100  PRINT "Enter cents:";
105  INPUT CENTS
110  IF CENTS = 0 THEN 200
120  READ COIN, COIN$
150   NUMBER = CENTS \ COIN 'Note integer division
```

```
 13
 14
 15
 16
 17
 18
 19
Done
```

Problem No. 4

```
100  COUNT = 1
110  TOTAL = 0
120  IF COUNT > 100 THEN 190
135   TOTAL = TOTAL + COUNT
140   COUNT = COUNT + 1
160   GOTO 120
190  PRINT TOTAL

run
 5050
```

Problem No. 6

```
100  COUNT = 10
120  IF COUNT < -10 THEN 190
130   PRINT COUNT
140   COUNT = COUNT - 1
160   GOTO 120
190  PRINT "Done"

run
 10
 9
 8
 7
 6
 5
 4
 3
 2
 1
 0
-1
-2
-3
```

```
160   IF NUMBER = 0 THEN 120
170   PRINT NUMBER; COIN$
175   CENTS = CENTS - NUMBER * COIN
180   GOTO 110
200  PRINT "Done"
900  DATA 50, Half dollars
902  DATA 25, Quarters
904  DATA 10, Dimes
906  DATA  5, Nickels
908  DATA  1, Pennies

run
Enter cents:? 91
 1 Half dollars
 1 Quarters
 1 Dimes
 1 Nickels
 1 Pennies
Done
```

Chapter 4

Section 4-1

Problem No. 2

```
100  FOR COUNT = 93 TO 80 STEP -2
110   PRINT COUNT;
120  NEXT COUNT

run
 93  91  89  87  85  83  81
```

Problem No. 4

```
100  FOR COUNT = 1 TO 15
110   PRINT COUNT, 1 / COUNT
120  NEXT COUNT

run
 1            1
 2            .5
 3            .333333
 4            .25
 5            .2
 6            .166667
```

Section 4-1 *Problem No. 4 (continued)*

```
 7              .142857
 8              .125
 9              .111111
 10             .1
 11             .0909091
 12             .0833333
 13             .0769231
 14             .0714286
 15             .0666667
```

Problem No. 6

```
100  FOR COUNT = 1 TO 11
110   PRINT COUNT, COUNT / 11
120  NEXT COUNT

run
 1              .0909091
 2              .181818
 3              .272727
 4              .363636
 5              .454545
 6              .545455
 7              .636364
 8              .727273
 9              .818182
 10             .909091
 11             1
```

Problem No. 8

```
100  FOR COUNT = 1 TO 1.2 STEP .1
110   PRINT COUNT
120  NEXT COUNT

run
 1
 1.1
 1.2
```

Section 4-2

Problem No. 2

```
80   A = 0 : B = 0 : FIB = 1
```

```
The last gift would go back one day
short of a year later.
```

Section 4-3

Problem No. 2

```
90   PRINT "Fibonacci numbers:"
100  B = 0 : FIB = 1
200  FOR J = 1 TO 10
210   PRINT FIB,
215   X = FIB * FIB
220   A = B : B = FIB : FIB = A + B
230   PRINT X, A * FIB, X - A * FIB
290  NEXT J

run
Fibonacci numbers:
 1             1             0             1
 1             1             2            -1
 2             4             3             1
 3             9            10            -1
 5            25            24             1
 8            64            65            -1
 13          169           168             1
 21          441           442            -1
 34         1156          1155             1
 55         3025          3026            -1
```

Section 4-4

Problem No. 2

```
100  PRINT "Pythagorean triples"
110  FOR LEG1 = 1 TO 25
120   FOR LEG2 = LEG1 + 1 TO 50
140    FOR HYPOT = LEG2 TO 75
145     IF LEG1*LEG1 + LEG2*LEG2 < HYPOT*HYPOT THEN 200
150     IF LEG1*LEG1 + LEG2*LEG2 > HYPOT*HYPOT THEN 190
180     PRINT LEG1; TAB(5); LEG2; TAB(10); HYPOT
182     GOTO 200
190    NEXT HYPOT
200   NEXT LEG2
210  NEXT LEG1

Pythagorean triples
 3    4    5
```

```
100  FOR COUNT = 1 TO 20
110   PRINT USING "### -> #####"; COUNT, FIB
120   A = B : B = FIB : FIB = A + B
130  NEXT COUNT

run
  1 ->       1
  2 ->       1
  3 ->       2
  4 ->       3
  5 ->       5
  6 ->       8
  7 ->      13
  8 ->      21
  9 ->      34
 10 ->      55
 11 ->      89
 12 ->     144
 13 ->     233
 14 ->     377
 15 ->     610
 16 ->     987
 17 ->    1597
 18 ->    2584
 19 ->    4181
 20 ->    6765
```

Problem No. 4

```
100  PRINT "The twelve days of Christmas"
110  PRINT
120  GIFTS = 0
130  TODAY = 0
150  FOR DAY = 1 TO 12
160   TODAY = TODAY + DAY
170   GIFTS = GIFTS + TODAY
180  NEXT DAY
190  PRINT GIFTS; "Gifts"
500  PRINT
502  PRINT "The last gift would go back one day"
504  PRINT "short of a year later."

run
The twelve days of Christmas

 364 Gifts
```



```
 5    12   13
 6    8    10
 7    24   25
 8    15   17
 9    12   15
 9    40   41
 10   24   26
 12   16   20
 12   35   37
 14   48   50
 15   20   25
 15   36   39
 16   30   34
 18   24   30
 20   21   29
 20   48   52
 21   28   35
 24   32   40
 24   45   51
```

Chapter 5

Section 5-1

Problem No. 2

```
100  FOR N = 1 TO 20
110   ROOT = SQR(N)
120   PRINT USING "### ##.#"; N, ROOT
190  NEXT N

run
  1  1.0
  2  1.4
  3  1.7
  4  2.0
  5  2.2
  6  2.4
  7  2.6
  8  2.8
  9  3.0
 10  3.2
 11  3.3
 12  3.5
 13  3.6
 14  3.7
 15  3.9
```

Section 5-1 *Problem No. 2 (continued)*

```
 16  4.0
 17  4.1
 18  4.2
 19  4.4
 20  4.5
```

Problem No. 4

```
100  INPUT "Enter a date in the form YYMMDD"; DATE
110  IF DATE = 0 THEN END
120  YEAR  = INT(DATE / 10000)
140  MONTH = INT( (DATE - YEAR*10000) / 100 )
160  DAY   = DATE - YEAR*10000 - MONTH*100
196 :
198  REM ** Let's validate the entered value
200  IF YEAR  < 0  THEN 300
210  IF YEAR  > 99 THEN 300
220  IF MONTH < 1  THEN 310
230  IF MONTH > 12 THEN 310
240  IF DAY   < 1  THEN 320
250  IF DAY   > 31 THEN 320
260  PRINT YEAR; MONTH; DAY
290  END
296 :
298  REM ** Error messages
300  PRINT "Bad year"  : PRINT : GOTO 100
310  PRINT "Bad month" : PRINT : GOTO 100
320  PRINT "Bad day"   : PRINT : GOTO 100

run
Enter a date in the form YYMMDD? 490212
 49 2 12

run
Enter a date in the form YYMMDD? 322104
Bad month
Enter a date in the form YYMMDD? 321204
 32 12 4
```

Section 5-2

Problem No. 2

```
10   HOME
200  IF YEAR  < 0  THEN 300
210  IF YEAR  > 99 THEN 300
220  IF MONTH < 1  THEN 310
230  IF MONTH > 12 THEN 310
240  IF DAY   < 1  THEN 320
250  IF DAY   > 31 THEN 320
260  Y$ = MID$(STR$(YEAR),2) : IF YEAR < 10 THEN Y$ = "0" + Y$
265  D$ = MID$(STR$(DAY ),2) : IF DAY  < 10 THEN D$ = "0" + D$
270  M$ = MID$(MONTH$,MONTH*3 - 2,3)
280  PRINT Y$; "-"; M$; "-"; D$
290  END
296 :
298  REM ** Error messages
300  PRINT "Bad year"  : PRINT : GOTO 100
310  PRINT "Bad month" : PRINT : GOTO 100
320  PRINT "Bad day"   : PRINT : GOTO 100

run
Enter a date in the form YY/MM/DD? 76/70/04
Bad month
Enter a date in the form YY/MM/DD? 76/07/04
76-Jul-04
```

Section 5-3

Problem No. 2

```
100  FOR I = 1 TO 200
120   COIN = INT( RND*2 )
130   IF COIN = 0 THEN HEADS = HEADS + 1
140   IF COIN = 1 THEN TAILS = TAILS + 1
150  NEXT I
180  PRINT HEADS; "Heads"
190  PRINT TAILS; "Tails

run
 106 Heads
 94 Tails
```

Section 5-4

Problem No. 2

```
10   REM ** Convert from Fanrenheit to centigrade
50   DEF FNC(X) = ( X-32 )*5 / 9
```

```
200  B$ = SPACE$(39)
210  READ A$ : IF A$ = "Done" THEN END
220  FOR I9 = 1 TO LEN(A$)
230   FOR X = 1 TO 40 : NEXT X 'For timing
240   B$ = RIGHT$(B$,38) + MID$(A$,I9,1)
250   HTAB 1 : PRINT B$;
260  NEXT I9
290  GOTO 210
898 :
900  DATA "Here we go, across the screen"
910  DATA " just like downtown.  Today"
920  DATA " there was big news in the"
930  DATA " computer world.  A creature"
940  DATA " from outer space took out a patent"
950  DATA " on a revolutionary device that"
960  DATA " will ...  "
998  DATA "                                        "
999  DATA "Done"
```

Problem No. 4

```
698  REM ** What day is this?
700  WEEK$ = "SUNMONTUEWEDTHUFRISAT"
710  INPUT "Weekday"; DAY$
715  IF LEN(DAY$) = 0 THEN END
720  DAY$ = LEFT$(DAY$,3) '3 characters to match
730  P    = INSTR(WEEK$,DAY$)
740  IF P = 0 THEN PRINT "Not found" : GOTO 710
780  PRINT "Day number"; P \ 3 + 1

run
Weekday? January
Not found
Weekday? Tuesday
Day number 3
```

Problem No. 6

```
90   MONTH$ = "JanFebMarAprMayJunJulAugSepOctNovDec"
100  INPUT "Enter a date in the form YY/MM/DD"; DATE$
105  IF LEN(DATE$) = 0 THEN END
120  YEAR  = VAL(LEFT$ (DATE$,2  ))
140  MONTH = VAL(MID$  (DATE$,4,2))
160  DAY   = VAL(RIGHT$(DATE$,2  ))
196 :
198  REM ** Let's validate the entered value
```

Problem No. 4

```
10   REM ** Convert from upper to lower case
50   DEF FNUL$(X$) = CHR$( ASC(X$) - 32*( ASC(X$)>64 AND ASC(X$)<91) )
```

Section 5-6

Problem No. 2

```
90   MONTH$ = "JanFebMarAprMayJunJulAugSepOctNovDec"
100  INPUT "Enter a date in the form YY/MM/DD"; DATE$
105  IF LEN(DATE$) = 0 THEN END
110  GOSUB 500 'Verify date INPUT
120  IF ERROR.MESSAGE$ <> "Ok" THEN PRINT ERROR.MESSAGE$ : GOTO 100
130  GOSUB 600 'Form the date string
140  PRINT NEW.DATE$
290  END
496 :
498  REM ** Verify the date
500  ERROR.MESSAGE$ = "Ok"
510  DAY   = VAL(RIGHT$(DATE$,2  ))
520  IF DAY   < 1 OR DAY   > 31 THEN ERROR.MESSAGE$ = "Bad day"
530  MONTH = VAL(MID$  (DATE$,4,2))
540  IF MONTH < 1 OR MONTH > 12 THEN ERROR.MESSAGE$ = "Bad month"
550  YEAR  = VAL(LEFT$ (DATE$,2  ))
560  IF YEAR  < 0 OR YEAR  > 99 THEN ERROR.MESSAGE$ = "Bad year"
570  IF LEN(DATE$) <> 8         THEN ERROR.MESSAGE$ = "Bad date"
590  RETURN
596 :
598  REM ** Form the date string
600  Y$ = MID$(STR$(YEAR),2) : IF YEAR < 10 THEN Y$ = "0" + Y$
610  M$ = MID$(MONTH$,MONTH*3 - 2,3)
620  D$ = MID$(STR$(DAY ),2) : IF DAY  < 10 THEN D$ = "0" + D$
630  NEW.DATE$ = Y$ + "-" + M$ + "-" + D$
690  RETURN

run
Enter a date in the form YY/MM/DD? 77/02/14
77-Feb-14
```

Chapter 6

Section 6-1

Problem No. 2

```
50   WEEK$ = "SunMonTueWedThuFriSat"
90   REM ** Enter the temperatures in array WEEK
```

Section 6-1 *Problem No. 2 (continued)*

```
100  FOR J = 1 TO 7
110   READ WEEK(J)
120  NEXT J
146 :
148  REM ** Set up initial conditions
150  SUM  = WEEK(1)
160  HIGH = WEEK(1) : LOW = WEEK(1)
170  LOW.DAY = 1 : HIGH.DAY = 1
196 :
198  REM ** Scan the week's temperatures
200  FOR J = 2 TO 7
210   SUM = SUM + WEEK(J)
220   IF WEEK(J) < LOW  THEN  LOW  = WEEK(J) : LOW.DAY  = J
230   IF WEEK(J) > HIGH THEN  HIGH = WEEK(J) : HIGH.DAY = J
290  NEXT J
300  PRINT "Average temp:"; SUM / 7
310  PRINT "Highest temp:"; HIGH
320  PRINT " Lowest temp:"; LOW
330  PRINT " Highest day: "; MID$(WEEK$,HIGH.DAY*3 - 2, 3)
340  PRINT "  Lowest day: "; MID$(WEEK$,LOW.DAY *3 - 2, 3)
896 :
900  DATA 71, 77, 82, 76, 79, 72, 74
990  END

run
Average temp: 75.8571
Highest temp: 82
 Lowest temp: 71
 Highest day: Tue
  Lowest day: Sun
```

Problem No. 4

```
50  RANDOMIZE
90   REM ** Drawing five random numbers from among ten
96  :
98   REM ** Make all values available
100  FOR J = 1 TO 10
110   A(J) = 1 'Value available
120  NEXT J
130  DUPLICATE = 0
196 :
198  REM ** Select five random values
```

Section 6-2

Problem No. 2

```
50   REM **    A simple sort
98   REM ** Load numbers to be sorted in A array
100  N = 0
110  READ X : IF X = -999999! THEN 200
120   N = N + 1
130   A(N) = X
140   GOTO 110
196 :
198  REM ** Here is the sort
200  FOR LAST = N - 1 TO 2 STEP -1
230   FOR J = 1 TO LAST
240    IF A(J) < A(J+1) THEN SWAP A(J), A(J+1)
250   NEXT J
280  NEXT LAST
296 :
298  REM ** Sort complete - display
300  FOR J = 1 TO N
310   PRINT A(J);
320  NEXT J
890  END
896 :
898  REM ** Test data
900  DATA 102, 32, -91, 982, 87
902  DATA 73, 23, -981, 234, 21
990  DATA -999999

run
 982  234  102  87  73  32  23  21  -91  -981
```

Section 6-3

Problem No. 2

```
90   REM ** Experimenting with a 5 by 7 array
100  FOR ROW = 1 TO 5
110   FOR COLUMN = 1 TO 7
120    A(ROW, COLUMN) = INT( RND*150 )
130   NEXT COLUMN
140  NEXT ROW
150  GOSUB 300 'Largest value in rows
155  PRINT : PRINT
160  GOSUB 200 'Largest value in columns
190  END
```

```
200  FOR J = 1 TO 10
210   RANDOM = INT( RND * 10 + 1 )
250   IF A(RANDOM) = 0 THEN DUPLICATE = DUPLICATE + 1 : GOTO 210
260   PRINT RANDOM;
270   A(RANDOM) = 0 'Value unavailable
280  NEXT J
300  PRINT
310  PRINT DUPLICATE; "Duplicates"

run
Random number seed (-32768 to 32767)? 3
 9  5  6  2  8  10  4  7  1  3
 61 Duplicates

run
Random number seed (-32768 to 32767)? 2
 1  9  7  2  3  6  5  4  10  8
 9 Duplicates
```

Problem No. 6

```
198  REM ** Read the arrays
200  READ N1
210  FOR I = 1 TO N1 : READ A(I) : NEXT I
246 :
250  READ N2
260  FOR I = 1 TO N2 : READ B(I) : NEXT I
296 :
298  REM ** Load the third array
300  FOR J = 1 TO N1 : C(J) = A(J) : NEXT J : N3 = N1
310  FOR K = 1 TO N2
320   FOR J = 1 TO N3
330    IF C(J) = B(K) THEN 360
340   NEXT J
350   N3 = N3 + 1 : C(N3) = B(K)
360  NEXT K
396 :
400  FOR J = 1 TO N3 : PRINT C(J); : NEXT J
490  END
896 :
900  DATA 4, 3, 5, 6, 17
910  DATA 5, 6, -9, 11, -13, 3

run
 3  5  6  17  -9  11  -13
```

```
196 :
198  REM ** Display column totals
200  FOR COLUMN = 1 TO 7
210  LARGEST = A(1, COLUMN)
220   FOR ROW = 2 TO 5
230    IF A(ROW,COLUMN) > LARGEST THEN LARGEST = A(ROW,COLUMN)
240   NEXT ROW
250   PRINT LARGEST; "Largest for column"; COLUMN
260  NEXT COLUMN
290  RETURN
296 :
298  REM ** Display row columns
300  FOR ROW = 1 TO 5
310  LARGEST = A(ROW, 1)
320   FOR COLUMN = 1 TO 7
330    IF A(ROW, COLUMN) > LARGEST THEN LARGEST = A(ROW,COLUMN)
340   NEXT COLUMN
350   PRINT LARGEST; "Largest for row"; ROW
360  NEXT ROW
390  RETURN

run
 118 Largest for row 1
 147 Largest for row 2
 143 Largest for row 3
 130 Largest for row 4
 117 Largest for row 5


 143 Largest for column 1
 147 Largest for column 2
 135 Largest for column 3
 117 Largest for column 4
 83 Largest for column 5
 145 Largest for column 6
 136 Largest for column 7
```

Problem No. 4

```
50   RANDOMIZE
60   DIM A(100)
90   REM ** Drawing five random numbers from among ten
91   '      With trial-and-error
96  :
98   REM ** Make all values available
```

Section 6-3 *Problem No. 4 (continued)*

```
100  FOR J = 1 TO 100
110   A(J) = 1 'Value available
120  NEXT J
130  PRINT "With trial and error"
156 :
160  INPUT "Start timing and press 'RETURN'"; A$
196 :
198  REM ** Select one hundred random values
200  FOR J = 1 TO 100
210   RANDOM = INT( RND * 100 + 1 )
250   IF A(RANDOM) = 0 THEN 210
260   PRINT RANDOM;
270   A(RANDOM) = 0 'Value unavailable
280  NEXT J
285  BEEP 10,10
298  PRINT "Stop timing"
290  INPUT "Press 'RETURN'"; A$
380  PRINT
390  PRINT "Without trial and error"
396 :
398  REM ** And now the fast way
489 :
490  REM ** Random values without replacement
492  '      and without trial-and-error.
500  FOR J = 1 TO 100
510   A(J) = J
520  NEXT J
556 :
560  INPUT "Start timing and press 'RETURN'"; A$
596 :
600  FOR J = 1 TO 100
610   LAST = 100 - J + 1
630   S = INT( RND * LAST + 1 )
640   PRINT A(S);
650   A(S) = A(LAST) 'Move last value
670  NEXT J
685  BEEP 10,10
690  PRINT "Stop timing"
900  END

run
Random number seed (-32768 to 32767)? 9
With trial and error
```


Section 6-5

Problem No. 2

```
100  GOSUB 900
120  FOR I = 1 TO 3
130   PRINT RND;
140  NEXT I
190  END
896 :
898  REM ** Request a name for seeding RND
900  INPUT "What's your name"; NAME.$
910  SEED = 0
920  FOR J = 1 TO LEN(NAME.$)
930   SEED = SEED + ASC(MID$(NAME.$,J,1))
940  NEXT J
950  RANDOMIZE SEED
990  RETURN

run
What's your name? John Adams
 .598175   .0355749   .30511

run
What's your name? JOHN ADAMS
 .0815393   .888169    .986779
```

Problem No. 4

```
10   REM ** Replace line 3030 in Program 6-8.
11   'Adjust the number multiplied by the
12   'sign length to suit.
3030  FOR J = 1 TO 50 * LEN(SIGNS$(R)) : NEXT J
```

Problem No. 6

```
98   REM ** Tabulate frequency of letters on signs
100  DIM ALPHA(26)
110  GOSUB 400 'Do the tabulation
120  GOSUB 600 'display the results
190  END
396 :
398  REM * Tabulate frequency of letters of the alphabet
400  READ A$ : IF A$ = "Done" THEN 490
410  FOR I = 1 TO LEN(A$)
420  B$ = MID$(A$,I,1) : X = ASC( B$)
440   IF X < 65 THEN 480
```

```
Start timing and press 'RETURN'?
 100  45  58  95  55  83  29  48  43
 99  50  46  59  6  2  15  20  5  68
 54  31  90  9  61  60  10  66  13  41
 17  44  94  24  85  62  89  80  93  34
 49  91  69  75  33  39  11  77  97  1
 72  37  28  64  7  52  88  76  53  14
 84  25  63  92  27  18  47  86  26  87
 74  22  56  21  8  96  79  12  19  78
 82  35  38  23  16  3  81  67  73  42
 40  32  71  4  98  51  30  70  36  57
 65
Stop timing
Press 'RETURN'?

Without trial and error
Start timing and press 'RETURN'?
 61  18  95  80  46  2  29  64  99  92
 40  77  88  22  26  66  5  67  39  75
 44  32  86  70  3  33  47  84  73  4
 15  52  93  100  76  97  87  35  16
 10  43  13  51  41  53  69  89  94  63
 31  90  23  36  83  81  14  25  78  30
 20  62  54  19  58  9  7  96  37  21
 68  8  72  6  11  79  59  55  34  53
 42  60  49  17  91  50  98  74  27  71
 1  82  57  12  28  38  85  48  45  24
 65
Stop timing
```


Section 6-4

Problem No. 2

```
90   MONTH$ = "JanFebMarAprMayJunJulAugSepOctNovDec"
100  FOR R = 1 TO 3
110   FOR M = 1 TO 12
120    PRINT " "; MID$(MONTH$,3*M-3+R,1); " ";
130   NEXT M
140   PRINT
150  NEXT R

run
 J  F  M  A  M  J  J  A  S  O  N  D
 a  e  a  p  a  u  u  u  e  c  o  e
 n  b  r  r  y  n  l  g  p  t  v  c
```

```
450   IF X > 96 AND X < 123 THEN X = X - 32
460   IF X > 90 THEN 480
470   ALPHA(X-64) = ALPHA(X-64) + 1
480  NEXT I
485  GOTO 400
490  RETURN
596 :
598  REM ** Display letter frequency
600  FOR I = 1 TO 26
610   PRINT CHR$(I+64); "  "; ALPHA(I)
620  NEXT I
690  RETURN
1496 :
1498  REM ** The signs
1500  DATA Stop, Al's Pizza, Dairy Queen, Burger King
1502  DATA Yield, One Way, This Way Out, Detour
1504  DATA One Show Only Tonight, Exit Only, Entrance Only Please
1506  DATA Florida 2138 mi., Fly United, Jet Set Diner
1508  DATA Give Her a Valentine, Give Him a Valentine
1510  DATA First Avenue, North Side
1598  DATA Done

run
A   13
B   1
C   1
D   7
E   26
F   3
G   5
H   6
I   18
J   1
K   1
L   10
M   2
N   18
O   12
P   3
Q   1
R   10
S   8
T   15
U   6
V   5
```

Section 6-5 *Problem No. 6 (continued)*

```
W   3
X   1
Y   8
Z   2
```

Problem No. 8

```
5   REM ** Play a geography game
10  DEF FNLU$(X$) = CHR$(ASC(X$) + 32 * (ASC(X$)>96 AND ASC(X$)<123)
20  DIM NA$(300),AV(300)
30  GOSUB 8000 'Read names array
32  INPUT "Do you wish to see the instructions"; A$
34  IF FNLU$(LEFT$(A$,1)) = "N" THEN 37
35  GOSUB 9000 'Instructions
37  GOSUB 4000 'Initialize available names array
40  GOSUB 7000 'Computer starts
50  GOSUB 6000 'Person response
58  IF PE$ = "QUIT" THEN 75
60   GOSUB 5000 'Computer response
65   IF CP$ <> "QUIT" THEN 50
75  INPUT "Do you want another game"; A$
85  HOME
90  IF FNLU$(LEFT$(A$,1)) = "N" THEN END
100  FOR I9 = 1 TO 1000 : NEXT I9
120  GOTO 32
3996 :
3998  REM ** Initialize available names array
4000  FOR J9 = 1 TO N0
4010   AV(J9) = 1
4020  NEXT J9
4090  RETURN
4996 :
4998  REM ** Computer response
5000  FOR I9 = 1 TO N0
5010   IF FNLU$(LEFT$(NA$(I9),1)) = FNLU$(RIGHT$(PE$,1))
                    AND AV(I9) = 1 THEN 5050
5015  NEXT I9
5020  PRINT : PRINT " I have run out of names"
5025  CP$ = "QUIT"
5030  GOTO 5090
5050  CP$ = NA$(I9) : AV(I9) = 0
5060  PRINT "     I choose: "; CP$
5090  RETURN
5996 :
9010  PRINT "with you.  You will take turns with the" : PRINT
9015  PRINT "computer.  Each of you will be trying to"; : PRINT
9020  PRINT "think of names of places such that the" : PRINT
9025  PRINT "first letter of your name is the same as"; : PRINT
9030  PRINT "the last letter of the previously used" : PRINT
9035  PRINT "place name." : PRINT
9045  INPUT "Are you ready? "; A$
9065  IF FNLU$(LEFT$(A$,1)) <> "Y" THEN 9045
9070  FOR I9 = 1 TO 1000 : NEXT I9
9090  RETURN
```

Chapter 7

Section 7-1

Problem No. 2

```
10   REM ** Make the following changes ---
20   GOSUB 800 'Get the month and year range
25   FOR YEAR = Y1 TO Y2
35   NEXT YEAR
796 :
798  REM ** Request month and range of years
800  INPUT "                What month"; MONTH
810  IF MONTH < 1 OR MONTH > 12 THEN 800
820  INPUT "Range of years from, to"; Y1, Y2
830  IF Y1 < 0 OR Y1 > 99 THEN 820
840  IF Y2 < 0 OR Y2 > 99 THEN 820
850  IF Y1 > Y2 THEN 820
890  RETURN

run
            What month? 5
Range of years from, to? 36,37

      May     1936

Sun Mon Tue Wed Thu Fri Sat

                        1   2
 3   4   5   6   7   8   9
10  11  12  13  14  15  16
17  18  19  20  21  22  23
24  25  26  27  28  29  30
31
```

```
5998  REM ** Person go
6000  PRINT
6005  INPUT "   Your turn"; PE$
6010  IF LEN(PE$) < 2 THEN PRINT "Name too short" : GOTO 6005
6020  IF LEN(PE$) <> 4 THEN 6040
6022   XPE$ = PE$
6025   FOR I9 = 1 TO 4 'Convert to uppercase
6030    MID$(XPE$,I9,1) = FNLU$(MID$(PE$,I9,1))
6035   NEXT I9
6037   IF XPE$ = "QUIT" THEN PE$ = XPE$ : GOTO 6190
6040    IF FNLU$(LEFT$(PE$,1)) = FNLU$(RIGHT$(CP$,1)) THEN 6050
6045    PRINT "No match" : GOTO 6005
6050    FOR I9 = 1 TO N0
6055     IF PE$ = NA$(I9) THEN 6100
6060    NEXT I9
6065    IF N0 = 300 THEN PRINT "No room" : GOTO 6005
6080    N0 = N0 + 1 : NA$(N0) = PE$ : AV(N0) = 0
6085    GOTO 6190
6096 :
6098    REM ** Place name is on the list
6100    IF AV(I9) = 0 THEN PRINT "Already used" : GOTO 6005
6150    AV(I9) = 0
6190  RETURN
6996 :
6998  REM ** Computer begin the game
7000  X9 = INT(RND*N0 + 1)
7020  CP$ = NA$(X9) : AV(X9) = 0
7030  HOME
7040  PRINT "First place : "; CP$
7090  RETURN
7996 :
7998  REM ** Read names
8000  I9 = 1
8010  READ NA$(I9)
8020  IF NA$(I9) = "Done" THEN 8080
8030   I9 = I9 + 1 : GOTO 8010
8080  N0 = I9 - 1
8090  RETURN
8096 :
8100  DATA  New York, Chicago, Philadelphia, Boston
8590  DATA  "Done"
8996 :
8998  REM ** Instructions
9000  HOME
9005  PRINT "This program will play a geography game" : PRINT
```



```
      May     1937

Sun Mon Tue Wed Thu Fri Sat

                            1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26  27  28  29
 30  31
```

Problem No. 4

```
20   GOSUB 600 'Get a date
25   IF DATE = 0 THEN END
30   GOSUB 700 'Display day of the week
90   END
292 :
294  REM ** Calculate
296  '      WEEKDAY - 1st of Month
298  '      N - No. of days in Month
300  LEAP = 0 : IF YEAR MOD 4 = 0 THEN LEAP = 1
310  WEEKDAY = (YEAR + INT((YEAR + 3) / 4)) MOD 7
320  N = 0
330  FOR M = 1 TO MONTH
340   WEEKDAY = (WEEKDAY + N) MOD 7
350   N = 30 + ((M + (M > 7)) MOD 2)
360   IF M = 2 THEN N = 28 + LEAP
380  NEXT M
390  RETURN
496 :
498  REM ** Verify the date
500  ERROR.MESSAGE$ = "Ok"
510  YEAR  = INT(DATE / 10000)
520  MONTH = INT( (DATE - YEAR*10000) / 100 )
530  DAY   = DATE - YEAR*10000 - MONTH*100
540  IF DAY   < 1 OR DAY   > 31 THEN ERROR.MESSAGE$ = "Bad day"
550  IF MONTH < 1 OR MONTH > 12 THEN ERROR.MESSAGE$ = "Bad month"
560  IF YEAR  < 0 OR YEAR  > 99 THEN ERROR.MESSAGE$ = "Bad year"
590  RETURN
596 :
598  REM ** Get a date
600  INPUT "Enter a date in the form YYMMDD"; DATE
605  IF DATE = 0 THEN 690
610  GOSUB 500 'Verify date INPUT
```

Section 7-1 *Problem No. 4 (continued)*

```
620  IF ERROR.MESSAGE$ <> "Ok" THEN PRINT ERROR.MESSAGE$ : GOTO 600
690  RETURN
696 :
698  REM ** Display day of the week
700  WEEK$ = "SunMonTueWedThuFriSat"
710  GOSUB 300 'Get first day of this month
720  WEEKDAY = ( WEEKDAY + DAY - 1 ) MOD 7
730  X = WEEKDAY * 3 + 1
740  PRINT
750  PRINT "Your date falls on "; MID$(WEEK$, X, 3)
790  RETURN

run
Enter a date in the form YYMMDD? 891301
Bad month
Enter a date in the form YYMMDD? 880101

Your date falls on Sun
```


Problem No. 6

```
90   MONTH$ = "JanFebMarAprMayJunJulAugSepOctNovDec"
100  INPUT "Enter a date in the form YYMMDD"; DATE
105  IF DATE = 0 THEN END
110  GOSUB 500 'Verify date INPUT
120  IF ERROR.MESSAGE$ <> "Ok" THEN PRINT ERROR.MESSAGE$ : GOTO 100
130  GOSUB 600 'Form the date string
140  PRINT NEW.DATE$
290  END
292 :
294  REM ** Calculate
296  '     WEEKDAY - 1st of Month
298  '     N - No. of days in Month
300  LEAP = 0 : IF YEAR MOD 4 = 0 THEN LEAP = 1
310  WEEKDAY = (YEAR + INT((YEAR + 3) / 4)) MOD 7
320  N = 0
330  FOR M = 1 TO MONTH
340   WEEKDAY = (WEEKDAY + N) MOD 7
350   N = 30 + ((M + (M > 7)) MOD 2)
360   IF M = 2 THEN N = 28 + LEAP
380  NEXT M
390  RETURN
496 :
498  REM ** Verify the date
```

```
196 :
200  INPUT "Enter a negative number in base ten form"; TEN
205  IF TEN > -1 THEN 200
210  DECIMAL = ABS(TEN)
220  X = DECIMAL MOD 2
230  A$ = RIGHT$(STR$(X),1) + A$
240  DECIMAL = DECIMAL \ 2
250  IF DECIMAL THEN 220
270  A$ = "0" + A$ : IF LEN(A$) < 16 THEN 270
280  PRINT "Binary form of "; ABS(TEN); "is ", A$
296 :
298  REM ** 1's to 0's and vice versa
300  FOR J1 = 1 TO LEN(A$)
310   X$ = MID$(A$,J1,1)
320   IF X$ = "0" THEN MID$(A$,J1,1) = "1"
                  ELSE MID$(A$,J1,1) = "0"
330  NEXT J1
396 :
398  REM ** Add 1
400  FOR J1 = LEN(A$) TO 1 STEP -1
410   X$ = MID$(A$,J1,1)
420   IF X$ = "0" THEN MID$(A$,J1,1) = "1" : GOTO 500
430    MID$(A$,J1,1) = "0"
440  NEXT J1
496 :
498  REM ** Display
500  PRINT "Two's complement form is", A$

run
Convert negative numbers to two's complement form

Enter a negative number in base ten form? 0
Enter a negative number in base ten form? -1
Binary form of  1 is        0000000000000001
Two's complement form is    1111111111111111
```

Chapter 8

Section 8-2

Problem No. 2

```
1    REM ** Enter these lines
118  TIMES = 0  'Count times missed
142  GOSUB 6000 'Is ALPHA1 on the sign?
144    IF CAPA <> 0 THEN 150
```

```
500  ERROR.MESSAGE$ = "Ok"
510  YEAR  = INT(DATE / 10000)
520  MONTH = INT( (DATE - YEAR*10000) / 100 )
530  DAY   = DATE - YEAR*10000 - MONTH*100
550  IF MONTH < 1 OR MONTH > 12 THEN ERROR.MESSAGE$ = "Bad month"
560  IF YEAR  < 0 OR YEAR  > 99 THEN ERROR.MESSAGE$ = "Bad year"
570  GOSUB 300 'Find number of days in this MONTH
580  IF DAY   < 1 OR DAY   > N  THEN ERROR.MESSAGE$ = "Bad day"
590  RETURN
596 :
598  REM ** Form the date string
600  Y$ = MID$(STR$(YEAR),2) : IF YEAR < 10 THEN Y$ = "0" + Y$
610  M$ = MID$(MONTH$,MONTH*3 - 2,3)
620  D$ = MID$(STR$(DAY ),2) : IF DAY  < 10 THEN D$ = "0" + D$
630  NEW.DATE$ = Y$ + "-" + M$ + "-" + D$
690  RETURN

run
Enter a date in the form YYMMDD? 890132
Bad day
Enter a date in the form YYMMDD? 880101
88-Jan-01
```


Section 7-3

Problem No. 2

```
100  PRINT "Convert base ten numbers to binary format"
110  PRINT
196 :
200  INPUT "Enter a value"; DECIMAL
210  X = DECIMAL - INT( DECIMAL / 2 ) * 2
220  A$ = STR$(X) + A$
230  DECIMAL = INT(DECIMAL / 2)
240  IF DECIMAL THEN 210
250  PRINT A$

run
Convert base ten numbers to binary format

Enter a value? 99
 1 1 0 0 0 1 1
```

Problem No. 4

```
100  PRINT "Convert negative numbers to two's complement form"
110  PRINT
145    IF TIMES <> 2 THEN 130
148    PRINT "You missed "; CHR$(ALPHA1); " twice" : GOSUB 1100
149    TIMES = 0 : ALPHA1 = ALPHA1 + 1 : GOTO 155
5070  TIMES = 0
5996 :
5998  REM ** Is ALPHA1 on the sign?
6000  FOR J = 1 TO LEN(SIGNS$(R))
6010   B$ = FNU$(MID$(SIGNS$(R),J,1))
6020   IF CHR$(ALPHA1) = B$ THEN 6050
6030  NEXT J
6040  GOTO 6090
6050  TIMES = TIMES + 1
6090  RETURN
```

Problem No. 4

```
80   DIM S(26) : FOR I = 1 TO 26 : S(I)=I : NEXT I
90   DIM ALPHA(26)
96  :
98   REM * Tabulate frequency of letters of the alphabet
100  OPEN "I", 1, "SIGNS.DAT"
102  INPUT #1, A$ : IF A$ = "Done" THEN 200
110  FOR I = 1 TO LEN(A$)
120   B$ = MID$(A$,I,1)
130   X = ASC( B$ )
135   IF X < 65 THEN 160
140    IF X > 90 THEN X = X - 32
150    ALPHA(X-64) = ALPHA(X-64) + 1
160  NEXT I
190  GOTO 102
196 :
198  REM ** Arrange in increasing order
200  N = 26
205  N = N - 1 : FLAG = 0
210  FOR I = 1 TO N
220   IF ALPHA(I) <= ALPHA(I+1) THEN 280
225    FLAG = 1
230    SWAP ALPHA(I), ALPHA(I+1)
240    SWAP S(I), S(I+1)
280  NEXT I
285  IF FLAG = 1 THEN 205
296 :
298  REM * Display frequency chart
300  FOR I = 1 TO 26
310   PRINT CHR$(S(I)+64); ALPHA(I),
320  NEXT I
```

Section 8-2 *Problem No. 4 (continued)*

```
390  END

run
B 1          C 1          J 1          K 1          Q 1
X 1          M 2          Z 2          F 3          P 3
W 3          G 2          V 5          H 6          U 6
D 8          S 8          Y 8          L 10         R 11
A 13         O 14         T 17         I 18         N 20
E 28
```

Section 8-3

Problem No. 2

```
1     REM ** Enter these lines
9200  IF INSTR(A$,"FOR") = 0 AND INSTR(A$,"NEXT") = 0 THEN 9290
9205  FOR J = 1 TO LEN(A$)
```

Section 8-4

Problem No. 2

```
1    REM ** Enter this line
140  INPUT #1, N$ : IF N$ <> N1$ THEN PRINT #2, N$
```

Problem No. 4

```
98   REM ** Add a name to a sequential file
100  OPEN "I", #1, "DEMO01.DAT"
110  OPEN "O", #2, "DEMO01.TMP"
120  INPUT "Delete a name"; N1$
140  INPUT #1, N$ : IF N$ <> N1$ THEN : PRINT #2, N$
150  IF N$ <> "End" THEN 140
160  CLOSE
170  KILL "DEMO01.DAT"
180  NAME "DEMO01.TMP" AS "DEMO01.DAT"
190  END
```

Chapter 9

Section 9-3

Problem No. 2

```
96  :
100  OPEN "R", #1, "ACNAMES.DAT", 30
901  DATA 2,  PP Taxes, Personal property taxes
902  DATA 9,  Medical,  Medical expenses
903  DATA 99, Misc,     Miscellaneous
904  DATA 22, S & W,    Sewer and water
905  DATA 38, C & M,    Cleaning and maintenance
906  DATA 44, Mortgage, Mortgage interest
990  DATA 0,  Done,     Done
```

Section 9-4

Problem No. 2

```
50   REM ** Display cities in % Growth order
79   REM ** Store the file record in POSITION
80   DIM ARRAY(10), POSITION(10)
96  :
100  OPEN "R", #1, "CITIES.DAT", 20
110  FIELD #1, 12 AS CITY$, 4 AS RANK$, 4 AS PERCENT$
196 :
198  REM ** First load the array with % Growth
200  FOR REC = 1 TO 10
210   GET #1, REC
220   G = CVS(PERCENT$)
230   ARRAY( REC ) = G
240   POSITION( REC ) = REC
250  NEXT REC
296 :
298  REM ** Now arrange according to % Growth
300  FOR LAST = 9 TO 1 STEP -1
310   FOR J = 1 TO LAST
320    IF ARRAY(J) <= ARRAY( J+1) THEN 350
330     SWAP POSITION(J), POSITION(J+1)
340     SWAP ARRAY(J),    ARRAY(J+1)
350   NEXT J
360  NEXT LAST
396 :
400  PRINT "City         Rank    % Growth"
410  FOR K = 1 TO 10
420   GET #1, POSITION( K )
430   R = CVS( RANK$ )
440   G = CVS( PERCENT$ )
450   PRINT USING "& ##       ###.#"; CITY$, R, G
480  NEXT K
490  CLOSE #1
890  END
```

```
110  FIELD #1, 30 AS X$
196 :
198  REM ** Request new account
200  INPUT "New account #, label"; N, A$
220  IF N>0 AND N<100 THEN 300
230   PRINT "Account number out of range" : GOTO 200
240  IF LEN(A$) <= 30 THEN 300
250   PRINT "Label too long" : GOTO 200
296 :
298  REM ** Go into the file
300  GET #1, N
310  IF LEFT$(X$,10) = "Unassigned" THEN 340
320   PRINT "That account number is in use" : GOTO 200
340  LSET X$ = A$
350  PUT #1, N
390  CLOSE
395  END
```

Problem No. 4

```
80   REM ** Initialize account label file
96  :
100  OPEN "R", #1, "ACNAMES.DAT", 38
110  FIELD #1, 30 AS X$, 8 AS L$
196 :
198  REM ** Fill file with "Unassigned"
200  LSET X$ = "Unassigned"
205  LSET L$ = "Empty"
210  FOR REC = 1 TO 99
220   PUT #1, REC
230  NEXT REC
296 :
298  REM ** Write out actual labels
300  READ N, S$, N$
310  IF N$ = "Done" THEN 390
320   IF N < 1 OR N > 99 THEN 380
330    LSET L$ = S$
340    LSET X$ = N$
350    PUT #1, N
360    GOTO 300
380   PRINT N; "Out of range"
390  CLOSE #1
395  END
896 :
900  DATA 1,  R Taxes,  Real estate taxes
```

```
run
City          Rank    % Growth
Detroit          6      -20.5
Philadelphia     4      -13.4
Baltimore        9      -13.1
Chicago          2      -10.8
New York         1      -10.4
Los Angeles      3        5.5
Dallas           7        7.1
San Antonio     10       20.1
San Diego        8       25.5
Houston          5       29.2
```

Chapter 10

Section 10-1

Problem No. 2

```
19   REM ** mailing list program
20   FILENAME$ = "NAMES"
30   DIM LABEL$(9), L(9), F$(9), KDATA$(9)
396 :
398  REM ** Control routine for editing names
400  GOSUB 2000 'Read data labels and limits
410  GOSUB 1900 'Read available-space parameters .ZER file
420  GOSUB 1800 'OPEN the .DAT file
430  GOSUB 4000 'Request ID to edit
440  IF ID = 0 THEN CLOSE : END
                'Terminate on zero ID
450  GOSUB 4100 'Read the entry and edit it
460  GOTO 430   'Do it again
1796 :
1798  REM ** OPEN the .DAT file
1800  OPEN "R", #2, FILENAME$ + ".DAT", RLENGTH
1810  FIELD #2, L(1) AS F$(1), L(2) AS F$(2), L(3) AS F$(3),
                L(4) AS F$(4), L(5) AS F$(5), L(6) AS F$(6),
                L(7) AS F$(7), L(8) AS F$(8), L(9) AS F$(9)
1890  RETURN
1896 :
1898  REM ** Read available-space parameters
1900  OPEN "R", #1, FILENAME$ + ".ZER", 8
1910  FIELD #1, 4 AS NEWID$, 4 AS OLDID$
1920  GET #1, 1
1930  NS = CVS(NEWID$)
1940  DS = CVS(OLDID$)
```

Section 10-1 *Problem No. 2 (continued)*

```
1990  RETURN
1996 :
1998  REM ** Read data labels and limits
2000  READ N0
2010  RLENGTH = 0
2020  FOR X9 = 1 TO N0
2030   READ LABEL$(X9), L(X9)
2040   RLENGTH = RLENGTH + L(X9)
2050  NEXT X9
2090  RETURN
2096 :
2098  REM ** DATA - labels and limits
2100  DATA 9
2102  DATA  ID #,     4
2104  DATA  CODE,     2
2106  DATA  LAST,    20
2108  DATA  FRST,    20
2110  DATA  ADDR,    30
2112  DATA  CITY,    20
2114  DATA  STAT,     2
2116  DATA "ZIP ",    5
2118  DATA  PHON,    17
3996 :
3998  REM ** Request ID to edit
4000  PRINT
4010  INPUT "EDIT ID #"; ID
4020  IF ID < NS AND ID >= 0 THEN 4090
4030   PRINT "NON-EXISTENT ID" : GOTO 4000
4090  RETURN
4096 :
4098  REM ** Read the entry if it is real
4100  GET #2, ID
4110  X = CVS(F$(1)) : IF X = ID THEN 4125
4120  PRINT ID; "Has been deleted" : GOTO 4190
4125  FOR I9 = 2 TO N0
4130   PRINT LABEL$(I9); ": "; F$(I9);
4135   PRINT TAB(L(I9)+8); ": OK"; : INPUT AN$
4140   IF LEFT$(AN$,1) = "Y" THEN 4185
4145    IF LEFT$(AN$,1) = "N" THEN 4160
4150     PRINT "'Y' OR 'N' Please" : GOTO 4135
4160    INPUT "    : ", KDATA$(I9)
4165    IF LEN(KDATA$(I9)) <= L(I9) THEN 4180
4170    PRINT "Too long" : GOTO 4160
4180    LSET F$(I9) = KDATA$(I9)
```

```
2100  DATA 9
2102  DATA  ID #,     4
2104  DATA  CODE,     2
2106  DATA  LAST,    20
2108  DATA  FRST,    20
2110  DATA  ADDR,    30
2112  DATA  CITY,    20
2114  DATA  STAT,     2
2116  DATA "ZIP ",    5
2118  DATA  PHON,    17
5996 :
5998  REM ** Request up to 10 ID's here
6000  PRINT "Enter up to 10 ID's"; : LINE INPUT A$
6005  A$ = A$ + ","
6010  FOR I9 = 1 TO 10 : ID(I9) = 0 : NEXT I9
6020  N1 = 0
6030  X = INSTR(A$,",") : IF X = 0 THEN 6090
6040   X9 = VAL(LEFT$(A$,X-1))
6050   IF X9 < NS AND X9 > 0 THEN 6070
6060    PRINT X9; "Out of range - reenter"; : INPUT X9 : GOTO 6050
6070   N1 = N1 + 1 : ID(N1) = X9
6080   IF N1 < 10 THEN A$ = MID$(A$,X+1) : GOTO 6030
6090  RETURN
6096 :
6098  REM ** Display labels here
6100  FOR I9 = 1 TO N1
6110   ID = ID(I9)
6120   GET #2, ID
6130   X = CVS(F$(1)) : IF X <> ID THEN 6180
6135   PRINT USING " & &"; F$(4), F$(3)
6140   PRINT USING " &"; F$(5)
6150   PRINT USING " & & &"; F$(6), F$(7), F$(8)
6160   PRINT : PRINT : PRINT
6180  NEXT I9
6190  RETURN
```

Chapter 11

Section 11-1

Problem No. 2

```
1     REM ** Load program 11-2 and
2     REM Type the following:
180
100
120   COLOR = INT(RND(1)*16)
```

```
4185  NEXT I9
4187  PUT #2, ID
4190  RETURN
```

Problem No. 4

```
19    REM ** mailing list program
20    FILENAME$ = "NAMES"
30    DIM LABEL$(9), L(9), F$(9), KDATA$(9)
496 :
498   REM ** Control routine for displaying labels
500   GOSUB 2000 'Read data labels and limits
510   GOSUB 1900 'Read available-space parameters .ZER file
520   GOSUB 1800 'OPEN the .DAT file
530   GOSUB 6000 'Request up to 10 ID numbers
535   IF ID(1) = 0 THEN CLOSE : END
                 'Terminate on no requested numbers
540   GOSUB 6100 'Display address labels for the requested ID's
560   CLOSE : END
1796 :
1798  REM ** OPEN the .DAT file
1800  OPEN "R", #2, FILENAME$ + ".DAT", RLENGTH
1810  FIELD #2, L(1) AS F$(1), L(2) AS F$(2), L(3) AS F$(3),
                L(4) AS F$(4), L(5) AS F$(5), L(6) AS F$(6),
                L(7) AS F$(7), L(8) AS F$(8), L(9) AS F$(9)
1890  RETURN
1896 :
1898  REM ** Read available-space parameters
1900  OPEN "R", #1, FILENAME$ + ".ZER", 8
1910  FIELD #1, 4 AS NEWID$, 4 AS OLDID$
1920  GET #1, 1
1930  NS = CVS(NEWID$)
1940  DS = CVS(OLDID$)
1990  RETURN
1996 :
1998  REM ** Read data labels and limits
2000  READ N0
2010  RLENGTH = 0
2020  FOR X9 = 1 TO N0
2030   READ LABEL$(X9), L(X9)
2040   RLENGTH = RLENGTH + L(X9)
2050  NEXT X9
2090  RETURN
2096 :
2098  REM ** DATA - labels and limits
```

Problem No. 4

```
10    REM ** Random colors at random points
100   GR 0
110   X = INT(RND(1)*39)
120    Y = INT(RND(1)*39)
130    COLOR = INT(RND(1)*16)
150    PLOT X, Y
170    GOTO 110
```

Problem No. 6

```
100   GR
110   HOME
120   PRINT "  ";
130   COLOR = 15
140   FOR I = 1 TO 9
150    READ A
160    VLIN 39 - I, 39 AT 3 * I
170    PRINT I;
180   NEXT I
190   PRINT "  Days"
200   PRINT "Daily temperature"
210   END
296 :
300   DATA 30, 26, 26
310   DATA 31, 26, 30
320   DATA 38, 36, 34
```

Section 11-2

Problem No. 2

```
98    REM ** Plot a one and a three
100   GR 0
196 :
200   COLOR = 15
210   FOR I = 11 TO 15
220    VLIN 1,7 AT I
230    VLIN 3,9 AT I + 10
240   NEXT I
296 :
300   COLOR = 0
310   PLOT 13,4 'Plot the "one"
330   PLOT 22,4 'Begin the "three"
340   PLOT 23,6
350   PLOT 24,8
```

Section 11-3

Problem No. 2

```
50   RANDOMIZE
98   REM ** Display a random die face
100  GR 0
196 :
200  COLOR = 15
210  FOR I = 0 TO 4
220   VLIN 0,6 AT I
230  NEXT I
296 :
300  COLOR = 0
310  R = INT(RND(1)*6) + 1
320  ON R GOSUB 1100,1200,1300,1400,1500,1600
390  END
1096 :
1098  REM ** Plot a 'one'
1100  PLOT 2,3
1190  RETURN
1196 :
1198  REM ** Plot a 'two'
1200  PLOT 1,1
1210  PLOT 3,5
1290  RETURN
1296 :
1298  REM ** Plot a 'three'
1300  PLOT 1,1
1310  PLOT 3,5
1320  PLOT 2,3
1390  RETURN
1396 :
1398  REM ** Plot a 'four'
1400  PLOT 1,1
1410  PLOT 1,5
1420  PLOT 3,1
1430  PLOT 3,5
1490  RETURN
1496 :
1498  REM ** Plot a 'five'
1500  PLOT 1,1
1510  PLOT 1,5
1520  PLOT 3,1
1530  PLOT 3,5
1540  PLOT 2,3
1310  PLOT X+3,Y+5
1320  PLOT X+2,Y+3
1390  RETURN
1396 :
1398  REM ** Plot a 'four'
1400  PLOT X+1,Y+1
1410  PLOT X+1,Y+5
1420  PLOT X+3,Y+1
1430  PLOT X+3,Y+5
1490  RETURN
1496 :
1498  REM ** Plot a 'five'
1500  PLOT X+1,Y+1
1510  PLOT X+1,Y+5
1520  PLOT X+3,Y+1
1530  PLOT X+3,Y+5
1540  PLOT X+2,Y+3
1590  RETURN
1596 :
1598  REM ** Plot a 'six'
1600  PLOT X+1,Y+1
1610  PLOT X+1,Y+3
1620  PLOT X+1,Y+5
1630  PLOT X+3,Y+1
1640  PLOT X+3,Y+3
1650  PLOT X+3,Y+5
1690  RETURN
```

Problem No. 6

```
98   REM ** 'Roll the dice'
100  GR 0
105  GOSUB 500
110  R = INT(RND(1)*6) + 1
120  X = 0 : Y = 32
125  COLOR = 15 : GOSUB 200 : GOSUB 300
140  R = INT(RND(1)*6) + 1
150  X = 10 : Y = 30
160  COLOR = 15 : GOSUB 200 : GOSUB 300
190  END
196 :
200  FOR I = X TO X + 4
220   VLIN Y,Y+6 AT I
230  NEXT I
290  RETURN
296 :
```

```
1590  RETURN
1596 :
1598  REM ** Plot a 'six'
1600  PLOT 1,1
1610  PLOT 1,3
1620  PLOT 1,5
1630  PLOT 3,1
1640  PLOT 3,3
1650  PLOT 3,5
1690  RETURN
```

Problem No. 4

```
10   RANDOMIZE 5
98   REM ** Display two dice at random
100  GR 0
110  R = INT(RND(1)*6) + 1
120  X = 0 : Y = 32
130  GOSUB 200 : GOSUB 300
140  R = INT(RND(1)*6) + 1
150  X = 10 : Y = 30
160  GOSUB 200 : GOSUB 300
190  END
196 :
200  COLOR = 15
210  FOR I = X TO X + 4
220   VLIN Y,Y+6 AT I
230  NEXT I
290  RETURN
296 :
300  COLOR = 0
320  ON R GOSUB 1100,1200,1300,1400,1500,1600
390  RETURN
1096 :
1098  REM ** Plot a 'one'
1100  PLOT X+2,Y+3
1190  RETURN
1196 :
1198  REM ** Plot a 'two'
1200  PLOT X+1,Y+1
1210  PLOT X+3,Y+5
1290  RETURN
1296 :
1298  REM ** Plot a 'three'
1300  PLOT X+1,Y+1
```

```
300  COLOR = 0
320  ON R GOSUB 1100,1200,1300,1400,1500,1600
390  RETURN
496 :
498  REM ** The rolling dice
500  FOR X9 = 1 TO 10
510   X = INT(RND(1)*33)
520   Y = INT(RND(1)*35)
530   COLOR = 15 : R = INT(RND(1)*6)+1
540   GOSUB 200 : GOSUB 300
560   COLOR = 0 : GOSUB 200
580  NEXT X9
590  RETURN
1096 :
1098  REM ** Plot a 'one'
1100  PLOT X+2,Y+3
1190  RETURN
1196 :
1198  REM ** Plot a 'two'
1200  PLOT X+1,Y+1
1210  PLOT X+3,Y+5
1290  RETURN
1296 :
1298  REM ** Plot a 'three'
1300  PLOT X+1,Y+1
1310  PLOT X+3,Y+5
1320  PLOT X+2,Y+3
1390  RETURN
1396 :
1398  REM ** Plot a 'four'
1400  PLOT X+1,Y+1
1410  PLOT X+1,Y+5
1420  PLOT X+3,Y+1
1430  PLOT X+3,Y+5
1490  RETURN
1496 :
1498  REM ** Plot a 'five'
1500  PLOT X+1,Y+1
1510  PLOT X+1,Y+5
1520  PLOT X+3,Y+1
1530  PLOT X+3,Y+5
1540  PLOT X+2,Y+3
1590  RETURN
1596 :
1598  REM ** Plot a 'six'
```

Section 11-3 *Problem No. 6 (continued)*

```
1600  PLOT X+1,Y+1
1610  PLOT X+1,Y+3
1620  PLOT X+1,Y+5
1630  PLOT X+3,Y+1
1640  PLOT X+3,Y+3
1650  PLOT X+3,Y+5
1690  RETURN
```

Chapter 12

Section 12-1

Problem No. 2

```
100  HGR
106 :
108  REM ** Prepare text window
110  HOME : VTAB 21
116 :
118  REM ** White border
120  HCOLOR = 11 : GOSUB 600
130  HCOLOR = 2
196 :
198  REM ** Plot graph here
200  FOR DAY = 1 TO 7
210   READ D$, T
220   FOR I = 0 TO 7
230    X = 35 * DAY + I - 3
240    Y = 157 - 3 * T
250    HPLOT X, Y TO X, 157
270   NEXT I
275   PRINT TAB(DAY*5); D$;
280  NEXT DAY
590  END
596 :
598  REM ** Plot a border
600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
690  RETURN
696 :
700  DATA Sun, 42,  Mon, 38
710  DATA Tue, 40,  Wed, 31
720  DATA Thu, 24,  Fri, 18
730  DATA Sat, 15
600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
690  RETURN
696 :
710  DATA Mon, 33.75
720  DATA Tue, 35.125
730  DATA Wed, 35
740  DATA Thu, 36.25
750  DATA Fri, 37.875
```

Section 12-2

Problem No. 2

```
100  HGR 1
110  GOSUB 200
190  END
196 :
198  REM ** Line plotting routine
200  READ C,X,Y,X1,Y1
210  IF C = -1 THEN 290
220   HCOLOR = C
230   HPLOT X, Y TO X1, Y1
240  GOTO 200
290  RETURN
996 :
998  REM ** Sailboat data
1000  DATA 11,30,40,40,50
1002  DATA 11,30,40,30,52
1004  DATA 11,30,50,40,50
1006  DATA 11,26,52,46,52
1008  DATA 11,26,52,28,56
1010  DATA 11,28,56,44,56
1012  DATA 11,44,56,46,52
1990  DATA -1,0,0,0,0
```

Problem No. 4

```
100  HGR 1 : HCOLOR = 11
196 :
198  REM ** Plot the flagpole
200  HPLOT 31, 10 TO 31, 150
296 :
298  REM ** Draw the flag
300  FOR H = 140 TO 12 STEP -2
310   HCOLOR = 11
320   HPLOT 35, H      TO 70, H
```

Problem No. 4

```
100  HGR
106 :
108  REM ** Prepare text window
110  HOME : VTAB 21
116 :
118  REM ** White border
120  HCOLOR = 11 : GOSUB 600
130  HCOLOR = 2
196 :
198  REM ** Plot graph here
200  FOR DAY = 1 TO 5
210   READ D$, P
220   FOR I = 0 TO 7
230    X = 35 * DAY + I - 3
240    Y = 157 - 4*P
250    HPLOT X, Y TO X, 157
270   NEXT I
275   PRINT TAB(DAY*5); D$;
280  NEXT DAY
590  END
596 :
598  REM ** Plot a border
330   HPLOT 35, H + 8  TO 49, H + 8
340   HPLOT 35, H + 1  TO 36, H + 1
400   HCOLOR = 10
420   HPLOT 35, H + 18 TO 70, H + 18
430   HPLOT 37, H + 2  TO 49, H + 2
440   HPLOT 35, H + 17 TO 36, H + 17
480   IF RND(1) > .9 THEN FOR K = 1 TO 100 : NEXT K
490  NEXT H
```

Section 12-3

Problem No. 2

```
1     REM ** Simply replace line 160 in program 12-9
2     REM You might try it without dividing by 100
160   DEF FNF(X) = (X^2+50*X-450)/100
```

Section 12-4

Problem No. 2

```
1     REM ** Simply replace line 210 in program 12-11c
2     REM with any of the listed equations
```

Index

6263-X

Microsoft™ BASIC Using the SoftCard™

JAMES S. COAN

A complete guide to programming in Microsoft BASIC-80 using the SoftCard on Apple II Plus and IIe computers. The author begins with simple, concise programs that introduce the reader to various features of the language, and then he gradually moves on to more complex programs that illustrate problem solving. Nearly 100 programs are presented and discussed, and problems included at appropriate points in the text make it suitable for the classroom as well as for the home.

Topics covered include a comparison between Microsoft BASIC-80 and Applesoft BASIC, a thorough explanation of low-resolution and high-resolution graphics, and a review of the various statements, variables, and functions of the SoftCard.

Each chapter is followed by a "Sidelight" section that presents special features, concepts, and advanced techniques. Appendixes include an ASCII chart, an index of programs, and solutions to even-numbered problems.

Other Books of Interest...

GETTING STARTED WITH CP/M®

Rob Patten and Paul Calandrino

A step-by-step initiation into using the world's most popular microcomputer operating system. Explains what CP/M is and what an operating system does, presents a detailed walk-through of a productive working session, and provides answers to most beginners' questions. The book also offers a concise summary of all operating system commands and includes a handy pull-out Command Reference Card to keep by your machine. #5208-1, paper, 112 pages.

GETTING THE MOST FROM YOUR MICRO

Ernest E. Mau

This valuable sourcebook shows you how to take care of all your computer equipment to ensure trouble-free operation. It provides specific preventive steps you can use to keep bothersome bugs out of your system, and it includes easy-to-follow instructions that enable you to fix many minor technical problems yourself. #6264-8, paper, 288 pages.

HAYDEN BOOK COMPANY, INC.
Hasbrouck Heights, New Jersey

ISBN 0-8104-6263-X