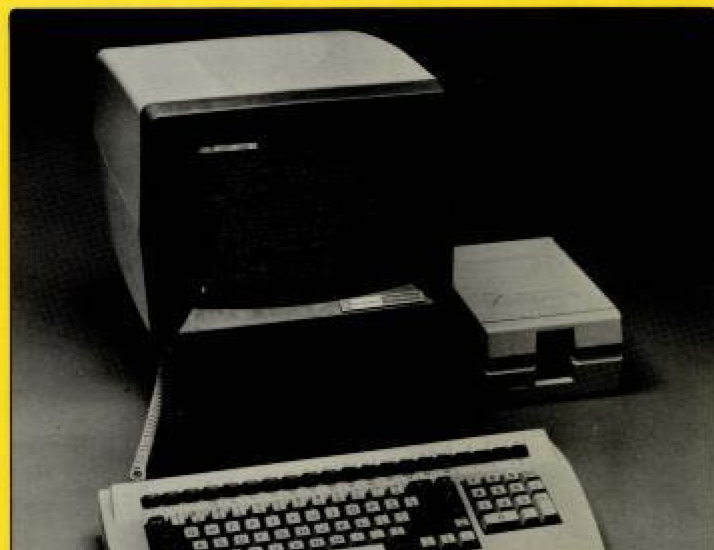
WAYNE T. WATSON

An Introduction to Structured BASIC for the Cromemco C-10



Fully authorized by Cromemco, Inc.

An Introduction to

Structured BASIC for the Cromemco C-10

An Introduction to

Structured BASIC for the Cromemco C-10

By Wayne T. Watson

Macmillan Publishing Company
New York
Collier Macmillan Publishers
London

Macmillan Publishing Company
866 Third Avenue, New York, New York 10022

Collier Macmillan Canada, Inc.

Library of Congress Cataloging in Publication Data

Watson, Wayne T.
An introduction to structured BASIC for the Cromemco C-10.

Includes index.
1. Cromemco C-10 (Computer)—Programming.
2. BASIC (Computer program language) 3. Structured programming. I. Title.
QA76.8.C76W37 1984 001.64'2 83-19328
ISBN 0-02-424580-1

Printing: 1 2 3 4 5 6 7 8 Year: 4 5 6 7 8 9 0 1 2 3

ISBN 0-02-424580-1

To

Karen, David and Greta
Mother, Dad, Marianne, and David
Rick, Keith, Kelly
Gerd, Vera, Alf and Dan

And

The Sierra, Grand Canyon
and Wilderness Memories

Contents

Preface

This book is an introduction to the Cromemco* Structured BASIC, or Structured BASIC, language, which is available for use only on Cromemco microcomputers. It is for individuals who want to know more about Structured BASIC. Knowledge of the CDOS or CROMIX operating system commands is assumed.

It is the intention of this book to describe and organize Structured BASIC in a way that will be useful to you, regardless of whether you are just starting to use computers, you have previous experience with computers, or you have already worked with Structured BASIC. Readers who do not have Structured BASIC available to them will find many useful examples and ideas which can be extended to other versions of BASIC.

A good deal of emphasis is placed on making useful information readily available, and in progressing from beginning material to more advanced material. Sample programs are provided to illustrate the concepts presented. At the end of each chapter, a summary of the key points of the chapter is given. The summary can be useful to the first-time user for a review of some of the concepts learned in the chapter. Review questions and exercises are found at the end of each chapter to emphasize concepts learned in the chapter. Furthermore, the organization of the book lends itself to a quick and handy reference to the concepts of Structured BASIC for both the beginning and the advanced user.

*The names Cromemco, CDOS, and CROMIX are trademarks of Cromemco Inc., Mountain View, Ca.

Much of the material in this introduction covers features which are common to many programming languages, and should help readers extend their general knowledge of computer languages when it is mastered.

An extension to the Structured BASIC language is a feature referred to as the keyed sequential access method feature, or KSAM. It is available in a version of Structured BASIC known as 32K Structured BASIC. It is not available in Structured BASIC for the C-10. While KSAM is undoubtedly of interest to some users, we feel that is of limited use in many applications, and is only of interest to some advanced users. There are virtually no differences between Structured BASIC for the C-10 personal computer and 32K Structured BASIC for other Cromemco computers except for KSAM. This book is applicable to either version of BASIC. Users of 32K Structured BASIC have the additional ability of tailoring BASIC to different operating environments.

A detailed description of Structured BASIC and 32K Structured BASIC are contained in the Cromemco Structured BASIC Instruction Manual (part no. 023-4050) and the 32K Structured BASIC Instruction Manual (part no. 023-0080), respectively, both available from Cromemco. Both manuals are good sources of additional information, and should be used when the user needs to know more details of the actual implementation of BASIC. For the 32K Structured BASIC user, KSAM, device drivers, and some other advanced features of BASIC are discussed at length in the 32K manual.

For the first-time user of BASIC or computers, we suggest that you start with Chapter 1 and continue through Chapter 9. When you have completed Chapter 9, you will have a solid introduction to the most used concepts in BASIC. By the time you have finished Chapter 7, you should be in a position to write some fairly complex programs. We urge you to try to program some of your own applications by then. If you are unsure of tackling your own problems, try modifying some of the larger examples in the book. The loan repayment, scatter plot, and histogram programs are useful for this purpose. For other ideas, see the suggestions at the end of Chapter 7. Remember, one of the best ways to learn how to program is to practice. If you have a hard time thinking of applications, try converting programs from other BASIC books into Structured

BASIC. Take a peek at Chapter 10 occasionally as you feel more comfortable with BASIC.

When you feel fairly confident about your abilities, move on to Chapter 11 on files. Take some time to really get acquainted with file operations; they are important to many applications. As you increase your abilities, move on to the last three chapters. Make use of subroutines (Chapter 9) when you feel that you are using a lot of the same codes in your programs. When you feel comfortable with subroutines, try Procedures (Chapter 14). Procedures are a really good feature of Structured BASIC and are not found in many other implementations of BASIC.

It is a pleasure to thank my wife, Greta, for her careful proofreading of the book for spelling and punctuation errors; Jim Drebert for his interest in and encouragement for the project; Charles Silbereisen for his attention to detail and his interest; and a number of friends who indirectly made this project worthwhile. Despite their heroic attempts, the problems and errors which remain are attributable to me.

Wayne T. Watson
The Software Hill
Mountain View, California

CHAPTER 1

Introduction

BASIC

As you are about to discover, Structured BASIC is a very powerful and useful language for solving problems using your microcomputer. The BASIC language has been available in a number of forms since the mid-sixties, when it was developed at Dartmouth University. Originally, BASIC was designed to provide a simple and easy-to-use language for solving problems using a computer. Since that time, its capabilities have grown. The Structured BASIC language extends the original BASIC concept to include modern features. At the same time, it continues to provide a simple and easy-to-use tool for solving problems.

Structured BASIC is known as a higher-level language (HLL). Other such languages are FORTRAN, COBOL, APL, PL/I, and Pascal. What a HLL means to you is that you will not have to know much about the inner workings of your computer. Other languages, usually assembly languages, require a detailed knowledge of what goes on inside the computer. A HLL allows you to get quickly at concepts and instructions that help you solve your computer application problems.

Another aspect of Structured BASIC is that it is also an interpretive language. This feature allows you essentially to stop your program, a set of instructions written in the BASIC language, in midstream and to make modifications, corrections, and additions and then continue with your computing from the point where you stopped. This is very useful when you are beginning to explore the language or develop a program. Structured BASIC uses an interpreter to execute your programs; many other languages use compilers. Compilers have the disadvantage that corrections are not so easily made. However, in general, compiled programs execute faster than interpreted programs.

PROGRAMMING

Now that you know something about the history and general capabilities of Structured BASIC, you may be wondering about what you are actually going to do with it to help you solve your problems.

Structured BASIC is a language. Specifically, it is a computer language. What this means is that it is composed of a set of instructions and commands for solving problems. Problems may be as diverse as differential equation solving, data plotting, information retrieval, data base management, forecasting, payroll accounting, and inventory control. A computer language provides a way of communicating certain instructions to a computer that will help solve these and other problems. The organization of the instructions in a group constitutes a program; the act of organizing the instructions is referred to as programming (writing a program). Writing a program, or programming, is somewhat like writing in a natural language such as English, except that the rules are different and much more restrictive than in a natural language. Structured BASIC is the language you will use to write programs. Programming is what this book is all about.

To give you a slightly more concrete look at what programming is about, consider the task of cleaning a window. Although computers do not wash windows, this example illustrates closely the process of communicating a task to a computer. We can write this task as a series of instructions to give to someone to perform, maybe your son or other family member, if you are lucky.

Here is a set of possible instructions:

1. Get a bucket, water, soap, and wash rag.
2. Go to the living room.

3. Select a window.
4. Wash the selected window.
5. Select another window and repeat instruction 4. If there are no other windows unwashed go to instruction 6.
6. Empty the dirty water in the laundry sink.
7. Return the bucket, soap, and wash rag to where you found them before performing instruction 1.

By way of analogy, these instructions constitute a program. Writing them is the act of programming. The written instructions are a program. The language is English. Later we will use BASIC. The person carrying out the instructions represents the computer.

Perhaps you are wondering why we did not just use one instruction that said, "John, go get some water and wash the living room windows, and make sure you return the equipment to where you found it!" This would make sense to humans, but computers are not quite as smart. We have to break things down into sets of smaller instructions that computers can deal with. The set of instructions a computer deals with is incorporated in languages such as Structured BASIC. Structured BASIC allows us to deal with one set of instructions. Another programming language might allow us to deal with another set. We have to understand each language so that we can determine how to provide the computer with the right set of instructions.

Besides instructions, we will have to provide data and information about the data. In our example, we might want to say that the soap is SNAPPY-WINDOW cleaner and the rag is a paper towel. So we might insert a statement such as

A. The soap is SNAPPY-WINDOW cleaner.
B. The wash rag is a paper towel.

These declarations are really defining data or some property of the data. At any rate, they are not instructions in the sense of instructions 1 through 7 given earlier.

Generally, we will refer to declarations, definitions, and instructions as statements. Statements are a very important aspect of a programming language. They are not the only part, but they deserve much attention.

Although we will not deal with the techniques of writing programs, it should be mentioned for the beginner that a sharp pencil and some paper are often used to sketch out a program before it is placed in the computer. By sketching, we mean that a loosely written set of instructions is written which may look something like the actual statements that are needed by the computer. Once we are satisfied that the loosely written instructions represent a solution to our problem, we enter the actual program at the terminal. Usually, a combination of sketching a program on paper and directly entering specific program statements at the terminal are used. Another technique often used is a flow diagram. A flow diagram is just a figure containing blocks and lines. These figures generally indicate logic flow, computational blocks, and data movement. This is probably the oldest form of describing programs and is often used. Figure 1.1 is a simple example of a flow diagram which shows how to find an ace in a deck of cards. We assume that the deck is face up on a table.

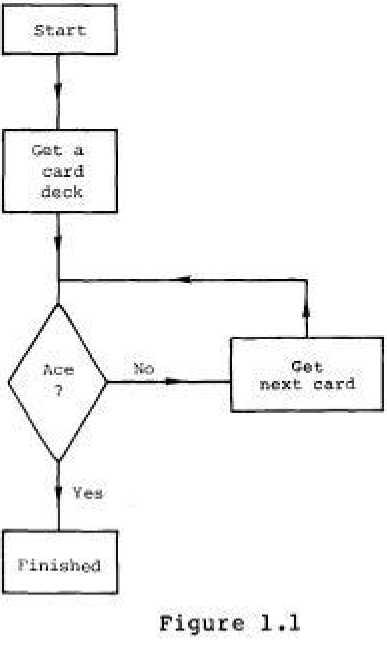


Figure 1.1

HIPO (Hierarchy-Input-Process-Output) charts (introduced by IBM) and Structured charts (introduced by Yourdon and Constantine) are other ways of describing programs. Pseudo-Design Languages (PDL) are yet another popular method. Unless you are fairly experienced at programming, you will not find these latter methods appealing.

Like any experience, programming involves making mistakes. In fact, it may seem that a major portion of our concern in writing programs involves errors. At the present state of human and computer development, this seems an unavoidable problem. There will be times when mistakes occur, but they can be corrected. Structured BASIC provides facilities for finding and fixing errors, and a chapter of this book is devoted to the topic. If you are just beginning to program, do not let your errors discourage you. With perseverance you will overcome them and you will soon be making fewer errors and moving toward solving larger and more complex problems. Just keep on truckin'.

Figure 1.2 Time to get crackin'.

Review Questions and Excercises

1. Can you draw a flow diagram for finding a two of spades and the first ace in the deck? Does it require one or two diamond figures in the diagram? Can you explain your diagram to someone? Does it allow both cards to be found?

2. If someone painted SLOH on a road sign, would you understand that it meant SLOW? Would a computer? Computer languages usually require that important words be spelled correctly. Do you think BASIC will allow you to misspell words in its instructions?

3. In Chapter 2 we will find that BASIC instructions are identified by simple names such as PRINT, LET, IF, and GOTO. These are very powerful statements in BASIC. Does this surprise you? What do you suppose they do?

4. If you make a mistake in writing a program, do you suppose that it cannot be corrected? Would you have to buy something to fix such a mistake as, say, in repairing a broken window?

Summary

BASIC
: A programming language.

Program
: A set of instructions written in a programming language which direct a computer to perform and solve an application problem.

Interpreter
: A particular implementation of a computing language which allows programs to be stopped in mid stream. This ability allows you to make modifications, corrections, and additions, and then to continue with your program without starting from the beginning again.

Structured BASIC
: An interpretive programming language implemented for use on Cromemco computer systems.

CHAPTER 2

Starting into Structured BASIC

Structured BASIC contains nearly one hundred statements which allow us to state our problem to the computer. Many of these statements need not be learned at first. We can gradually learn how to use all of them; we learn by starting with the simpler and most frequently used statements first and then by moving to others as our experience grows. The same approach can be used in learning other parts of the language.

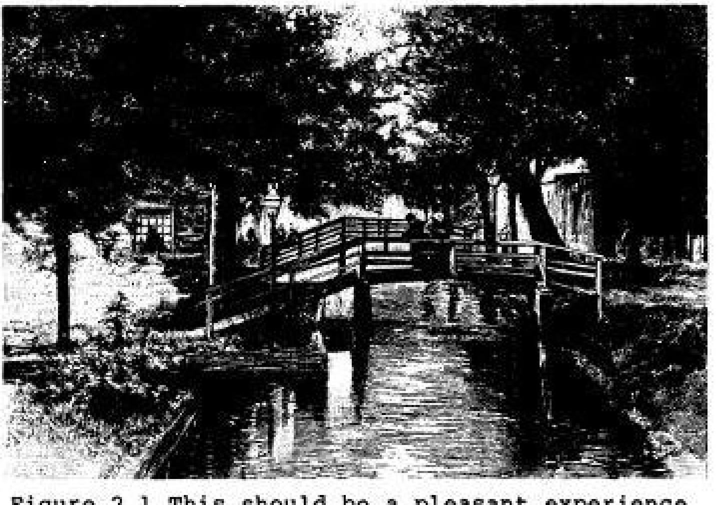


Figure 2.1 This should be a pleasant experience.

To get started on this process, we will go directly to Structured BASIC and use it. We will introduce you to what you need as we go along, and eventually turn you loose in succeeding chapters to explore and understand more on your own.

BECOMING FAMILIAR WITH BASIC

The first thing you will want to do is simply enter the Structured BASIC interpreter; so fire up your computer and let's get started.

The interpreter is nothing but a special program that allows you to operate BASIC. It is delivered with your copy of Structured BASIC on a floppy disk, and the program is called SBASIC. Look for a file called SBASIC.COM with your CDOS or CROMIX commands. Put this file on a disk which is not write-protected. If you are not sure what "write-protected" means or how to transfer files from one disk to another, see your CDOS or CROMIX manual.

Once you have placed your floppy disk containing SBASIC.COM in a drive, enter the following at the keyboard of your terminal

```
SBASIC
```

(From a C-10, select option 5 from your main menu.) Make sure you hit the carriage return on your keyboard afterwards. We will not say much more about hitting carriage returns, but unless we say otherwise, when you are asked to enter something, it needs to be followed by a carriage return. This signals the computer that you want something done.

We will refer to the process, just described, of running the BASIC interpreter as "entering, running, or executing BASIC".

The computer should respond with

```
>>
```

This is BASIC's way of telling you that it is ready for you to enter something. Hit the carriage return several times and notice that BASIC responds with >> each time.

In our description of BASIC, you will be asked from time to time to enter something at the terminal. In the text, we will show this by printing the text you should enter in bold characters. For example,

```
        >>PRINT 1.5
```

indicates that you should enter "PRINT 1.5" at your terminal console.

Before going very far, let's find out how to get out of BASIC and back to the operating system, CDOS or CROMIX. Just enter the BYE command

```
        >>BYE
```

Presto. Nothing to it. We are out of BASIC interpreter. Re-enter BASIC by typing in SBASIC, and we will continue.

Now, let's enter a program. Enter the following exactly as shown

```
        >>100 PRINT "HELLO"
        >>110 END
```

If you make a mistake and BASIC prints ERROR-1 SYNTAX or something similar, just re-enter the line of text. If you misspell something before hitting the carriage return, just back up your cursor with the delete or left-cursor-arrow key on your terminal and continue typing once the misspelled character is erased by the key. If you misspelled a line number, eliminate the line by entering just the mistyped line number with a carriage return, and then enter the correct line. We will delve more into deleting and correcting lines shortly.

You have just entered a two-line program. Each line begins with a number, and represents a program statement. The first item after the line number is the name of the statement, in this case the PRINT and END statements.

The PRINT statement usually contains some information that you wish to print at the terminal. The items to be printed follow the word PRINT. Strings of alphabetic and non-alphabetic characters or strings to be printed are enclosed in double quotes ("). In later chapters, we will learn that the PRINT statement has many more capabilities. The END statement simply marks the end of the program statements and stops execution of the program when it is encountered. Later we will learn about the

STOP statement, which is like END, but is a little less definitive.

Enter LIST as shown

```
>>LIST
```

BASIC should have responded by listing the two-line program previously entered

```
100 Print "HELLO"
110 End
```

Note that the PRINT and END statements have been slightly revised. Only the first character is a capital letter. BASIC likes to make statements a little more readable when possible. It might be useful to mention that although we are typing commands and statements in capital letters, BASIC will recognize commands and statement names (PRINT and END) in lowercase letters. We use capitals to highlight entries.

Incidentally, if you forgot the double quote (") at the end of the line containing PRINT, you may notice that BASIC put one there for you. BASIC always inserts a double quote at the end of a line if it needs one. Sometimes this can save you a few extra key strokes. We will refer to (") as a double quote. This is to distinguish it from the single quote ('), which will always be referred to as an apostrophe. The double quote and apostrophe are separate keys on the keyboard.

SOME OFTEN-USED COMMANDS—LIST, RUN, SCR

In our preceding discussion, LIST produced a listing of the program at your terminal in the order of the line numbers specified. LIST was not preceded by a number and is not part of the program. LIST is one of many commands available in BASIC. Commands are not preceded by a line number and they are executed immediately by BASIC. They allow you to display, edit, and manipulate programs.

Although we have just stated that commands are not preceded by line numbers and, hence, are not part of a program, in later chapters we will find that some commands can be preceded by a line number and can become part of a program. For now, ignore this dual ability of commands.

LIST may also be used to list specific lines or groups of lines. For example, LIST 100 lists only line 100. LIST 100,150 lists all lines from 100 through 150, in our case, just lines 100 and 110. Omitting a line number after the comma causes LIST to list all lines from the specified line to the end of the program. If a line number referenced in the command does not exist, LIST, like many other commands, finds the next closest line number available and proceeds.

Another command is RUN. It is very important. Use it by entering

```
>>RUN
```

The results which appear should be

```
HELLO
***110 End***
```

If you do not get this result, retype the program. The string of characters HELLO is the output of the program. The ***110 End*** indicates that the program has stopped running at line 110. In effect, RUN causes BASIC to begin running, or executing, the program from the lowest numbered line and then to proceed to successively higher numbered lines. Later, we will find that there are ways of controlling the execution sequence.

The program just entered is said to be in a work space. LIST allows us to list whatever we have in the work space. The work space can be cleared by entering the command SCR (scratch). Enter SCR followed by LIST

```
>>SCR
>>LIST
```

This time LIST shows that the work space is scratched or empty. Enter the following in the order shown

```
>>120 PRINT "STRUCTURED BASIC"
>>110 PRINT "IS FUN"
>>130 END
```

Enter the LIST command and you will find that you have

```
110 Print "IS FUN"
120 Print "STRUCTURED BASIC"
130 End
```

Entering RUN produces

```
IS FUN
STRUCTURED BASIC
***130 End***
```

Somehow this does not look like reasonable output. The intent was to produce

```
STRUCTURED BASIC
IS FUN
```

How do we correct the program?

CORRECTING PROGRAM LINES

Obviously, the sequence of program statements in our program is incorrect. A possible way to correct this is to first enter 120

```
>>120
```

Now enter LIST

```
>>LIST
```

This produces

```
110 Print "IS FUN"
130 End
```

We have discovered that entering just the line number deletes the line. Now type

```
>>100 PRINT "STRUCTURED BASIC"
```

followed by LIST

```
>>LIST
```

which produces

```
100 Print "STRUCTURED BASIC"
110 Print "IS FUN"
130 End
```

Now, try running this program to see if you get the desired output. Use RUN.

SAVING YOUR PROGRAM

Suppose we like our little program and decide that we would like to keep it for future reference. The BASIC SAVE command is useful here. Enter

```
>>SAVE "EXAMPLE.BAS"
```

This command causes the work space containing your program to be saved as a file with the name EXAMPLE.BAS on the current disk. As you may recall from your knowledge of CDOS or CROMIX, files have a name and an extension. EXAMPLE is the file name and BAS is the extension, and they are separated by a period. File names may contain numbers and other characters like a dollar sign. Usually it is wise to use just letters and numbers. An extension need not be given but it is usually advisable to do so. In order to place a program on a specific disk, a disk specifier may precede the file name. For example, "B:EXAMPLE.BAS" would be used to save the program on the B-disk or corresponding directory (in CROMIX).

By now the program should be saved. Let's see how this really works. Enter the SCR command, followed by the LIST command. BASIC should respond by showing you there is nothing in your work area. Now enter

```
>>LOAD "EXAMPLE.BAS"
```

When BASIC prompts you for input, enter the LIST command. This should produce a listing of the EXAMPLE program that we just entered. Execute the program by entering the RUN command.

Actually, we did not have to scratch the work area before loading EXAMPLE. LOAD always clears the work area first. Try entering some PRINT statements at random line numbers and then use LOAD again without first using SCR. Use LIST to see what has happened.

Another way of loading a program from disk and executing it is to enter

```
RUN "EXAMPLE.BAS"
```

When a file is specified with the RUN command, RUN first loads the program file and then executes it.

Try RUN and LOAD with the names of program files that you do not have on your disk. For example, RUN "NOWHERE.BAS". BASIC should give you a message stating that the file does not exist.

SAVE and LOAD give us a way of saving and retrieving program files. Another pair of commands, ENTER and a variation of LIST, are also useful for this purpose and will be discussed in Chapters 8 and 13. The discussion in Chapter 13 concerns the use of these commands to merge and combine programs.

MISCELLANEOUS COMMANDS—DIR, DSK

Another handy command that you might need at this stage is DIR. DIR allows you to examine your directory. Some examples are

```
DIR
DIR "*.BAS"
DIR "B:M*.*"
```

Note that this is similar to the CDOS DIR command, but that double quotes are used when specifying files.

DSK is another command that occasionally is useful. It helps us designate a new master drive. It functions like the entry of A: or B: in CDOS. Some examples are

```
DSK "A:"
DSK "C:"
DSK "-A:"
```

In the last example, the presence of the minus sign indicates that the disk in drive A should be ejected. A disk is actually only ejected on 8-inch Persi drives. For other drives, a message is issued to remove the disk.

It is advisable to use the DSK command to "log in" a disk drive if you physically remove and replace floppy disks while still in BASIC. You log in the disk by using DSK to establish the drive, with the new disk as the current drive. Then you use DSK to establish the drive that you want as the current drive for the next portion of your terminal session. This procedure gives BASIC and your operating system a chance to recognize the new disk. It is even more advisable to exit from BASIC if you need to switch floppy disks. The point here is that some care is required to make sure that the operating system and BASIC have the opportunity to adjust to the switch.

RENUMBERING PROGRAM LINES

A command that is useful for organizing programs is the RENUMBER command. It renumbers programs in the work area. This will be particularly useful as we write larger programs. It's probably a good idea to try the command now just so you have it available when you do some experimenting on your own. Let's reload our EXAMPLE program. It looks like this

```
100 Print "STRUCTURED BASIC"
110 Print "IS FUN"
130 End
```

Suppose we add a line 115.

```
>>115 PRINT "AND HANDY"
```

Now we have

```
100 Print "STRUCTURED BASIC"
110 Print "IS FUN"
115 Print "AND HANDY"
130 End
```

By entering

```
>>RENUMBER
>>LIST
```

the work area now shows

```
10 Print "STRUCTURED BASIC"
20 Print "IS FUN"
30 Print "AND HANDY"
40 End
```

RENUMBER, in this form, always renumbers the first line to 10 and increments each following line by 10. We can do more. Enter

```
>>RENUMBER 1000,100
>>LIST
```

to produce

```
1000 Print "STRUCTURED BASIC"
1100 Print "IS FUN"
1200 Print "AND HANDY"
1300 End
```

The two numbers following RENUMBER, separated by a comma, indicate that line numbering should begin at 1000 and be incremented by 100. We can do a lot more with this command, and we will explore some possibilities in a later chapter.

Two other forms of RENUMBER exist that are frequently helpful

```
RENUMBER 4000,10,5000,6000
RENUMBER 1000,50,1100,
```

The first form renumbers lines 5000 through 6000 so that the new lines begin at 4000 and are incremented by 10. In the second form, all lines from 1100 to the end of the program are renumbered so that they start at 1000 and are incremented by 50.

An interesting aspect of the RENUMBER command occurs when we renumber a group of lines so that their new line number causes them to be placed ahead of an other group of lines. The renumbering takes place, but the lines are not actually moved. Hence, using LIST will show the line numbers out of sequence. This use of RENUMBER does not occur frequently, but when it does and you are disturbed by the presence of out-of-sequence line numbers, use the LIST-ENTER procedure discussed in Chapter 8.

PHANTOM LINE NUMBERS

There is a peculiarity in the RENUMBER command and BASIC that is worth mentioning. If a line is removed just by entering its line number, that number will be bypassed if an attempt is made to renumber lines which include that line number. For example, consider the very simple program

```
510 PRINT "HELLO"
520 PRINT "HI"
530 END
```

We can remove line 510 by just entering

```
510
```

Now if we renumber with RENUMBER 500,10 and LIST, and we obtain

```
500 Print "HI"
520 End
```

line 510 is bypassed. It still exists as sort of a phantom line number. This peculiarity does not cause any problems; it just jumbles our line numbers slightly. We can use 510 again because only the RENUMBER command is affected. The problem can be eliminated with another version of LIST in combination with the ENTER command. This topic will be discussed in Chapter 8.

DELETING MANY LINES

While we are on the subject of deletions, BASIC contains a DELETE command. Its three forms are seen in the examples

```
DELETE 400
DELETE 300,900
DELETE 1000,
```

Form one deletes only line 400. The second form deletes lines 300 through 900, and the last form deletes everything from line 1000 to the last line. Naturally, some care should be exercised in entering the second form because omitting the second number will cause deletion to the end of the program. The command DELETE, without any specified line numbers, will result in an error and no lines will be deleted.

Phantom line numbers will be created by the use of DELETE.

MORE MISCELLANEOUS COMMANDS—ATTR, RENAME, ERASE

File protection attributes (read, write, and erase) of disk files will be changed with the ATTR (or ATRIB) command; this command has two spellings. The ATTR command is similar to the CDOS ATTR command. Two examples are

```
ATTR "MYFILE.DAT","E"
ATTR "SOMEFILE.DAT","+W"
```

The protection attribute of MYFILE.DAT is changed to erase-protection only. In the case of SOMEFILE.DAT, the write-protection attribute is added to whatever other attributes are in effect for the file.

Files may be renamed with the RENAME command. This command is similar to the CDOS REN command except that its parameters are reversed from REN. An example is

```
RENAME "NEWMASTR.DAT","OLDMASTR.DAT"
```

The name of the file OLDMASTR.DAT becomes NEWMASTR.DAT.

Files may be erased by the use of the ERASE command. This command is similar to the CDOS ERA command. An example of how it may be used is

```
ERASE "TESTPROG.BAS"
```

This simply removes the file TESTPROG.BAS from the current directory.

EXPLORING ON YOUR OWN

At this point, you might want to explore some of the concepts that we have learned. Try using some of the commands and statements in your own program. Try entering something at line zero (BASIC will object). What is the highest line number you can use? (Answer: 99999.) What happens when you enter 123456 or 10.50 as a line number? What happens when you misspell PRINT or LIST? What does LIST 100,50 do? (Nothing; BASIC will object.)

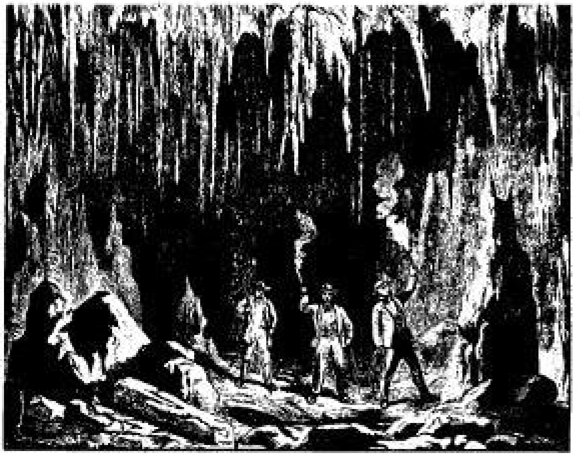


Figure 2.2 Are you sure we came in this way?

Review Questions and Exercises

1. The program line

```
190 PRINT
```

   causes BASIC to skip a line. Is this useful? When?

2. Will the following program print anything when you use RUN? Why not?

```
100 END
110 PRINT "TODAY IS TUESDAY"
120 PRINT "THE TIME IS 10:30 PM"
130 END
```

3. Should you enter programs without leaving room between line numbers to insert new lines? What is a reasonable increment between lines? (Ten is frequently used as a safe increment.)

4. What is a phantom line number?

5. What command saves programs on disk?

6. How do you erase a program from a disk? How do you scratch the work space?

Summary

Line Number
: A number which precedes a statement and is used to number and organize program statements in the work area and program.

Work Area
: An area within BASIC which contains program statements entered from the keyboard or from disk files.
: A program must be in the work area in order to be executed.

Program Statement
: A BASIC instruction which is formed by preceding the statement form with a line number.
A statement represents an operation or instruction which BASIC is to perform when the statement is executed.

Command
: An operation or instruction which BASIC performs immediately when the command is entered.
Similar to program statements except that commands generally are used to manipulate the work area of a program, or to direct BASIC or the operating system to perform some non-programming operation.

Phantom Line Number
: A line number which has been deleted from the work area and is not reused for a new statement.

BYE
: Returns you to CDOS or CROMIX.

PRINT
: Program statement which prints a character string at the terminal.

END
: Program statement which ends execution of a program.

LIST
: Lists the work area.
Several forms exist to selectively list lines.

LOAD
: Places a program file from disk into the work area.
Clears and resets the work area before loading the program.

RUN
: Executes the program in the work area.
Loads and executes programs from disk.
Only loads and executes programs that have been placed on disk with the SAVE command.

SAVE
: Places a program in the work area onto the disk.

SCR
: Scratches, clears, and resets the work area.

DELETE
: Deletes lines of code.
Several forms exist to selectively delete lines.

RENUMBER
: Renumbers program lines in equally spaced intervals.
Several forms exist to renumber lines selectively.

DSK
: Alters the current disk default drive.

DIR
: Lists files in a directory.

RENAME
: Renames a disk file.
Parameters are reversed from the CDOS REN command.

ERASE
: Erases a disk file.

ATTR or ATRIB
: Changes the protection attributes of a file.

Boldface
: Used in this book to indicate that the reader is to enter those program lines and commands at his or her computer terminal.

\>>
: Prompt symbols output by BASIC to indicate that you are to enter statements or commands.

CHAPTER 3
Exploring PRINT and LET

In this chapter, we will cover the capabilities of the PRINT statement in greater depth. We will also introduce elements of the LET statement, which will allow us to perform complex calculations. Numeric and string variables will be introduced, which will permit us to store data for use in other statements.

EXPANDING OUR KNOWLEDGE OF PRINT

We have learned a number of things about BASIC, but the programs we can write with just a simple PRINT are not very useful. However, the capabilities of PRINT have not yet been fully explored.

Clear your work area with the SCR command and enter the following program.

```
>>100 PRINT "HELLO"
>>110 PRINT
>>120 PRINT "How are you?"
>>130 PRINT
>>140 PRINT 100.55
>>150 PRINT "Monday","Tuesday","Wednesday"
>>160 PRINT 10*2
>>170 @ 4,5
>>180 END
```

After you have entered this program, enter the RUN command to execute it. The results should be

```
HELLO

How are you?

100.55
Monday            Tuesday           Wednesday
20
4                 5
```

We note that the PRINT statement at 110 does not have anything following PRINT. This is legal; the effect is to skip a line. The output produced by line 120 is in upper-and lowercase as shown in the string. The string enclosed in quotes is produced exactly as it was entered into the program. Line 140 does not contain a character string. Instead, it contains a number, and the number is printed. Line 150 contains three different strings, which are separated by commas. The effect is to produce the three strings on the same output line with some spacing between them. Line 160 contains 10*2 and produces 20 in the output. The notation 10*2 indicates that 10 is to be multiplied by 2. The result is output by the PRINT statement. Finally, line 170 shows that an @ symbol is an alternate notation for the PRINT statement. This single symbol reduces the number of key strokes needed to enter the often-used PRINT statement. We will continue to use PRINT for clarity.

The notation * denotes the multiplication operator when it appears between numbers. Several other arithmetic operators are available: / for division, - for subtraction, and + for addition. Examples are

```
15*30    45/15    10+90    90-25    400/40    15+5
```

Before pursuing the ability of BASIC to perform computations, let's digress for a bit and consider mistakes that are related to statements.

ERRORS IN FORM—SYNTACTIC ERRORS

What constitutes a mistake? At present, our concern is with what are called syntactic errors, which are caused by not following the specific form of a statement. This is probably a good time to discuss the subject, since our statements are getting complicated by the presence of strings, numbers, and computations. Other errors, such as logic errors, will be touched upon later.

BASIC assumes a specific form for each statement. This form must be adhered to, or BASIC will object with a message that says something like "Error-1 Syntax" when you enter an invalid statement. For example

```
PRIN  100,200
```

will produce such a message because PRINT is misspelled. A similar result is obtained for

```
PRINT "WEIGHT,"SPEED"
```

except the error here is that the string "WEIGHT does not have a double quote at the end. BASIC will allow you to enter these statements but will issue a syntax error message which indicates that the syntax or form of the statement is incorrect. This is a very useful feature of BASIC, because you are warned of such mistakes immediately instead of at some later, and perhaps more critical, time. The permissible forms of a statement are stated in the Structured BASIC Instruction Manual. Consult this manual when you have serious trouble with syntax.

What happens if a syntactic error is not corrected immediately and you try to execute an incorrect statement? BASIC will object by stopping and issuing a message telling at which line it stopped. Let's go through the procedure. Scratch your work area and enter

```
>>500 PRI 100,200
>>510 PRINT "FINISHED"
>>520 END
```

Of course, when you enter line 500, the syntax error message is issued, but ignore it. Enter RUN when you've finished entering this program. The program will stop at line 500 with an error message

```
Error 5 at line 500 - Illegal statement
```

We can list the offending statement by entering

```
>>LIST 500
```

Now enter the correct line

```
>>500 PRINT 100,200
```

and then enter RUN. Everything should work properly now.

COMPUTING WITH THE LET STATEMENT

Let's return to our discussion of computations in connection with the PRINT statement

```
PRINT 10*2
```

Use SCR to scratch the work area and enter

```
>>100 PRINT "INCHES IN THREE FEET:",3*12
>>110 PRINT "CUBIC INCHES IN A CUBIC FOOT",12*12*12
```

Now we have strings and numeric computations in the PRINT statement. These items are separated by commas. Use RUN and observe the output. The quantity 3*12 is printed as 36 and 12*12*12 is printed as 1728. Now we are getting somewhere. We have a real calculator with printed information to tell us what the calculations mean! Let's break away from the PRINT statement and explore calculations further.

While we can perform some interesting calculations with the PRINT command, we can do better. Suppose we are interested in calculating the weight of people in ounces, given their weight in pounds. The following simple program might do for this purpose. You need not enter it.

```
300 PRINT "WEIGHT IN OUNCES",175*16
```

Here the weight is assumed to be 175. Suppose instead that there are a lot of different weights which are to be converted to ounces. Every time we change the weight, line 300 must be retyped. To overcome this problem, let us consider the following program instead

```
200 LET WGT=175
300 PRINT "WEIGHT IN OUNCES",WGT*16
```

Line 200 introduces the LET statement. It says set something called WGT equal to 175. WGT represents the name of a location in which we store the number 175. In line 300 it replaces the number 175. This program will produce exactly the same results as our preceding program. However, if our weight changes, we need to retype only line 200 and use RUN to execute the program.

WGT represents what is termed a variable. Variables are used to store things for computations or for other purposes needed somewhere else in a program. Variables have arbitrarily assigned names with a length of from 1 to 32 characters. Other possibilities are

```
LET WEIGHT=175
LET W=175
LET WT=175
```

and so on. The first character of a name must be alphabetic but the remaining characters may be any combination of alphabetic and numeric characters or apostrophes: WEIGHT, WGT1, WEIGHT'IN'1940, STONE'WEIGHT. Blanks are not allowed in a name. There are some restrictions on names that will be discussed later.

The LET statement is sometimes referred to as an assignment statement since it assigns a value to the variable that appears to the left of the equal sign. Further examples of this statement are

```
LET VOLUME=10*15*20
LET COST=100*300+1000
LET SPEED=400/20
```

Note that an arithmetic expression may appear on the right side of the equal sign. We read, for example, COST=100*300+1000 as assigning 100 times 300, plus 1000, to the variable COST. When this particular LET statement is executed, 31000 will be placed in COST. COST may be used later in some computation or displayed in a PRINT statement. COST will contain 31000 until it is changed elsewhere in the program.

LET has other forms. Scratch your work area with SCR and enter the following program

```
>>100 LET AREA=5*10
>>110 LET VOLUME=AREA*5
>>120 PRINT "Volume and Area:",VOLUME, AREA
>>130 END
```

Line 100 causes 50 (10 times 5) to be saved in AREA. The next line says multiply AREA by 5 and put the result, 250, in VOLUME. Line 120 displays the contents of VOLUME and AREA. Note that each item in PRINT is separated by commas. Try executing this program with RUN.

An often-used LET statement in programs is

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

What does this statement do? A value of 1 is added to the data in COUNT and put back into COUNT. The result is that COUNT has increased by 1. Of course COUNT could be any variable name, RED'AUTOS, BEANS, MARBLES. Another point to note here is that a variable appearing on the left of an assignment may also appear on the right. That is, the computations on the right are carried out first and then placed in the assignment variable on the left, thus changing the value of the assignment variable.

Let's return to variable names. Earlier we mentioned that numeric variable names consist of from 1 to 32 characters. This is true, but it is usually not a good idea to use long names for variables. The reason for this is that if the variable is used frequently, it will have to be entered in the program in a number of places. This can lead to a lot of typing effort and introduces the possibility of typing mistakes. Usually names with 3 to 10 characters are sufficient to describe most variables.

By now you should realize that we are beginning to have the elements of some pretty powerful statements for writing programs and solving problems. Let's expand our knowledge of LET.

USING LET WITH STRING DATA

Let's learn something else about variables. Can we store strings such as "JOHN DOE" or "Portland, Oregon" in variables? The answer is yes, but there are some restrictions. All variable names for variables which may contain alphabetic or non-alphabetic string information must contain a dollar sign as their last character. Examples are CITY$, NAME$, PARTNAME$, WEEKDAY1980$, or CLASS$. Some possible LET assignments are

```
LET CITY$="DALLAS"
LET NAME$="JOHN DOE"
LET DATE$="MAY 15, 1984"
```

These variables are referred to as string variables because they contain strings of alphabetic and non-alphabetic data.

Scratch the work area again and enter

```
>>100 LET CITY$="NEW YORK"
>>110 LET NAME$="SMITH"
>>120 PRINT NAME$," IS FROM ",CITY$
>>130 END
```

Try executing this program with RUN. You should see output something like

```
SMITH          IS FROM          NEW YORK
```

Although this is a very simple program, you should see that there are great possibilities for writing programs which we can adapt to different needs by changing just a few instructions. Variables, both numeric and string, are the keys to this ability.

Like numeric variables, a string variable can be saved into another string variable. Consider the following program

```
100 LET NAME$="ROBIN"
110 LET HOLD$=NAME$
120 PRINT NAME$,HOLD$
130 END
```

It produces as output

```
ROBIN          ROBIN
```

Some other points about string variables must be discussed.

When an assignment of the type

```
LET ADDRESS$="1857"
```

is made, ADDRESS$, although it contains a "number," cannot be used in a computation. In this case, 1857 represents a string, not a number. Such strings cannot be used in arithmetic operations unless they are converted to a numeric representation. The BASIC VAL function, which will be discussed in a later chapter, may be used to perform such a conversion.

WORKING WITH LONGER STRINGS

String variables are restricted to 11 characters of data, unless you tell the computer differently. We'll discuss how you can tell it differently shortly. For now, consider the following program. If you decide to enter it into BASIC, make sure you put the $ at the end of the variable names.

```
100 LET NAME$="FRANKENSTEIN"
110 LET COUNTRY$="NORTHERN IRELAND"
120 PRINT NAME$," IS NOT FROM ",COUNTRY$
130 END
```

Both NAME$ and COUNTRY$ are set to strings which are longer than 11 characters. BASIC will just use the first 11 characters so the result will look like

```
FRANKENSTEI     IS NOT FROM     NORTHERN IR
```

We can remedy this problem with a DIM statement. DIM, for dimension, is a declarative statement that tells BASIC to define variables in a certain way. It also can be used with numeric variables to specify arrays and vectors, but here the interest is in allowing longer strings to be stored in string variables. In our example, suppose we do not expect a country name to exceed 20 characters and a person's name to exceed 15 characters. Adding

```
90 DIM COUNTRY$(19),NAME$(14)
```

to the program tells BASIC to allow 20- and 15-character strings for COUNTRY$ and NAME$. We use 19 instead of 20 and 14 instead of 15 in the DIM. This is because BASIC reserves space beginning with position 0. Hence, when we specify a number like 19, we are asking to reserve space for character positions 0 through 19, which makes 20 characters. Try entering the program with and without line 90 to see the difference.

As an aside for intermediate or advanced users, DIM is an executable statement. If it is not executed, it has no effect.

USING PORTIONS OF STRINGS—SUBSTRINGS

Another useful feature of string variables is that portions of the string contained in the variable may be referenced. Consider the following

```
100 LET NAME$="MARY SMITH"
110 PRINT "FIRST NAME:",NAME$(0,3)
120 PRINT "LAST NAME :",NAME$(5,9)
130 END
```

Executing this program produces

```
FIRST NAME:     MARY
LAST NAME :     SMITH
```

The notations (0,3) and (5,9) following the string name NAME$ tell BASIC to select a range of characters shown in the parentheses. The first character in a string is in position 0, so (0,3) indicates that positions 0 through 3 (MARY) are selected. When the parenthesis notation follows a string variable, the resulting string that is selected is called a substring. Other substring notations exist. For example, a way of selecting a substring that starts in a certain position and continues to the last position is to just reference the starting position. For example, NAME$(5) is the substring reference for positions 5 to 10 (10 is the last position of an 11 character string).

There is more to say about substring references, but for most applications, the substring references discussed are sufficient. We particularly recommend using both the beginning and ending position to reference a substring. We will return to substring references in Chapter 10.

MISCELLANEOUS REMARKS ABOUT LET

There are several important facts to know about the LET statement. First, it is really not necessary to put LET in the statement. BASIC is able to determine when a LET or assignment statement is used. Hence, the following are all legal LET statements:

```
INTEREST=0.06*MONEY
TOTAL=TOTAL+PREVIOUS'AMT
AREA=300*5+60*4+20*3/2
NAME$="WILSON, WOODROW"
CITY$=CAPITAL$
```

We will continue to use LET, but it is not necessary.

Second, it is not possible to assign numeric values to string variables and strings to numeric variables. The following are illegal statements

```
LET CITY="BOSTON"
LET NUMBER="THREE-HUNDRED"
LET ADDRESS$=1450
```

Third, a LET statement is not the only way of assigning data to a variable. You may have accidentally discovered this fact for yourself if you mistyped a variable name. Consider the simple program

```
100 PRINT TOTAL
120 PRINT CITY$
130 END
```

Executing this program will produce

```
0

***130 End***
```

TOTAL has a value of 0 and CITY$ contains nothing. The empty line following the output of 0 represents CITY$. We have discovered that BASIC initializes numeric variables to 0 and string variables to an empty string, which is sometimes referred to as a null string.

RESERVED NAMES

Although there is a lot of freedom in composing variable names, there are some exceptions. A number of reserved names exist in BASIC. These are names used to define statements like PRINT or LET. The following will cause BASIC to object

```
        LET PRINT=100.00
        LET BYE=34*HEIGHT
        LET LET=LET+1
```

In general, it is best not to use command or statement names for variable names. Function names are not usable as variable names either; we will discuss functions more fully in a later chapter.

PRINT FIELD WIDTH

You may have noticed in some of the PRINT examples that the output contains a number of apparently random spaces between items. For example

```
        PRINT "RESULTS:",500,"POUNDS"
```

produces

```
        RESULTS              500                 POUNDS
```

The reason for this is that the PRINT statement places each item in a field of 20 characters in width. If we put a gauge over the column's output, you will see this more clearly

```
                 column 10
                 |
                 v
                 1         2         3         4         5
        12345678901234567890123456789012345678901234567890
        RESULTS              500                 POUNDS
```

We might inquire as to what happens if we place more items in a PRINT statement than can fit in the width of the terminal output screen. For example

```
PRINT 1,2,3,4,5
```

Write a one-line program with this statement and see what happens. You will see that the numbers fold over onto a second line.

Here is a small program that illustrates a possible use for the orderly placement of data in columns

```
100 PRINT "COMPOSERS","MOVIES"
110 PRINT "---------","------"
120 PRINT "BACH","STAR WARS"
130 PRINT "BRAHMS","JAWS"
140 PRINT "WAGNER","MY FAIR LADY"
150 PRINT " ","FIVE EASY PIECES"
160 PRINT " ","CASABLANCA"
170 END
```

This program produces the output

Composers	Movies
BACH	STAR WARS
BRAHMS	JAWS
WAGNER	MY FAIR LADY
	FIVE EASY PIECES
	CASABLANCA

CONTROL OF SPACES AND CARRIAGE RETURNS WITH PRINT

In some applications it may be necessary to control output spacing. This may be accomplished by separating items with a semicolon.

```
PRINT "RESULTS:";500;"POUNDS"
```

produces

```
RESULTS:500POUNDS
```

When semicolons appear, all blanks between items are removed. Of course, some blanks in the strings might make this output more readable.

```
PRINT "RESULTS: ";500;" POUNDS"
```

produces

```
RESULTS: 500 POUNDS
```

Semicolons and commas may be used in any combination in a PRINT statement. For example

```
PRINT 5550,"and",300;" are not the same"
```

produces

```
5550                and                 300 are not the same
```

Commas indicate that the next print item should begin at the left-most position of the next 20-character field. Semicolons remove spaces between items that are output to the terminal.

The semicolon has another use in the PRINT statement which can be quite handy. Whenever a PRINT statement is executed, it always performs a return or skip to the next line when the last item is printed. However, if a semicolon is placed after the last item, the return does not occur. Consider

```
100 PRINT "HELLO. ";
110 PRINT "STRUCTURED BASIC IS FUN"
120 PRINT "AND USEFUL."
130 PRINT
140 PRINT 1,2;
150 PRINT 3;" TESTING"
160 END
```

This program produces as output

```
HELLO. STRUCTURED BASIC IS FUN
AND USEFUL.

1                   2                   3 TESTING
```

The semicolon at the end of line 100 caused suppression of the return. The same is true of line 140. Note the blank before TESTING appears because it is part of the string " TESTING". This relatively simple feature of BASIC will prove to be very useful in many application programs, so keep it in mind.

Other ways of controlling the spacing of output from the PRINT statement are with the use of the SPC and TAB functions. We will discuss a number of functions that can be used in BASIC later, but, for now, think of these two functions as special indicators to the PRINT statement that something special is to occur when they are used.

SPC simply inserts a specified number of spaces in the output. For example,

```
PRINT SPC(50);"PAGE 44"
```

causes 50 spaces to be output before printing "PAGE 44". The number of spaces needed is always enclosed in parentheses.

TAB allows us to position the next item output by the PRINT statement in a specific column on the line. For example

```
PRINT "REPORT 6";TAB(41);"VERSION 4"
```

prints "REPORT 6", skips to column 41, and prints "VERSION 4". Like many position-dependent features of BASIC, the first column is referenced as column 0. Hence, TAB(41) causes the string "VERSION 4" to begin in the forty-second column.

When using either TAB or SPC, we separate items in the PRINT list with semicolons to ensure that the 20 column subfield definitions discussed previously do not interfere with the placement of items on a line.

LITERALS AND EMBEDDED QUOTES

A point that is worth mentioning is that numbers and strings, such as 22.44 and "HELLO", which have been used as part of our programs, are often termed literals; they are sometimes referred to as numeric literals and string literals, to make the distinction finer.

Another point is that when you need to place a double quote inside a string, you must place two of them side by side, as in

```
"QUOTATION:""IF,"" HE SAID."
```

This is printed upon output from a PRINT statement as

```
QUOTATION:"IF," HE SAID.
```

Review Questions and Exercises

1. Which statement in the following program is incorrect?

```
100 PRINT 34.0,22.5
110 PRINT ,500
120 END
```

   What is the easiest way to correct line 110?

2. Why will using PRINT as a variable name result in an error in a LET statement?

3. Which of the two statements is preferable?

```
PRINT LASTNAME$,FIRSTNAME$
PRINT LASTNAME$;FIRSTNAME$
```

4. Why are these LET statements wrong?

```
LET ABC$=100.0
LET DATE="SEPTEMBER 25, 2001"
LET TIME$='1300 HOURS'
```

5. Using

```
LET ALPHA$="ABCDEFGHIJK"
```

   causes only A through J to be found in ALPHA$. Why?

6. The names hours and HOURS are the same to BASIC. What type of variable does HOURS represent? (Numeric or string?)

7. What is the use of a semicolon in a PRINT statement? What does it mean when the semicolon is used at the end of a PRINT statement?

8. What is the difference between a literal and a numeric constant? What is the difference between a literal and a string constant?

9. Why does the statement

```
520 DIM STATE$(19)
```

   allow STATE$ to contain as many as 20 characters?

10. If the variable VALUE has not been set by any statement before it is used, why will it contain the value zero?

11. If TOTAL has a value of 15.0 prior to BASIC encountering

```
200 LET TOTAL=TOTAL+18.0
```

why does TOTAL have the value 33.0 after executing this statement?

12. What is the meaning of NAME$(10,15)? How many characters does (10,15) refer to? Is the substring NAME$(0,0) valid? What does it generally refer to, assuming NAME$ contains the first name of someone?

13. In the following program, why does BIRD$ have a null value?

```
100 PRINT BIRD$;" is the name of a bird."
110 END
```

What will be printed by line 100?

14. What do you suppose was meant by the following valid statement?

```
900 PRINT "RED","BROWN""GREEN"
```

Is something missing? Why is the statement still valid?

Summary

PRINT

Items in a PRINT statement may be numbers, strings, arithmetic expressions, numeric variables, string variables, or substrings.
Items following the PRINT are separated by commas or semicolons.
Commas cause output to be left-justified in 20-column fields.
Semicolons remove spaces between output items.

If no item appears after PRINT, an empty line appears in the output; that is, a line skip occurs.
A semicolon at the end of a list of items suppresses the carriage return after the items are printed.

The @ symbol is an alternate notation for PRINT in the PRINT statement.

LET

Performs arithmetic computations and assigns or places the results in a variable.
Assigns or places strings into string variables.
The arithmetic expression on the right of the equal sign may contain complex arithmetic operations involving constants and numeric variables.

Variables--Numeric

Contain and store a numeric value which may be the result of a numeric computation with a LET.
Numeric variable names begin with an alphabetic character and may contain as many as 32 alphabetic characters, numeric characters, and apostrophes.
Variables which are not set by the program are initialized to 0.

Variables--String

Contain and store a string of characters which may be assigned to a variable with a LET.
String variable names are formed in the same way as numeric variable names but they must end with a dollar sign.
A string up to 11 characters long may be kept in a string variable without being redefined in a DIM.
The first position of a string is position 0.
String variables which are not set by the program are initialized to a null string.

DIM

Allows the length of the string stored by a string variable to be defined.
DIM is not effective unless it is executed.
All strings begin at position 0.

Substrings
    Referenced by appending
    (start-position, end-position) notation to a
    string variable name.

Reserved Name
    Any name which is used as a BASIC statement
    or command name (and function) cannot be used
    as a numeric variable name.

Syntax Errors
    Caused by improperly formed statements or
    commands.
    A syntactically invalid statement cannot be
    executed.

Literals
    A name for numeric constants and strings of
    characters surrounded by double quotes.

TAB
    A special function for use with PRINT that
    positions the next item to be printed at a
    specific column.
    PRINT columns are numbered from zero.

SPC
    A special function for use with PRINT that
    produces a specified number of spaces in the
    output.

CHAPTER 4

Data INPUT and Numbers

We have come a fair distance in the initial chapters. We have learned a number of commands and two frequently used statements, PRINT and LET. So far, nothing has been said about getting data into the computer by any means other than through a LET statement. However, this can be a tedious way of entering data, because it requires changing program statements each time a new data value is needed. In this chapter, we will learn of some ways that make entering data easier. In addition, we will discuss the representation of numeric data.

INPUT **WITH NUMERIC** DATA

One of the simplest ways of entering data into a BASIC program is with the INPUT statement. INPUT allows us to enter data when a program is executing. Let's consider a simple example to see how it works. Scratch your work area and enter the program

```
>>100 PRINT "ENTER SIDE OF SQUARE"
>>110 INPUT SIDE
>>120 PRINT "AREA OF SQUARE:  ";SIDE*SIDE
>>130 END
```

This program computes the area of a square, given the length of a side. Line 110 is actually a request to enter a value which will be placed in the variable SIDE. SIDE is multiplied by SIDE in line

120 to give the resulting area. Enter RUN to see what happens. You should get

```
ENTER SIDE OF SQUARE
?
```

The question mark is printed by the INPUT statement and means that the program wants you to respond with a number. Enter the number 5, as in

```
ENTER SIDE OF SQUARE
? 5
```

You should see the following printed after you enter 5 and hit the carriage return

```
AREA OF SQUARE: 25
```

What happens if we enter RUN again, but respond with a carriage return without first typing in a number? Until you enter a number, INPUT will continue to prompt you with a question mark. Try it.

What happens if you enter something that is not a number, like ABC? BASIC will object and issue the message

```
Error 204 at line 110 -- INPUT
```

Try it. The way to recover from this error is to enter RUN again and input a proper number.

We can get fancier with INPUT. Scratch the work area and enter

```
>>100 PRINT "ENTER THE TWO SIDES OF A RECTANGLE"
>>110 INPUT SIDEA,SIDEB
>>120 PRINT "AREA OF RECTANGLE:  ",SIDEA*SIDEB
>>130 END
```

This program computes the area of a rectangle when the lengths of two adjacent sides are given. If you enter RUN, you should get

```
ENTER THE TWO SIDES OF A RECTANGLE
?
```

The question mark again is the prompt from INPUT. If you look at INPUT on line 110, it wants two values, one for SIDEA and the other for SIDEB. To enter the values, we separate them by commas as in

```
ENTER THE TWO SIDES OF A RECTANGLE
? 5,10
```

which produces

```
AREA OF RECTANGLE: 50
```

What happens if only one number is entered? BASIC will prompt you for the next number with a double question mark as in

```
ENTER THE TWO SIDES OF A RECTANGLE
? 5
?? 10
AREA OF RECTANGLE: 50
```

Suppose three numbers are entered in response to an INPUT request that only requires two numbers. What happens? BASIC produces error message 204, which we saw above. Incidentally, if you are having trouble using INPUT at this point, hit the escape key on your terminal. Hitting the escape key is a useful way to terminate a program that seems to be hung-up.

INPUT WITH STRING DATA

Numeric values are not the only data INPUT accepts. Let's try string data. Scratch the work area and enter

```
>>100 PRINT "WHAT IS YOUR NAME"
>>110 INPUT NAME$
>>120 PRINT "HELLO, ";NAME$
>>130 END
```

Enter RUN and you will get the following request:

```
WHAT IS YOUR NAME
?
```

Try responding with the name MR. SPOCK and you should get the sequence

```
WHAT IS YOUR NAME
? MR. SPOCK
HELLO, MR. SPOCK
```

Unlike our previous experience with strings, INPUT did not require quotes. This is not always true, as we shall shortly discover. What happens if we use RUN again and enter the name MR. STEVENSON?

```
WHAT IS YOUR NAME
? MR. STEVENSON
HELLO, MR. STEVENS
```

Our name has been shortened; the input MR. STEVENSON is output as MR. STEVENS. As we noted before, NAME$ may only contain 11 characters if it is not declared to have a greater length with a DIM statement. INPUT simply truncates the string after the eleventh character.

Suppose that instead of reading one string we try to read two. Clear your work area and enter

```
>>100 PRINT "ENTER NAME AND THE MONTH YOU WERE BORN"
>>110 INPUT NAME$,MONTH$
>>120 PRINT NAME$;" WAS BORN IN ";MONTH$
>>130 END
```

The INPUT statement requires data for two different variables: NAME$ and MONTH$. If we enter, for example

```
JOHNSON,MAY
```

BASIC has no way of deciding which part of this string belongs in NAME$ and which part belongs in MONTH$. The comma is not sufficient to divide the two since INPUT reads any character as a string. The solution is that we need to enter each separate string in quotes after having entered RUN, as in the following instance. Try running the program to produce the following sequence

```
ENTER NAME AND THE MONTH YOU WERE BORN
?  "JOHNSON","MAY"
JOHNSON WAS BORN IN MAY
```

INPUT WITH STRINGS AND NUMBERS

Whenever a string variable and any other variable is requested as input through an INPUT statement, it is always necessary to enclose the strings in quotes. An appropriate response to the program segment

```
150 PRINT "ENTER NAME AND AGE"
160 INPUT NAME$,AGE
```

for a person with the name SMITH and who is age 25 would be "SMITH",25.

DRESSING UP THE INPUT STATEMENT

The question mark output from INPUT can be eliminated by placing a string immediately before the first item in the INPUT list. Instead of a question mark prompt, the string is printed in its place. This device can be used to produce a more acceptable looking prompt for users of your computer programs. Clear your work area again and enter

```
>>100 INPUT "ENTER YOUR NAME: ",NAME$
>>110 PRINT "YOUR NAME IS ";NAME$
>>120 END
```

Using RUN and entering the name JONES produces the following sequence

```
ENTER YOUR NAME:  JONES
YOUR NAME IS JONES
```

The string "ENTER YOUR NAME: " appearing in line 100 is substituted for the question mark that would otherwise appear. This device is used very often in interactive programs, i.e., programs which generally use yes, no, and names for responses.

INPUT is used most frequently for obtaining interactive responses and for obtaining small amounts of data. Another statement, READ, is useful for working with larger amounts of data.

ANOTHER DATA INPUT METHOD—READ, DATA, AND RESTORE

READ is similar to INPUT except that it expects data from DATA statements rather than from the terminal.

As an illustration, clear your work area and enter

```
>>100 DATA 200,50,20
>>110 READ A,B,C
>>120 PRINT "SUM OF THE NUMBERS:  ";A+B+C
>>130 END
```

Entering RUN produces

```
SUM OF THE NUMBERS: 270
```

The values 200, 50, and 20 in the DATA statement are read into the numeric variables A, B, and C by the READ statement. This program produces the same results

```
100 DATA 200,50
110 DATA 20
120 READ A
130 READ B,C
140 PRINT "SUM OF THE NUMBERS: ";A+B+C
150 END
```

The READ at line 120 reads the value 200 into A. The READ at line 130 reads the values 50 and 20 into B and C, respectively. We see that READ just reads data from the various DATA statements in the order in which the DATA statements are entered. DATA statements need not occur in any particular part of the program. For example, they may occur after the first READ. Usually, for readability and organization, they are placed together at the beginning or end of the program.

DATA statements may contain string data as well. Strings must be enclosed in double quotes. Data items may also be arithmetic expressions, such as 3+10. Some examples of valid DATA statements are

```
DATA 200,"ARBOR HOSPITAL"
DATA 400,200/30+100,"TAXABLE CAPITAL"
DATA "JAN","FEB","MAR","APR"
```

Unlike the INPUT statement, which prompts us with a double question mark when a data value has not been entered, the READ statement produces an error message if an attempt is made to read more values from DATA statements than exist. Of course, READ will balk at reading a string into a numeric variable and vice versa.

Sometimes it is necessary for data values to be read several times in a program. The RESTORE statement allows us to perform this task. Consider

```
100 DATA 1800,2000,200
110 READ A,B,C
120 RESTORE
130 READ D,E,F
140 PRINT "SUM: ";A+B+C
150 PRINT "DIFFERENCES: ";D-E,E-F
160 END
```

When executed this program produces

```
SUM: 4000
DIFFERENCES: -200            1800
```

When the RESTORE at line 120 is executed, it tells BASIC to start the next READ at the first DATA statement in the program. The next READ, at line 130, re-reads the data into D, E, and F. These are the same values read into A, B, and C. Although this example is so simple that there seems to be little value in re-reading the data because the result on line 150 could be obtained as A-B,B-C, it does illustrate the capability of RESTORE. RESTORE can be used specifically to reset the pointer to the next DATA statement to be read by entering RESTORE, followed by a line number, as in

```
RESTORE 200
```

This statement would cause the next READ to take place at the first DATA statement on or following line 200.

NULL CHARACTERS

An important aspect of string variables is their relationship to null characters. A null character is simply the absence of a character. A null character cannot be displayed on a printer or at your terminal, but it does occupy space in a string variable. Their normal function is to signal the end of a string of characters. As an illustration, consider the following statements:

```
100 DIM STATE$(14)
110 LET STATE$="TEXAS"
```

STATE$ is a 15- (remember the zero position) character string; "TEXAS" is placed in its first five positions. The remaining positions are padded with null characters. Denoting a null character by a ^, the string is kept internal to BASIC as:

```
TEXAS^^^^^^^^^^
```

Trailing null characters are important to BASIC. It uses them as a means of marking an end of string. You normally do not have to worry about them, but it is useful to know about them.

When using PRINT to display a string variable, only the portion of the string up to the trailing null characters are output. When using INPUT or READ, if the input string is smaller than the dimension of the string variable, then the string variable is padded with trailing null characters when the input string is stored in the variable. This is similar to what happened in the LET example.

An instance when the trailing null characters may cause you some concern occurs when a semicolon is used to separate two string variables, as in

```
100 CITY$="SAN DIEGO,"
110 STATE$="CALIFORNIA"
120 PRINT CITY$;STATE$
130 END
```

Both CITY$ and STATE$ may contain up to 11 characters, by default. Line 100 puts a 10-character string in CITY$. CITY$ will contain 1 trailing null character. You might expect line 110 to produce

```
SAN DIEGO, CALIFORNIA
```

However, the single null character does not print, and it does not occupy space in the output. Instead, we get

```
SAN DIEGO,CALIFORNIA
```

with a space missing between the city and state. Of course, we could remedy this by inserting a blank literal in the PRINT statement. However, the point we are making is that such shifts may be disturbing in some applications where items are expected in particular columns of the output line. In the next chapter, we will find that the PRINT USING statement gives us a way of making sure that items are positioned where we want them.

A REMARKABLE STATEMENT—REM

It is already possible to write some fairly large programs with just the few statements that we have discussed. When programs begin to go beyond several statements, it is usually a good idea to be able to put some descriptive information in them that will enable us to understand what is happening if we have to come back to the program at a later time. BASIC provides us this facility through the REM, or remark, statement. A REM statement does absolutely nothing as far as BASIC is concerned. BASIC ignores these statements during execution. Examine the program

```
100 REM DATA VALUE IS PI
110 DATA 3.1415
120 REM THIS IS THE BEGINNING OF THE PROGRAM
130 READ PI
140 PRINT "THE AREA OF A CIRCLE ";
150 PRINT "WITH A 10 INCH RADIUS IS: ";
160 PRINT PI*10*10
170 END
```

Lines 100 and 120 contain REM statements which describe something to us, not to BASIC, about the program which are ignored by BASIC during execution.

REM statements require storage space in your work area and in program files stored on disk. In fact, they can take up a considerable portion of your program area if used excessively. It is a good idea to use them, but used them sparingly. A reasonable rule of thumb might be that 1 out of every 10 to 20 statements can be a REM. The topic of program size and how to control it will be covered later. The discussion of the DELREM command in a later chapter illustrates a way to conveniently delete large numbers of REM statements.

SOME MORE ABOUT NUMBERS

Until now, our examples involving numbers have, for the most part, shown nice whole numbers and no decimals. As you may have discovered on your own, BASIC is perfectly capable of working with other types of numbers as well. Let's take a look at how we represent some numbers in BASIC.

Numbers are generally written in a familiar notation, with some restrictions. Some valid numbers are

```
   100      200.55     30.45E+4     -75.88      65700.00
 -800.0      90.44      .00077       5.5E8        -0.5E-4
```

Some invalid numbers are

```
   1,000.00    one      zero       --6.5     III
    40 x 10    1 1/4    +20.4
```

The reasons why these examples are either valid or invalid will be discussed next.

Numbers may be coded in several different ways: integer, decimal, or E-notation.

Integers are numbers that are expressed without a decimal point, such as 1984, 400, 252, or 1000. Typically such numbers represent counts or amounts.

Decimal numbers are written with a decimal point and some decimal portion, such as 3.1415, 188.45, or .0045. Typically, such numbers represent measurements, scores, percentages, or monetary amounts.

Numbers may be expressed in E-notation when they represent very large or very small quantities. Numbers in this notation are sometimes referred to as being in "scientific notation" or "floating point" notation. They are formed by appending to a number the letter E, which represents "power of 10," and the amount 10 is to be raised by. The result represents the number raised to the power of 10 specified. For example, the number 4.5E6 represents 4.5 times 10 raised to a power of 6. That is, 4.5E6 represents 4.5 x 1000000 or 4500000. A sign may appear immediately after the E to indicate whether the power is positive or negative. The absence of a sign is interpreted as indicating a positive power. The power must not be more than two digits. Other examples of numbers in E-notation are 0.055E4, 18.71E-15, and 3.4578E+2.

Numbers may be signed, that is, expressed as positive or negative. A minus sign, - , prefixing a number indicates a negative number. Some examples of negative numbers are -45.28, -190, and -34.33E10. Positive numbers are indicated by an absence of a minus sign, as in 345, 1000.55, and 0.05.

Numbers may not contain commas to denote groups of 1000 as in 4,000,000. Such a number must be entered as 4000000 or 4E6.

Another type of number that can used in BASIC is a hexadecimal number. These numbers have special uses and will be discussed later in this chapter.

When using numbers with computers, it is desirable to understand how large numbers can be and how many digits can be represented accurately within the computer or language. The largest and smallest non-zero numbers representable, without regard to sign, are 1E+62 and 1E-65, respectively. This is an extremely wide range. If we try to work outside of this range, BASIC will complain. Numbers may contain 14 significant digits. If you enter more than 14 digits, BASIC will simply truncate the number.

HEXADECIMAL NUMBERS

Another type of number that may be represented in BASIC is a hexadecimal number. A hexadecimal is one which uses base 16 instead of base 10, which is used in the more common decimal system. The following shows the correspondence between base 10 numbers and base 16 numbers.

Decimal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hexadecimal	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

For example, the hexadecimal number D corresponds to the decimal number 13. These numbers are used in applications which deal more directly with the internal computer representation of addresses and (ASCII) characters. In BASIC, a hexadecimal number is denoted by the number surrounded by percent signs. For example, %20% is the hexadecimal number 20, or 32 in base 10 notation. We will not have much need for hexadecimal numbers in our programs, and they will not be discussed further.

LITERAL AND VARIABLE STORAGE

As we have seen, BASIC allows us to use integer, decimal, and scientific forms of numbers for our numeric data. These literals or constants are kept in the storage area of a program, and they occupy space in storage, or memory, as this type of storage is often called. For reasons of efficiency, BASIC keeps or stores numbers internally in two forms. The two forms are the integer form and the floating form. These forms allow BASIC certain advantages in storage and computational speed. (Actually, a third form is used for hexadecimal numbers. Hexadecimal numbers are stored exactly in the form they are written in the program.)

The integer form of numbers is familiar to us from the previous discussion. Actually, the floating form is familiar to us as well. Numbers expressed in floating, or floating point, form are similar to the numbers expressed in the scientific notation previously discussed. When we enter a simple decimal number, BASIC converts it internally to its floating notation, i.e., to a form that contains a power of 10. Numbers in scientific notation are already in floating notation. BASIC finds it efficient to convert any integer number we enter that lies outside the range from -10000 to +10000 to a floating point number as well. Hence, a number like 432 is represented as an integer internally, but 12345 is represented as a floating number internally.

In addition to representing literals as floating or integer, BASIC allows us to specify the type of data a variable may store. Later, we will learn how to specify whether a variable is to contain floating or integer data. For now, we will state that all the variables will store numbers in their floating forms. Two forms of floating variables exist, a short form and long form. The short form only permits the storage of about 6 significant digits whereas the long form allows up to 14 digits of accuracy. The ability to specify what type of number is represented in a variable allows us to control how much storage is used by a program. This has an impact on the size of the programs we can write. All BASIC variables have the long floating storage attribute as a default. We will learn how to control these storage attributes in a later chapter.

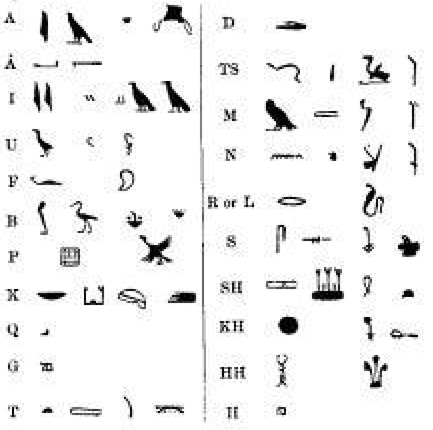


Figure 4.1 Some of these won't work as input data!

Review Questions and Exercises

1. Does the following statement expect a numeric or string value?

```
800 INPUT TIME
```

2. Consider the statement

```
500 INPUT "Enter three ages - ";A,B,C
```

How do you enter on a single input line the three values requested?

3. When are quote marks required for a response to an INPUT statement that involves entering string data?

4. What statement does BASIC use to read data from a DATA statement? Must DATA statements appear in any special place in a program?

5. What BASIC statement permits you to read a DATA statement again?

6. A programmer entered the following DATA statements and expected to define 12 three-character strings identifying each month of the year. Instead, only 11 strings are defined. What is wrong? Why doesn't BASIC generate a syntax error message?

```
320 DATA "JAN","FEB","MAR","APR","MAY""JUN"
330 DATA "JUL","AUG","SEP","OCT","NOV","DEC"
```

7. What is a null character? Can a null character be printed? How much space does a null character occupy in a print line?

8. Why does the following statement result in a syntax error?

```
200 PRINT +34.44
```

9. What decimal number does 4.5E+1 represent? How about 3.3E-1 and 123.45E-1?

Summary

INPUT
Allows the entry of data from the terminal.
Data are entered in response to a question mark prompt.
Numeric and string data may be input.
A request for a single string data item does not require quotes around the string.
Several numeric and string variables may be entered if string data are enclosed in double quotes.
A string, placed as the first item in the INPUT statement, may be substituted for the question mark normally produced by INPUT.

DATA
Used in specifying data items that are to be read by the READ statement.
Strings must be enclosed in quotes.
Data items may be arithmetic expressions.
DATA statements are read in the order they are found in the program.

RESTORE
Sets the line number at which the next READ statement begins reading from the DATA statements.

REM
Makes it possible to place descriptive remarks within the program that can be used as program documentation.
REM statements require storage space in your program.

READ
Reads data from DATA statements placed throughout the program.
Any combination of numeric and string data may be read.

Numbers (Constants, Literals)
Integer, decimal, and scientific notation are allowed.
Numbers may range in magnitude from 1E-65 to 1E+62.
Up to 14 significant digits may be used.
Numbers are regarded as integer, floating, or hexadecimal by BASIC and each type occupies a different amount of storage.

Floating and Integer Variable Storage
Numeric variables may have storage attributes so that they occupy different amounts of storage.

CHAPTER 5 Functions and PRINT Formats

In this chapter, we will learn more about LET. We will discuss how arithmetic expressions are evaluated, and you will be introduced to the many useful arithmetic and string functions that are available. An important statement, PRINT USING, which allows us to more accurately specify the printed output formats of data, will be introduced.

INCREASING OUR UNDERSTANDING OF LET

Let us return to the arithmetic LET statement that was introduced in a Chapter 3. LET is capable of performing some fairly complex arithmetic and logical calculations. First, let us consider arithmetic calculations.

We stated earlier that arithmetic operations included addition, subtraction, multiplication, and division. Other possible operations include negation and exponentiation. Negation simply refers to turning a result into its negative value. For example

```
LET ABC=-400.55
LET VALUE=-MONEY
```

result in the storage of -400.55 in ABC and the negative value of MONEY in VALUE. Placing a positive sign in front of a value or expression is illegal, as in LET A=+5. Exponentiation refers to the process of raising a number to a power. This is the method often used to express the square of a number. Exponentiation is represented by the symbol ** or ^. The examples

```
LET AREA=10**2
LET VOLUME=SIDE^3
```

result in the square of 10 placed in AREA and the cube of SIDE placed in VOLUME.

As suggested in our earlier discussion of LET, the arithmetic expression can be fairly complex. For example

```
LET TOTAL=LOAN*.08+LOAN+DEBT/12-NUMBER'SOLD*COST
```

The complexity of such expressions raises the question of in what order the operators are applied in the expression. Consider

```
LET A=5*3+1
```

Does this result in 1 being added to 5 times 3 or 3 being added to 1 and the result multiplied by 5? Such expressions are evaluated from left to right, with multiplication and division being performed before addition and subtraction. The result placed in A in our example is 16 even if the statement is written

```
LET A=1+5*3
```

Multiplication and division are performed in the order they occur when they are found in successive operations. The same is true for addition and subtraction. Exponentiation is performed before multiplication and division. Negation is performed before exponentiation; so -3**2 is 9, and not -9.

The manner in which expressions are evaluated is fairly close to what one might normally expect. However, when you are not certain of what might happen or when you want to clarify an expression, parentheses are allowed. The statement

```
LET LENGTH=(1+SIDEA)*(SIDEB/20.5)
```

is clear as to how the computation is to be performed. Remember, when using parentheses, that for every left parenthesis, (, there must be a right parenthesis,). Left and right parentheses always surround a collection of computations that are considered to be together.

At the beginning of this section, we mentioned that LET is capable of performing logical calculations. This topic is probably most suitable for the experienced programmer, and will be touched upon only lightly here. More will be said later about logic in connection with the IF statement. For the experienced programmer, an example of a logical calculation is

```
LET ANSWER=4>8=0
```

The result in ANSWER is 1. Logical expressions always result in a value of 1 or 0. The Structured BASIC Instruction Manual contains additional details on logical calculations.

Another interesting topic in connection with LET is the use of mixed mode arithmetic. We will introduce this topic by asking: What is the result of the following statement?

```
LET Z=4/3
```

If you said 1.333, you are wrong. The result placed in Z is 1.0, assuming Z is a floating variable. The reason for this result is that 4 and 3 are both integers, and BASIC evaluates the expression using integer arithmetic. A division of two integers results in an integer; any decimal portion is removed or truncated. The resulting integer is converted to a floating value when it is placed in Z. We can obtain the desired fractional representation by using either 4 or 3, or both, as floating literals as in

```
LET Z=4.0/3.0
```

When an expression consists of integer or floating literals and variables, the arithmetic performed is referred to as mixed mode arithmetic. The result may not be what we expect, but for most applications it does not cause many problems. As a caution, however, until we return to this topic, it is best to specify decimal constants in expressions which require division by constants. We will return to mixed mode arithmetic again in a later chapter when the SHORT, LONG, and INTEGER statements have been introduced.

Figure 5.1 The detail is getting mighty fine.

ARITHMETIC FUNCTIONS

Another aspect of arithmetic expressions is that there are special functions included with BASIC that permit the calculation of many mathematical functions, e.g., the sine and cosine functions. Even if you are not interested in functions such as the sine and cosine, pay attention here because some of the functions have application to commercial programs as well.

Functions have names. Following the name is an item which is enclosed in parentheses that is called the argument of the function. Some functions have multiple arguments, which are separated from each other by commas. An example of the more common function with a single argument is SIN(ANGLE). This notation indicates that the sine of the argument ANGLE is needed. In a LET statement, we might have

```
LET SIDE=100*SIN(ANGLE)
```

Arguments can contain complex arithmetic expressions as in

```
LET SIDE=100*SIN((ANGLE1+ANGLE2/2.0)+30.0)
```

The following is a list and brief description of the arithmetic functions in BASIC (X represents a number, arithmetic expression, or variable):

Function	Meaning
ABS(X)	Absolute value of X
ATN(X)	Arctangent of X
COS(X)	Cosine of X
EXP(X)	Mathematical constant e raised to X
FRA(X)	Fractional part of X
INT(X)	Integer, whole, portion of X
IRN(X)	Integer random number from 0 to 32767
LOG(X)	Logarithm of X in base e
MAX(X1,...)	Maximum value of X1, X2, etc.
MIN(X1,...)	Minimum value of X1, X2, etc.
RND(X)	Random number from 0.0 to 1.0
SIN(X)	Sine of X
SGN(X)	One with sign of X, i.e., -1 or +1
SQR(X)	Square root of X
TAN(X)	Tangent of X

FRA and INT are very commonly used functions and have numerous uses in commercial applications. The following sample program illustrates their use.

```
100 MONEY=100.50
110 DOLLARS=INT(MONEY)
120 CENTS=FRA(MONEY)
130 PRINT "$"; DOLLARS;" AND ";CENTS*100;" PENNIES"
140 END
```

This program produces as output

```
$100 AND 50 PENNIES
```

When BASIC reaches line 130, DOLLARS contains 100 and CENTS contains 0.50. To get the proper number of pennies for line 130, we multiply CENTS by 100.

An experienced programmer might wonder why a modulo function is not included in the above list of functions. BASIC does not contain this function. For the interested reader, the concept of a modulo operation is perhaps best explained by an illustration. The hour number on most household clocks is the number of hours from midnight modulo 12. That is, the remainder of hours since the clock was last at 12 is the hour number of the day modulo 12. Minutes are taken modulo 60 and so on. A way of computing the modulo of a number, say, modulo 12, can be accomplished using the INT function as in

```
LET CLOCKTIME=DAYHOUR-12*INT(DAYHOUR/12)
```

If we are 15 hours into the day, represented by DAYHOUR, then the result placed in CLOCKTIME is 3. If DAYHOUR is 12, the result is 0, 12 o'clock.

IRN and RND generate random numbers and are often used in applications involving games or simulations. Actually the argument for these functions is not needed and may be any legitimate argument; using the number 1.0 is sufficient. Let's scratch the work area and enter the program

```
>>110 LET NUMBER1=IRN(1.0)
>>120 LET NUMBER2=IRN(1.0)
>>130 PRINT "FIRST RANDOM NUMBER: ";NUMBER1
>>140 PRINT "SECOND RANDOM NUMBER: ";NUMBER2
>>150 END
```

(Note that statement 100 is omitted.) Now enter RUN and look at the result. My run shows

```
FIRST RANDOM NUMBER: 2423
SECOND RANDOM NUMBER: 363
```

These look like two legitimate random numbers. Enter RUN a second time and observe the result. My run shows

```
FIRST RANDOM NUMBER: 2423
SECOND RANDOM NUMBER: 363
```

We see that we get the same two numbers! This does not seem too random. In some applications this may be desirable, especially when checking out programs for problems. Repeatability may be useful in such a case, but in many instances it is not desirable. The RANDOMIZE statement allows us to proceed in a different way. RANDOMIZE tells BASIC that a new starting point is needed the next time a random number is generated. That is, each time the program is RUN, the execution of RANDOMIZE will cause the next use of the RND function to start at a different random number than was used in a previous execution of the program. RANDOMIZE need be used only once in a program. Let us add it to the program

```
>>100 RANDOMIZE
```

Now use RUN two times as above. My results are

```
FIRST RANDOM NUMBER: 18002
SECOND RANDOM NUMBER: 4991
```

and

```
FIRST RANDOM NUMBER: 7381
SECOND RANDOM NUMBER: 14808
```

Now we have obtained numbers that really look random from run to run. Try running the program several times, and compare the results. Incidentally, these numbers are properly called pseudo-random numbers. They have many of the useful properties of random numbers, but they are generated by a mechanical process, and not a truly random process. They are nevertheless extremely useful.

USER-DEFINED FUNCTIONS—DEF

Earlier we mentioned that BASIC does not contain a modulo function. We saw that we could produce the same results as a modulo function by writing a relatively short arithmetic expression. In applications where the modulo function is needed repeatedly, it becomes tiresome to have to write out such an expression. Indeed, this is the case for a number of application-dependent arithmetic expressions whose forms remain the same, but whose input variables change between usages. This situation is similar to our need to have functions like COS, INT, and LOG that we discussed earlier. The difference is that BASIC permits us to define our own functions in terms of arithmetic expressions which we specify.

BASIC provides a statement, DEF, which permits us to define functions representing arithmetic expressions that are used repeatedly. Instead of writing the expressions repeatedly, we can just use the function name. Let's see how this works in calculating the area of a circle when we are given the radius of the circle. Our DEF statement is

```
DEF FNAREA(RADIUS)=3.14*RADIUS*RADIUS
```

The DEF statement defines a function called FNAREA. All function names defined with DEF must begin with FN and be followed by a name fashioned in the manner of a BASIC variable name. An argument list of names follows the function name. In this case, we have only one argument, RADIUS. The arithmetic expression usually is composed of variables in the argument list. DEF simply defines the form of the function. When this statement is executed, nothing happens except that FNAREA becomes a known function to BASIC. When FNAREA is used in a BASIC statement, the expression is used to evaluate the function. For example, suppose we want to calculate the cost of a circular table that is 2 feet in radius and costs 20 dollars a square foot. We might write the statement

```
LET COST=FNAREA(2)*20.0
```

A value of 2 is used for RADIUS in the DEF expression for FNAREA, and BASIC returns a value of 3.14*2*2, or 12.56, which is multiplied by 20.

RADIUS in the DEF statement is just an artificial variable; it and any other variables found in the argument list define variables which are used in the DEF arithmetic expressions. If RADIUS is used elsewhere, it is not considered to be the same RADIUS found in the DEF.

Variables which are not arguments of the function may appear in the DEF expression. These variables take on whatever values they currently have when the function is used.

Consider the following program that is used to compute the volume of two cylinders, given their radius and height, and to compute the circumference of a circle with a radius of 4.5.

```
100 DEF FNCYLVOL(RAD,HGT)=PI*RAD*RAD*HGT
110 PI=3.14
120 RAD=4.5
130 PRINT "CIRCUMFERENCE OF A CIRCLE",RAD*2*PI
140 PRINT "FIRST CYLINDER",FNCYLVOL(2,10)
150 PRINT "SECOND CYLINDER",FNCYLVOL(6.25,15)
160 END
```

Line 100 uses a DEF to define a function, FNCYLVOL, containing two arguments which represent the radius and height of the cylinder. Lines 110, 120, and 130 are used to compute the circumference of a circle. When line 140 is executed, 2 is substituted for RAD, and 10 is substituted for HGT in the expression defined by FNCYLVOL. Because PI in that expression is not an argument variable, the value of PI is taken as the value set in line 110. The execution of the statement at line 150 is similar. When the program is finished, RAD has a value of 4.5, not 6.25, even though 6.25 was substituted in the RAD variable of line 100 when executing line 150. This last statement emphasizes the fact that argument variables are not the same as variables found elsewhere in the program, despite the fact that the same names may be used.

STRING FUNCTIONS

You might be wondering whether there are string functions as well, or maybe you hadn't considered such a possibility. The answer is yes, they do exist, and they are very useful. String functions operate on strings in various ways. Some produce strings as output as the result of examining a numeric argument. Others produce a numeric value (usually an integer) as the result of examining a string. String functions are highly useful in many applications. String functions which return strings, i.e., produce strings as output, end with a $.

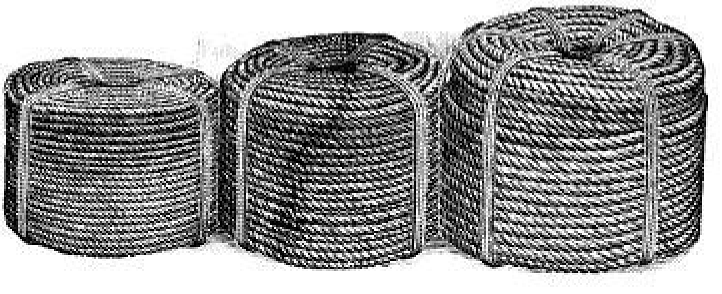


Figure 5.2 A string function? No. No.

In the list and brief description of the string functions that follow, A$ and B$ represent strings, n represents an integer number, and X represents a number, numeric variable, or arithmetic expression.

String Functions

Function	Meaning
ASC(A$)	Returns the ASCII numeric value of the first character in A$.
CHR$(X)	Produces a single character which is the ASCII character corresponding to the value of X.
DATE$(A$)	If A$ is "yymmdd", sets the date for 3102 terminals. yy (year), mm (month), dd (day). If A$ is "" (null), date is returned.
HEX$(X)	Produces a four-digit hexadecimal ASCII representation of X.
LEN(A$)	Returns the current length, i.e, number of characters, of A$, exclusive of trailing null characters.
POS(A$,B$,n)	Finds the first position in A$, beginning at position n, which contains B$. Return -1 if no match.
STR$(X)	Converts X to its character representation. For example, 1.34 to "1.34".
TIME$(A$)	If A$ is "hhmmss", sets the time for 3102 terminals. hh (hr), mm (min), ss (sec). If A$ is "" (null), time is returned.
VAL(A$)	Convert A$ to its numeric representation. For example, "1.34" to 1.34.
VALC(A$)	Same as VAL except that it sets an error condition when A$ contains bad information. The error may be detected by the ON statement.

Probably the most frequently used string functions are LEN and POS. In the program

```
100 NAME$="BOB SMITH"
110 LENGTH=LEN(NAME$)
120 WHERESMITH=POS(NAME$,"SMITH",0)
130 PRINT "LENGTH OF NAME: ";LENGTH
140 PRINT "SMITH LOCATED AT CHARACTER: ",WHERESMITH+1
150 END
```

the use of LEN at line 110 and POS at line 120 produces as output

```
        LENGTH OF NAME: 9
        SMITH LOCATED AT CHARACTER: 5
```

We see that although NAME$ has a maximum default size of 11 characters, it really contains only 9 characters. Furthermore, the search for "SMITH" indicated in line 120 begins at position 0, the first position in the string NAME$. To obtain the character position that we might normally expect the string "SMITH" to begin, we add 1 to WHERESMITH on line 140.

ASC, CHR$, and HEX$ relate to ASCII character codes and will be left to the interested reader to explore. The ASCII character codes are the numeric equivalents of all the allowable characters in BASIC. A table of these codes is located in the appendix.

VAL, VALC, and STR$ can be useful in many applications requiring conversion of strings representing numbers to numeric values, and vice versa. The interested reader should explore these independently. There are several peculiarities of the VALC function that you may want to be aware of if the function is of interest to you. In particular, VALC needs to have numbers on both sides of any decimal point. See the Structured BASIC Instruction Manual for a more detailed explanation.

DATE$ and TIME$ are used to set and read the time from the Cromemco 3102 terminal. When the date or time are set, a LET may be used with a dummy string variable on the left, as in

```
        LET DUMMY$=DATE$("830804")
```

INSERTING INTO A STRING—EXPAND

A particularly useful statement related to the string functions is the EXPAND statement. It can be used to help insert characters into the middle of a string. This statement is used to insert null characters into a string variable. The substring of null characters can then be replaced with another string using a LET. For example

```
EXPAND MAGAZINE$(5),3
```

inserts three null characters in MAGAZINE$ starting before position 5 in MAGAZINE$.

Consider the need to insert the string "ALAN" into the middle of "ROBERT BROWN". A program that performs this insertion is

```
100 DIM NAME$(20)
110 LET NAME$="ROBERT BROWN"
120 EXPAND NAME$(7),5
130 LET NAME$(7,11)="ALAN "
140 PRINT "HIS NAME IS ";NAME$
150 END
```

produces

```
HIS NAME IS ROBERT ALAN BROWN
```

Position 7 in NAME$ is occupied by the letter B. We insert 5 null characters to accommodate the extra space needed in the middle name by using EXPAND in line 120. Line 130 actually inserts the name "ALAN ", with a blank provided to take the place of the fifth null character.

BINARY FUNCTIONS

Structured BASIC contains several functions that are referred to as binary functions. These are probably only of interest to advanced users, and they will only be mentioned briefly here. There are five binary functions

Function	Meaning
BINADD(e1,e2)	Binary addition
BINSUB(e1,e2)	Binary subtraction
BINAND(e1,e2)	Binary logical "and"
BINOR(e1,e2)	Binary logical "or"
BINXOR(e1,e2)	Binary logical "exclusive or"

The arguments e1 and e2 represent integer expressions, 16-bit numbers, or hexadecimal constants. More information on these functions may be found in the Structured BASIC Instruction Manual.

PRINT USING WITH NUMBERS

By now you may have discovered on your own that when numbers are printed with PRINT they may be printed in a format that is not entirely useful for certain applications. In fact, it may have occurred to you that there should be more control of how an output line is formatted. The PRINT USING statement gives you this control. PRINT USING is similar to PRINT, except that a format for the variables printed is specified. This format is contained in a string literal or string variable included with the statement. A simple illustration using a string literal for a format is

```
PRINT USING "####.#",3.12
```

The string "####.#" is the format. It indicates that exactly six print positions will be output, one for each character in the format. The sharp signs (#) indicate where numbers are to occur, and the period shows where the decimal point is. The output of this statement is

```
   3.1
```

Three spaces occur before the 3 since there are no digits in these positions. Only one decimal place is printed, as indicated by the format. The result is rounded.

Now consider

```
PRINT USING "####.#",1000.4,400.77,100
```

This produces

```
1000.4 400.8 100.0
```

That is, each number is printed with the same format until the list of numbers is exhausted. Note carefully that the space between the numbers is generated because there is no digit there in the second and third numbers, and not because BASIC politely put them there. To make sure there is a space between numbers, we should actually use a format with a trailing blank, as in "####.# ". The trailing blank in this format is always output, and leaves a desirable separator between the fields. Note that the number 400.77 is rounded when placed in the output field.

Suppose the following is used

```
PRINT USING "###.# #####.## ",100.4,12345.67
```

The output is

```
100.4 12345.67
```

Hence, two fields are defined in the format and they are used successively by the items in the list.

What happens if there are more items in the list than fields defined in the format? The format is recycled. So

```
PRINT USING "###.# #####.## ",100.4,12345.67,144.5
```

produces

```
100.4 12345.67 144.5
```

Textual information may be included in the format, as in

```
PRINT USING "ANSWER QUESTION ##.# AND SCORE ##",12.3,20
```

which results in

```
ANSWER QUESTION 12.3 AND SCORE 20
```

We will say more about using textual information in a format in a short while.

Of course, variables can be used with PRINT USING as in the program segment

```
400 LET MYFORMAT$="SIZE: ##"
410 LET SIZE=15
420 PRINT USING MYFORMAT$,SIZE
```

A common problem with formats occurs when not enough room in a field is specified. For example, in the preceding sequence of statements, suppose SIZE is set to 150 instead of 15. BASIC cannot fit 150 into the two-digit field specified, so it prints asterisks instead. The result is

```
SIZE: **
```

If you try to print a negative number with what we have learned thus far, you will be surprised to find that the # does not help us. The # specifies that the field it represents is to contain a number or space and nothing else. Another format symbol, the - (minus) or + (plus) can be used to get us around this problem. Incidentally, the #, -, and + are called format specifier symbols when they are used to specify a format. When present in a format field, the first sign specifier in a field indicates that a sign or blank is to be output in the corresponding position. Subsequent occurrences of a sign in the field indicate that a sign, number, or space is acceptable as output characters. Let's try it and see what happens. Scratch your work area and enter

```
>>100 PRINT USING "-###.# ",  50.0, -5.4,-234.5, 6000
>>110 PRINT USING "+###.# ", -50.0,  5.4, 234.5, 6000
>>120 END
```

Enter RUN. We find the output

```
 50.0 -  5.4 -234.5 ****.*
 50.0 +  5.4 +234.5 ****.*
```

The sign is always in a fixed position. The sign output for the positive number in the first format is just a blank. In the second format the sign is always printed. The use of the sign specifier here indicates that the corresponding position is reserved for a blank or a sign. Hence, the value 6000 cannot be accommodated, and is therefore printed as a field of asterisks.

It may not always be desirable to have the sign occur in a fixed position. Successive use of sign specifiers permits the sign to "float" in the field. For example

```
PRINT USING "--##.# ",-300.0,-20.0,-1.0,6000
```

produces

```
-300.0  -20.0  - 1.0 ****.*
```

The sign is allowed to float to the right-most position of the two fields containing the sign specifier. Here the second minus is able to accommodate a digit as well as a sign. The first minus, as shown previously, permits only a sign; so the 6000 prints as asterisks.

Several signs may appear in succession as in the following example format strings

```
"++++.##"  "-----.##"  "---##.###"  "-----.---"
```

In the last string, the minus symbols in the decimal portion of the field are the same as # there. That is, the sign output is always placed to the left of the decimal.

We have tacitly used the decimal point as a specifier. A comma has a similar use. For example

```
PRINT USING "#,###.## ",1000.50,234.56
```

produces

```
1,000.50   234.56
```

We see that the size of the number determines whether the comma is output, as we would hope.

A particularly useful specifier for commercial applications is the $ symbol. It is very similar to the sign specifiers. Consider the sample program

```
100 PRINT USING "$###.## ",5.0,100.0
110 PRINT USING "$$##.## ",250.0,10.0,0.50
120 PRINT USING "-$##.## ",-40.0,2.0,-800.50
130 PRINT USING "$-##.## ",-5,25
140 END
```

This program produces

```
$  5.00 $100.00
$250.00  $10.00  $ 0.50
-$40.00  $ 2.00 ****.**
$- 5.00 $ 25.00
```

The asterisks are produced in the third output line because in line 120 the -800.50 is too large; there is no room for the digit 8. Its position is needed for the $ according to the format.

When a fixed $ format is used, specifiers preceding the $ are made blank unless they are sign specifiers, in which case a sign may be output. In the case of a floating $, only one specifier may precede the first $. It must be a sign.

In a number of scientific applications, it is useful to print a number in E-notation. The symbol ! used successively in four locations allows E-notation output. The example

```
PRINT USING "--.###!!!! ",100.0, 0.05, -1200.0
```

outputs

```
 1.000E+02  5.000E-02 -1.200E+03
```

The most significant digit is placed in the left-most position that a number is allowed.

There are two other specifiers that may be used in connection with numeric fields, the & and *. These specifiers are like the # in that they provide for digits. But when there is no leading digit to print, the & indicates the leading digits of zero, and * indicates the leading digits are asterisks. From the sample program

```
100 PRINT USING "&&&.## ",100,10,1
110 PRINT USING "***.## ",100,10,1
120 PRINT USING "-&&&.# ",-20,200,5
130 END
```

we find the results

```
100.00 010.00 001.00
100.00 *10.00 **1.00
-020.0  200.0  005.0
```

PRINT USING WITH STRINGS

Up to this point, our concern has been with numbers. It is certainly possible to use strings in the output list of a PRINT USING. When a string appears in a PRINT USING, any of the specifiers we have discussed reserve positions for characters of the string. Variable or literal strings are permitted.

For example

```
100 CITY$="DETROIT"
110 STATE$="MICHIGAN"
120 PRINT USING "###### AND ######### BASE","FIRST","SECOND"
130 PRINT USING "CITY:&&&&&&#### STATE:-$***.###",CITY$,STATE$
140 PRINT USING "####","FINISHED"
150 END
```

produces upon execution

```
FIRST  AND SECOND   BASE
CITY:DETROIT   STATE:MICHIGAN
FINI
```

In the preceding example, we see that the string is always left-justified, i.e., pushed to the left, in the field. In line 120, the strings "FIRST" and "SECOND" are shorter than the fields into which they are placed. Looking at the output, we see that blanks fill out the portions of the fields that are not occupied by these strings. The same is true for the string variables used in line 130. This is sometimes called padding with blanks on the right. The string output in line 140 is truncated because it is too long for the field. Although lines 120 and 130 show a mixture of specifiers in the format, it is usually a good idea, for consistency, to use just # for string fields.

There is an interesting situation in which the field containing a string variable does not seem to be padded on the right when the string is shorter than the field. As stated in our earlier discussions of string variables, if a string is placed in a variable which has a size greater than the string, the string is padded to the right with null characters. These null characters are actually characters. They appear to occupy no space in the output when printed. If we insist on printing them, which may happen when a substring reference is given, the output may seem erratic. The program

```
100 CITY$="CHICAGO"
110 PRINT USING "CITY(##########)",CITY$
120 PRINT USING "CITY(##########)",CITY$(0,8)
130 END
```

produces the output

```
CITY(CHICAGO   )
```

```
CITY(CHICAGO )
```

CITY$ is a string variable and has a default size of 11 characters; "CHICAGO" contains 7 characters which occupy positions 0 through 6. There are 4 null characters at the end of CITY$. In line 110, BASIC assumes the end of string is encountered when it finds null characters, and it pads the output with blanks. However, in line 120, we have explicitly asked to print the 2 null characters in our substring reference. They occupy no space in the output, and this is shown by what appears as two missing spaces before the final parenthesis.

PRINT USING WITH NUMBERS AND STRINGS

Of course, there is nothing to prevent us from using PRINT USING with both numbers and strings. For example

```
100 PRINT USING "#####.## ######",300.44,"KAREN"
```

produces

```
300.44 KAREN
```

There is really nothing very complicated about such forms. The only problem that we face is that of making sure that our list of variables and literals matches the format items.

PRINT USING—MISCELLANEA

In some of our examples, non-specifiers appear. In the previous example, CITY, (, and) are non-specifiers. When such symbols appear, they are assumed not to specify any portion of a field and they may be used freely. Only the specifiers

```
#  *  &  $  ,  +  -  .  !
```

define a format field. Hence, if these special symbols are needed as descriptive information in the output, they must be output from strings. For example

```
100 PRINT USING "############","HI, THERE!"
110 END
```

This produces

```
HI, THERE!
```

containing the special characters the comma and the exclamation point.

Like the PRINT statement, PRINT USING permits the inclusion of a mixture of numeric and string data and variables in the list following the format string. A semicolon at the end of the list suppresses the carriage return. The example

```
100 WGT=20.4
110 NAME$="SMITH"
120 PRINT USING "###.## ########",100.4,NAME$;
120 PRINT USING "##.# POUNDS",WGT
130 END
```

produces the output

```
100.40 SMITH   20.4 POUNDS
```

Finally, let's combine much of what we have learned by presenting some of the more interesting format combinations in the following table. An entry corresponds to the result of applying a data item to a format, e.g., applying +20 to ### produces 20.

Formats At a Glance
Print Items

FORMAT	+20	-20	+ 5	- 5	100	"ABC"
###	20	20	5	5	100	ABC
***	*20	*20	**5	**5	100	ABC
&&&	020	020	005	005	100	ABC
&*#	020	020	0*5	0*5	100	ABC
+##	+20	-20	+ 5	- 5	***	ABC
++#	+20	-20	+5	-5	***	ABC
+++	+20	-20	+5	-5	***	ABC
-##	20	-20	5	- 5	***	ABC
--#	20	-20	5	-5	***	ABC
---	20	-20	5	-5	***	ABC
$##	$20	$20	$ 5	$ 5	***	ABC
$$#	$20	$20	$5	$5	***	ABC
$--	***	***	$ 5	$-5	***	ABC
-##!!!!	20E+00	-20E+00	50E-01	-50E-01	10E+01	ABC
---.--	20.00	-20.00	5.00	-5.00	***.**	ABC
---,--	20	-20	5	-5	1,00	ABC
OUT:---	OUT: 20	OUT:-20	OUT: 5	OUT: -5	OUT:***	OUT:ABC

Review Questions and Exercises

1. Two of the following LET statements are invalid. Which ones are they?

```
LET TIME=TIME*22.4
COUNT=COUNT+1
LET XPERCENT=(YEAR*33.4/INVENTORY)+18.004/8.2
LET TOTAL=(100.0+SALES-TAXES))+400
LET LOSS=DELTA*(150-WORK
```

2. Write a LET statement that takes the sine of THETA and stores the result in SINTHETA.

3. What does the RANDOMIZE statement do?

4. Why do the following statements cause KILOMETERS to contain 16.0?

```
900 DEF FNCONVERT(MILES)=1.6*MILES
910 NO'OF'MILES=10.0
920 KILOMETERS=FNCONVERT(NO'OF'MILES)
```

5. IF ALPHA$ contains "ABCDEF", why is the value returned by POS(ALPHA$,"C",0) a 2?

6. What BASIC statement permits you to insert null characters into a string variable?

7. What is a format? Give an example of a format that can be used to print a three-digit number without any decimal point.

8. Which PRINT USING statement is wrong?

```
PRINT USING "####.#",NUMBER
PRINT USING "####.#";NUMBER
```

9. What is the difference between the formats "--.##" and "-#.##" when the number -5.0 is printed? When -0.2 is printed?

10. Can string literals and variables be printed with PRINT USING?

Summary

PRINT USING
  Permits numeric and string data to be

formatted in specific fields and positions of an output line.
A format is specified with a string or string variable.
Special format specifiers define the permissible contents of a field.
Formats allow decimal point positioning.
Commas may be inserted in fields to represent units of 1000.
Descriptive information may be included in formats.
Arithmetic signs and dollar signs are allowed to "float" at the beginning of a field.
Strings are left-justified in a field; any specifier defines a character field.
Numbers may be output in E-notation.
Mixtures of numeric and string data may appear in the PRINT USING list.
A semicolon at the end of a PRINT USING list suppresses the carriage return.

LET

Arithmetic expressions are evaluated from left to right but with a preference that is dependent upon the operators involved. The preference is in the order: negation, exponentiation, multiplication and division, addition and subtraction.
Parentheses may be used freely to order and clarify the order of calculation.
A number of useful arithmetic functions may be included in arithmetic expressions.
Functions contain arguments which are provided in a list enclosed in parentheses following the function name.
String functions exist for manipulating string data.
Several binary functions are available for performing arithmetic and logical operations.

RANDOMIZE

Permits the RND and IRN functions to initialize at new random numbers when they are first used.

DEF

Allows user-defined arithmetic functions.
Function names must begin with FN.

EXPAND

When used with LET, provides a way of inserting substrings into the middle of string variables.

CHAPTER 6
EDIT Facilities

Now that we have learned several BASIC statements, we are able to construct some fairly sizable programs. As a result, our programs are more subject to modification. Earlier, in Chapter 2, we learned how to make modifications by deleting a line, or by retyping a line. In this chapter, we will take a brief but useful diversion into some commands which will help us modify and construct programs and program statements. BASIC provides four very useful commands for this purpose: EDIT, FIND, CHANGE, and AUTOL. You should become quite familiar with these commands because they can save you a lot of time in entering and correcting program lines.

EDITING LINES—EDIT

As a matter of course, it is desirable to use EDIT to make changes in lines rather than to retype a line. This is preferable not only because it reduces typing time, but also because it reduces the possibility of introducing new mistakes into a line which originally may have suffered only minor problems. Let's see how EDIT works.

Scratch your work area and enter

```
>>100 PRINT "THIS IS A TERBLE MIS TAKE"
>>110 PRINT "THIS SHOULD BE THE FIRST LINE",ABC
```

Now enter EDIT 100 to edit line 100 as in

```
>>EDIT 100
```

and you will get the response

```
-  100   Print"THIS IS A TERBLE MIS TAKE"
:
```

You are now in the edit mode. BASIC has positioned your cursor after the colon and is waiting for a response. You can leave the edit mode by hitting the escape key on your keyboard. Assuming that we want to continue, BASIC expects us to move the cursor from a point on the line containing the colon to the point where we wish to make a change. The cursor can be moved by using the space bar to move right or the delete key to move left. Let's try to remove the space between MIS and TAKE. Move the cursor to the space we want to delete and type a D as in

```
-  100   Print"THIS IS A TERBLE MIS TAKE"
:                                  D
```

This symbol designates that we want to delete the character above it. Several D's may be entered if needed. Here we only need one. Now hit the carriage return and you should get

```
-  100   Print"THIS IS A TERBLE MISTAKE"
:
```

The space between MIS and TAKE is missing, as we wanted. We are still in the edit mode, as indicated by the colon. Now let's correct the spelling of the word "terrible." Move the cursor to below the B in TERBLE and type IRI as in

```
-  100   Print"THIS IS A TERBLE MISTAKE"
:                           IRI
```

The first I designates the insert operation and anything typed after it will be inserted before the I on the line above. Hit the carriage return and you should get

```
- 100   Print"THIS IS A TERRIBLE MISTAKE"
:
```

Now hit the carriage return without entering anything, and the familiar BASIC prompt will reappear. We will be out of the edit mode, and the changes we made will be in effect. If we had hit the escape key, we would leave the edit mode without the changes being in effect. Enter LIST to list the work area and you should find

```
100   Print"THIS IS A TERRIBLE MISTAKE"
110   Print"THIS SHOULD BE THE FIRST LINE",ABC
```

Incidentally, entering EDIT without a line number following it causes BASIC to prompt you for edit changes for each line in the work area. Hitting the carriage return causes the next line to become available for editing, until all the lines have been exhausted.

Note that only one I symbol is allowed on a line because any other I that follows it is interpreted as part of the character string to be inserted into the line.

Now let's assume that ABC in line 110 is extraneous and thus should be eliminated. We could enter EDIT 110 and place DDDD under ",ABC" but let's try one other edit symbol, K. Here we go! Make sure you hit the carriage return after entering K, as shown.

```
>>EDIT 110
- 110   Print"THIS SHOULD BE THE FIRST LINE",ABC
:                                            K
- 110   Print"THIS SHOULD BE THE FIRST LINE"
:
```

The K means kill all characters from the position of the K to the end of the line.

Do not hit the carriage return yet; we have more to do on this line. We now suppose, as line 110 says, that it should be ahead of line 100. Let's modify its line number as in the sequence

```
- 110   Print"THIS SHOULD BE THE FIRST LINE"
: DDDI90
-  90   Print"THIS SHOULD BE THE FIRST LINE"
:
```

Here the notation DDDI90 effectively says to delete 110 and to insert 90 in its place. The line below our entry of DDDI90 shows that this has happened. Any portion of a line may be edited, including the line number. Now hit the carriage return to exit from the edit mode and enter LIST. You should have

```
90 Print"THIS SHOULD BE THE FIRST LINE"
100 Print"THIS IS A TERRIBLE MISTAKE"
110 Print"THIS SHOULD BE THE FIRST LINE",ABC
```

We see that line 110 has not been eliminated and that all of our changes have been placed on a new line 90. When EDIT is operating on a line of text from the program, it does not make the changes directly in the work area until you are finished with the line. It then inserts the edited text into whatever line is shown in the final edit. If we had not changed the 110 to 90, line 110 would have been replaced. This can be a very effective way of moving lines around in BASIC without retyping the entire line. To get rid of line 110 in the work area, we simply enter 110. As we learned in an earlier chapter, this deletes the line, and we are left with lines 90 and 100.

The EDIT command actually comes in four varieties. Examples of each are

```
EDIT
EDIT 600
EDIT 300,540
EDIT 2000,
```

The first form allows us to successively to edit all lines in a program in the fashion mentioned previously. The second form has just been illustrated. The third form tells BASIC to apply EDIT to each of the lines from 300 to 540 successively. Finally, the last form applies from line 2000 through the last program line.

LOCATING AN ITEM—FIND

There are times when we want simply to locate a particular string in our work area. Visual inspection of statements using LIST can be time-consuming. The FIND command allows us to locate a string quickly. When we enter FIND as in

```
>>FIND
```

we are prompted with

```
FIND:
```

BASIC wants us to enter the string that it should search for. Entering "Print" as in

```
FIND:Print
```

will cause BASIC to list all the lines in which the string "Print" occurs.

There are four forms of FIND:

```
FIND
FIND 400
FIND 8900,9500
FIND 6000,
```

The first case has just been discussed. The other three forms allow us to restrict the lines searched in the same manner that EDIT can be limited to certain lines.

The FIND command can be terminated at any time by hitting the escape key on your terminal.

MAKING MULTIPLE CHANGES—CHANGE

The CHANGE command is useful when we want to apply a similar change over a range of lines. When CHANGE is entered, a prompt, FROM:, is issued by BASIC requesting the string that is to be changed. After entering the string and hitting the carriage return, you are again prompted by TO: to enter the string that is to replace the one you just entered in response to the FROM: prompt. Here is a typical sequence which changes all occurrences of the symbol @ to PRINT

```
>>CHANGE

FROM:@

TO:PRINT
```

When you hit the carriage return after entering the string PRINT, BASIC begins searching for all occurrences of the @ symbol (a single character string). When it finds the string, it displays the line the string is contained in, positions the cursor under the string, and waits for a response. You may enter a carriage return to reject the change at that point, a C to accept the change, or an asterisk(*) to accept all changes from that point onward. In the first two instances, BASIC will continue to wait for a response from you as it finds each new occurrence of the string. In the last case, BASIC will automatically make the required changes as it finds each string to be replaced, without waiting for you to respond.

Like EDIT and FIND, CHANGE has four forms which allow you to select the lines that you wish it to operate on. The CHANGE command may be terminated by hitting the escape key.

LETTING BASIC NUMBER YOUR LINES—AUTOL

AUTOL is useful when a number of statements are to be added to your program and you do not want to type the line numbers yourself. AUTOL automatically supplies the line numbers. As each number is generated, you enter the statement on the line whose line number is generated. You may terminate the command simply by responding with a carriage return, without entering any other text for the line, or by hitting the escape key. An example of this command is

```
AUTOL 500,10
```

The first number, 500, is the first line number you want generated. The second number, 10, is the amount by which you want each subsequently generated line number to be incremented. Clear your work area and enter

```
>>AUTOL 1000,5
```

BASIC will respond with

```
>>1000
```

The 1000 was generated by AUTOL. Now enter PRINT after the 1000 as in

```
>>1000 PRINT
```

and BASIC will respond with

```
>>1005
```

It is waiting for you to enter something on line 1005. Enter REM or some valid statement and continue experimenting until you are satisfied with your understanding of AUTOL.

Figure 6.1 Back to the easy life.

Review Questions and Exercises

1. How does the CHANGE command respond to a C, space, carriage return, and asterisk?

2. How are entries to an AUTOL command terminated?

3. Is it possible to EDIT a line number? What happens?

Summary

EDIT

Permits the editing of lines on a character-by-character basis, as needed, with special edit symbols.
Quick summary of EDIT codes:
K : Kill remaining portion of line
I : Insert input text
D : Delete text

FIND

Finds all occurrences of a specified string of characters by displaying the lines containing the string.

CHANGE

Changes all occurrences of a specified string to another string while providing the opportunity to selectively decide whether the change should be made.
Quick summary of CHANGE codes:
C : Accept change
* : Accept all following changes
Carriage Return : Reject Change

AUTOL

Automatically generates line numbers and prompts for the statement for the line.

CHAPTER 7

Looping, Branching, and Logic

Up to this point, the programs that we have been able to write contained statements that executed in a sequential order. Progress through the statements was made by moving from the first line of code to the last line, without any side trips or jumps. While this method of operations is useful in some applications, this is not how powerful, flexible, and really useful programs are written. There are times when we would like to move off the beaten trail and do something more complicated. And, like so many efforts that take us off the beaten trail, the statements that we learn about will not be used in our applications without some risk. The risk evolves from the fact that these statements also make it more complicated to get our programs to behave correctly. We will attempt to tackle that problem when we get to the chapter on debugging. Nevertheless, the statements we will learn in this chapter disclose a whole new dimension in our ability to solve complex applications on the computer. We should approach them with some caution, but with much enthusiasm for their ability to help us solve difficult application problems.

SIMPLE LOOPS—FOR AND NEXT

The first of these statements are the FOR and NEXT statements. They always occur as a pair, and they permit us to execute repetitively a section of statements for a fixed number of times. Such an activity is usually regarded as a loop, or simple loop. Often the two statements are referred to as defining a FOR-NEXT loop. Here is a program that we will enter to illustrate the two statements. Clear your work area first, and enter

```
>>100 FOR J=1 TO 4
>>110 PRINT "VALUE OF J: ";J
>>120 NEXT J
>>130 PRINT "FINAL VALUE OF J: ";J
>>140 END
```

When RUN, this program produces

```
VALUE OF J: 1
VALUE OF J: 2
VALUE OF J: 3
VALUE OF J: 4
FINAL VALUE OF J: 5
```

Statement 110 is executed four times; each time it is executed, J is increased by 1. The reason for this is that the FOR statement tells BASIC to loop by starting with J and 1 and, each time the loop is completed, to increase J by 1. The 4 following the TO in the FOR statement tells BASIC that looping should stop when J has exceeded 4. The NEXT statement defines the end of the loop, causes J to be increased by 1, and the execution to be returned to the FOR. When J becomes 5, the loop is terminated, and the program goes to the statement following the NEXT. We sometimes refer to this as falling through the bottom of the loop, or falling out of the loop. The variable J is referred to as the loop variable and 1 and 4 are referred to as the starting and limiting values.

It is important to understand that the NEXT is actually responsible for both incrementing and testing the loop variable, even though symbolically the FOR statement appears to have both of these functions. In the example program, the reason for the termination of the loop is that after the NEXT statement increments the loop variable, J, it checks the loop variable value against the limit. If the value is too large, the loop is terminated by going to the statement following the NEXT.

In our example program, only one statement is included inside the loop, but there is no reason why more statements could not be included. If you LIST the above program, you will notice that the statements of the loop are indented. BASIC performs this editing to make the loop stand out more readily in a program listing.

A common program structure seen with these two statements is demonstrated in the following program. Scratch your work area and enter

```
>>100 DATA 5,15,20,35,40,50
>>110 READ NDATA
>>120 TOTAL=0
>>130 FOR LOOP=1 TO NDATA
>>140 READ VALUE
>>150 LET TOTAL=TOTAL+VALUE
>>160 NEXT LOOP
>>170 PRINT "TOTAL: ";TOTAL
>>180 END
```

This program adds the five values 15, 20, 35, 40, and 50 and prints the total. Line 100 contains the five values in a DATA statement. The first value in the DATA statement, 5, is the number of data values and is read into the variable NDATA in line 110. TOTAL is used to sum the results and is set to zero in line 120. The FOR-NEXT loop uses the loop variable LOOP and the loop is executed with LOOP starting at 1 and continuing until it becomes greater than NDATA, or 5. The loop is executed 5 times. Inside the loop, the next data item is read into VALUE at line 140. The LET statement adds VALUE to TOTAL and puts the result in TOTAL. When the loop is completed, line 170 is executed next. The loop containing LET statements that sum a series of numbers is commonly found in many applications.

Try executing the above program. You should find the result

```
TOTAL: 160
```

Something else to notice about our example is that the limiting value is a variable. The starting value can be a variable as well. In fact, either value can be an expression.

COMMON LOOP ERRORS

Let's consider several simple programs involving a FOR-NEXT loop which are incorrect in some way.

```
Program 1:
100 FOR K=1 TO N
110 PRINT "VALUE OF K: ";K
120 NEXT K
130 END

Program 2:
100 FOR LOOP=1 TO 8
120 PRINT "VALUE OF LOOP: ";LOOP
130 NEXT LOP
140 END

Program 3:
100 FOR Q=1 TO 50
110 PRINT "VALUE OF Q: ";Q
120 END

Program 4:
100 TOTAL=0
110 FOR NUMBER=1 TO 100000
120 TOTAL=TOTAL+NUMBER
130 NEXT NUMBER
140 PRINT "TOTAL: ";TOTAL
150 END

Program 5:
100 FOR NUMBER=1 TO 10
110 PRINT "PI*";NUMBER;" IS ";NUMBER*3.14
120 NUMBER=NUMBER+0.5
130 PRINT "PI*";NUMBER;" IS ";NUMBER*3.14
140 NEXT NUMBER
150 END
```

Program 1 will execute the loop exactly one time with K set to 1. The reason for this is that, first, N is zero. Variables which are not initialized, i.e., set to a value by the program, are set to zero by BASIC. Second, recall that the loop variable is incremented and tested at the NEXT. Hence, when the NEXT is reached, we fall out of the loop. It is very common to not initialize a limiting variable and thus to have a loop execute fewer times than expected. A FOR-NEXT loop is always executed at least once.

In Program 2, an error message will appear when line 130 is executed. The loop variable is not the same in the NEXT as in the FOR. In Program 3, the NEXT is missing entirely and an error message is generated when the END is executed.

Program 4 does not really contain any mistakes. It attempts to add all the numbers from 1 to 100000. If left to run long enough, it will eventually succeed. We may really have wanted to add only the first 1000 numbers, and have mistakenly entered 100000. The problem here is that we are asking to loop 100,000 times. You will find that although computers are quick, they do have limitations. Looping this many times will be slow. It may be a good 7 to 8 minutes or more before the loop is completed. The point of this program is that we must exercise some care in what we expect a computer to do; the computer may not match our expectations in computational speed. Our program may appear to fail as a result of a programming error, when this is not actually the case.

The problem illustrated by Program 4 is often encountered, and we need a way of getting out of the loop to repair it in case it takes too much time. The execution of BASIC programs may be stopped by hitting the escape key on your keyboard. BASIC will stop and display a message telling you at which line you have stopped. You can repair the program and enter RUN again. We will see that there are other ways to restart a stopped program in the chapter on debugging.

Program 5 apparently is an attempt to print 1*pi, 2*pi, ..., 10*pi and 1.5*pi, 2.5*pi, ..., 10.5*pi where pi is the mathematical constant 3.14. The loop variable is changed inside the loop at line 120. This is a bad practice. This program will work, but in general, this type of coding is to be avoided. Very often, you will find yourself staying inside loops either a lot longer or a lot shorter than you expected, if you modify the loop variable.

STEP CONTROL IN FOR-NEXT

Another form of the FOR-NEXT contains a step size and it is illustrated by the example

```
100 FOR COUNT=1 TO 10 STEP 5
110 PRINT "COUNT IS: ";COUNT
120 NEXT COUNT
130 END
```

The STEP 5 portion of the FOR at line 100 indicates that the increment should be 5 instead of 1. In this example, COUNT will start at 1 and then be incremented by 5 at the NEXT statement. The result is

```
COUNT IS: 1
COUNT IS: 6
```

The loop is completed when COUNT is incremented to 11; since 11 is greater than 10, the limiting value, the loop is terminated. Remember, as noted in the Program 1 example, the loop variable is incremented at the NEXT statement and its value is then tested against the limit value.

There are times when it is useful to loop from a large number down to a small number using a negative step size. BASIC permits this operation with the FOR-NEXT statements. Here is an example

```
100 FOR COUNT=10 TO 1 STEP -5
110 PRINT "COUNT IS: ";COUNT
120 NEXT COUNT
130 END
```

The STEP -5 portion of the FOR at line 100 indicates that the increment is -5. In this example, COUNT will start at 10 and then be decremented by 5 at the NEXT. The result is

```
COUNT IS: 10
COUNT IS: 5
```

When the step size is negative, the test used to determine when the loop is finished is different than when the step is positive. When decrementing, we drop out of the loop when the loop variable is less than the limiting value.

NESTED LOOPS

There is no reason why loops cannot be nested within one another, as in

```
100 DATA 100,154
110 DATA 120,135
```

```
120 DATA 215,302
130   FOR ROWS=1 TO 3
140   SUM=0
150     FOR COLS=1 TO 2
160     READ VALUE
170     SUM=SUM+VALUE
180     NEXT COLS
190   PRINT USING "SUM OF ROW ## IS #####.##",ROWS,VALUE
200   NEXT ROWS
210 END
```

This program sums the rows of a 3 x 2 table of data values found in the DATA statements. The outer loop is controlled by ROWS and is responsible for looping over rows. The inner loop is controlled by COLS and is responsible for looping over the column data within a row. The terminology inner and outer stems from one loop being inside the other. You might try entering the previous program, and examine how it works.

A common fault in constructing programs which contain nested loops is to cause an inner loop to fall partly outside of the outer loop. This is illustrated by

```
100 FOR ROWS=1 TO 3
110 FOR COLS=1 TO 2
120 PRINT "ROW ## COLUMN ##",ROWS,COLS
130 NEXT ROWS
140 NEXT COLS
150 END
```

When executed, this program will stop at 130 with BASIC issuing an error message. The cause of the problem is that BASIC expects to increment COLS since it was the the loop variable in the most recent FOR statement encountered. Instead it finds a NEXT for ROWS. The correct program is

```
100 FOR ROWS=1 TO 3
110 FOR COLS=1 TO 2
120 PRINT "ROW ## COLUMN ##",ROWS,COLS
130 NEXT COLS
140 NEXT ROWS
150 END
```

Another way of modifying the execution order of programs is with the GOTO statement. This statement is found in many programming languages and has a controversial reputation in modern "structured" programming techniques. Nevertheless, you will find it very useful. As you become more accustomed to programming, you may find that in large application programs, the DO and ENDDO statements, coupled with the IF statement, are useful replacements for the GOTO. The IF statement will be introduced in this chapter and the DO and ENDDO in a later chapter.

TRANSFERRING CONTROL—GOTO

The GOTO simply causes execution to continue at the line number identified in the the GOTO. For example, when GOTO 400 is executed, the next statement executed will be at line 400. Scratch your work area and enter the example program

```
>>100 GOTO 130
>>110 PRINT "ONE MAN'S MEAT IS ANOTHER MAN'S POISON"
>>120 GOTO 140
>>130 PRINT "FOUR SCORE AND SEVEN YEARS AGO ..."
>>140 END
```

Enter RUN, and you should get the results

```
FOUR SCORE AND SEVEN YEARS AGO ...
```

The GOTO on line 100 causes the program to skip to line 130 without executing lines 110 and 120. Now enter

```
>>100
```

thereby knocking out statement 100. Enter RUN again and you should get

```
ONE MAN'S MEAT IS ANOTHER MAN'S POISON
```

The GOTO at line 120 skips directly to line 140. Line 130 is not executed. Although this is not a very useful way to write a program, i.e., selecting which part is to be executed by removing statements, it does illustrate the GOTO.

If a program containing GOTO statements is renumbered, BASIC adjusts the line numbers in the GOTO statements to point to the new line numbers. If a GOTO does not reference a valid line number when it is renumbered, the invalid reference is changed to a number greater than the last line number in the program.

At times, referring to line numbers with GOTO statements can prove to be a tedious exercise when there are many transfer points in a program. To simplify line references, BASIC allows us to label lines and to use the labels in GOTO statements. Here is the preceding example with a few labels added

```
100 GOTO ABEQUOTE
110 PRINT "ONE MAN'S MEAT IS ANOTHER MAN'S POISON"
120 GOTO QUIT
130 *ABEQUOTE:PRINT "FOUR SCORE AND SEVEN YEARS AGO ..."
140 *QUIT:END
```

ABEQUOTE and QUIT, on lines 130 and 140, respectively, are labels. Each is preceded by an asterisk(*) and followed by a colon (:). Each label is an alternate reference to the line on which it is found. The GOTO on line 100 now points to ABEQUOTE rather than 130. This ability to use labels makes programs far more readable than using line numbers. Labels are also called line names. Line names, or labels, are formed in the same way as numeric variable names: alphabetic and numeric characters and an apostrophe may be used; reserved names may not be used. For obvious reasons, line names must be unique within a program.

If a GOTO uses a line name that does not exist, which is possibly caused by a misspelling, the error will not be detected until BASIC attempts to execute the GOTO. A message is issued and the program stops.

MULTIPLE TRANSFERS—ON-GOTO

Another form of the GOTO is the ON-GOTO. This statement is similar to the GOTO, but it allows transfer of the program to another line, depending upon a value in the statement. Scratch your work area and enter

```
>>100 *START: PRINT "ENTER 1 OR 2";
>>110 INPUT NUMBER
>>120 ON NUMBER GOTO BRONE, BRTWO
```

```
>>130 GOTO START
>>140 *BRONE: PRINT "YOU ENTERED A ONE"
>>150 GOTO QUIT
>>160 *BRTWO: PRINT "YOU ENTERED A TWO"
>>170 *QUIT: END
```

This program uses a ON-GOTO at line 120. If NUMBER contains a 1, the program branches to the first line name in the line name list following the GOTO, BRONE. If NUMBER contains a 2, the program branches to the second line name in the list, BRTWO. If NUMBER is not an integer (a whole number), it is rounded to the nearest integer and used; the value stored in NUMBER is not changed. If NUMBER does not contain a value corresponding to a line name, for example, 3, then the next statement following the ON-GOTO is executed. Try RUN and respond to the prompt with a 1. We have

```
ENTER 1 OR 2?1
YOU ENTERED A ONE
```

Try RUN again and respond with a 2.

```
ENTER 1 OR 2?2
YOU ENTERED A TWO
```

Try once again and respond with a 3.

```
ENTER 1 OR 2?3
ENTER 1 OR 2?
```

In this case, NUMBER is 3 when we get to line 120, and there is no line to branch to for this value in the ON-GOTO so we fall through to the next line

which is a GOTO that transfers us to START. At START, the prompt is printed again.

The item following ON does not need to be a simple variable; any arithmetic expression is allowed. For example

```
ON VALUE+1 GOTO MOVE, SET, KEEP, FIX
ON INT(COS(A/2)*2)+1 GOTO SWIFT, ROBIN, FOX, WHEAT
```

LOGICAL CONTROL—IF

One of the most-used statements for changing the order of program execution is the IF statement. It also allows us to test expressions and relations, and then to act upon the result of the test. This statement introduces into our programs the ability to use a substantial amount of logic. Scratch your work area and enter the following program

```
>>100 PRINT "ENTER A NUMBER";
>>110 INPUT NUMBER
>>120 IF NUMBER=0 THEN GOTO ZERO
>>130 PRINT "THE NUMBER IS NOT ZERO"
>>140 GOTO QUIT
>>150 *ZERO: PRINT "THE NUMBER IS A ZERO"
>>160 *QUIT:  END
```

Before running this program, notice that line 120 contains an IF statement. It is read "If NUMBER equals 0 then go to line name ZERO". The equal sign does not mean assignment as in the case of the LET statement. It expresses a logical relationship. Now enter RUN and respond to the prompt with a 0.

```
ENTER A NUMBER?0
```

This produces

```
THE NUMBER IS A ZERO
```

The IF statement at line 120 tests NUMBER to see if it is zero. It is zero, so execution is transferred to line name ZERO. Now enter RUN again and respond to the prompt with 400.

```
ENTER A NUMBER?400
```

This produces the result

```
THE NUMBER IS NOT ZERO
```

Since NUMBER contains 400 when the IF is encountered, it is certainly not zero, and the statement following the THEN, GOTO ZERO, is not executed. Instead, the next statement in sequence, line 130, is executed.

The general form of the IF statement is as was just illustrated. There is a logical expression which is tested. It is followed by the word THEN, and finally the THEN is followed by a statement.

Many different types of logical expressions may be used in an IF. A logical expression is nothing more than a way of dealing with the relationship of two objects, as in

```
Is X greater than 100.0?
Is X not equal to Y?
```

X, Y, and 100 are the objects; the relationships desired are "greater" and "not equal." In IF statements, relationships are expressed by relational operators. The available operators are

Relational Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> or #	Not equal to

Some examples of their use in the IF are

```
IF NUMBER>400 THEN GOTO TOOBIG
IF VALUE=PI THEN VALUE=VALUE*PI
IF AGE>65 THEN NO'RETIRE=NO'RETIRE+1
IF COST<>0 THEN TOTAL=TOTAL+COST
IF YES THEN GOTO YESRESPONSE
IF SALES>=40000*MONTHS THEN GOTO SETSALES
```

The second to last example shows that a relationship is not needed. In this case, if the simple expression is 1 (true), then control is transferred to YESRESPONSE. This type of coding in an IF should probably be avoided for most beginning programmers. It may occur, however, as the result of a typing error, if the complete expression is not entered. In the last example, the logical expression contains an arithmetic expression, 40000*MONTHS. This is perfectly legal, and any valid arithmetic expression is allowed. Several of the examples show that the statement to be executed when the logical portion is true can be something other than a GOTO.

It is possible to have several statements following the THEN. The statement

```
IF AGE=20 THEN SUM=SUM+AGE : COUNT=COUNT+1
```

illustrates this point. The colon separates additional statements that are executed when the logical relationship is true.

The relational expression in an IF can be compounded with logical or Boolean operators of the sort: or, and, not, exclusive or. In a logical expression, the following Boolean operators are allowed:

```
AND     OR      NOT     XOR (exclusive or)
```

The "and" and "or" are the two most widely used. They are illustrated in the examples

```
IF AGE>30 AND WEIGHT<=200 THEN TOTWGT=TOTWGT+WEIGHT
IF COST>100.50 OR INVENTORY<=300 THEN GOTO REWORK
IF A=1 OR A=2 OR A=3 THEN VALUE=200
IF (C=100 OR B=50) AND M=15 THEN GOTO BADHEALTH
```

Logical operators separate relational expressions.

Of course, IF statements can be used with string variables and strings. As an illustration

```
100 PRINT "ENTER YOUR FIRST NAME"
110 INPUT NAME$
120 IF NAME$="JIM" THEN GOTO JIM
130 PRINT "YOUR NAME IS NOT JIM"
140 GOTO QUIT
150 *JIM:PRINT "HELLO, JIM"
160 *QUIT: END
```

Line 120 compares the string entered into NAME$ in the INPUT statement. If the name JIMMY is entered, the comparison between JIMMY and JIM made by the IF will not result in the strings being considered equal. In this case, the program would continue to line 130.

As an aside, a common mistake is to compare lowercase characters with uppercase characters, and expect an equal comparison. The string "Jimmy" is not the same as "JIMMY".

Another common mistake that is made when using the IF is in connection with comparing a string variable with a blank. For example, consider

```
IF NAME$=" " THEN GOTO NONAME
```

In the IF statement, the first characters in NAME$ up to the final non-null character are compared with " ". If NAME$ contains a single blank character, the comparison is true; otherwise, it is false. If NAME$ contains two or more blank characters only, the result is false. This can pose a small dilemma in some applications. A way out of the dilemma is to use NAME$(0,0) in the comparison.

LOAN REPAYMENT PROGRAM EXAMPLE

We have arrived at a point where it is useful to illustrate some of the concepts that we have learned with a specific example. Consider the problem of determining the monthly payment, M, on a loan with a certain principal, P, to be paid at a given interest rate, I, if the loan is to be paid in N years. Actually, we would like to produce a table showing the monthly payment for a loan with a variety of years for repayment and a variety of loan amounts. Suppose we want to determine the monthly rates for 15 through 25 years and for loans of $30000, $35000, $40000, $45000, and $50000. We will use 6.5% as an interest rate. The formula for calculating M is

```
M = (I/12) * Q/(Q-1) * P

where Q is  (1+(I/12))**N and N is in months.
```

Recall that ** is the notation for raising a number to a power.

The program that performs the required computations follows, along with a table of output values. From the table, we see that a loan of $30000 for 20 years at an interest rate of 6.5% requires monthly payments of $223.67. Let's look at the program.

Lines 100 through 130 set the interest rate variable, INTEREST, and print some title information. Since our principals range from 30000 to 50000 in steps of 5000, we use a FOR-NEXT loop at 150 to 170 to print the principal values as column headers. The variable PRINCIPAL is used to hold the values of the principal. Note that the PRINT at 180 causes a carriage return and line feed to terminate the title line produced by line 160. Lines 190 and 200 print additional title information. Lines 210 to 290 contain a loop on the variable NOMONTHS. NOMONTHS begins with 180 (15 years) and goes to 300 (25 years) in steps of 12 (1 year). When the number of years are needed in the output at line 240, NOMONTHS is divided by 12. Essentially, this loop controls the printing of rows. Lines 220 and 230 evaluate the formula for M. The 1200.0 appearing in these lines is 12*100, where dividing by 100 converts INTEREST to a number between 0.0 and 1.0. Since the formula is dependent upon the principal and we have not specified it, we compute M1. M1 is the portion of the formula for M without the multiplication by the principal. The loop from lines 250 to 270 causes PRINCIPAL to go from 30000 to 50000 in steps of 5000. Inside this loop, we know the current value of the principal; so, at line 260, PRINCIPAL is multiplied by M1 to give M. M is printed in this statement. Since the PRINCIPAL loop is interior to the NOMONTHS loop, we end line 260 with a semicolon to place all the principal calculations for a given NOMONTHS on the same output line. The PRINT on line 280 provides a carriage return and line feed to begin the next line of the table.

Monthly Payment Program

```
100     INTEREST=6.5
110     PRINT
120     PRINT USING"INTEREST: ##.##",INTEREST
130     PRINT"                              - PRINCIPAL -"
140     PRINT"      ";
150       FOR PRINCIPAL=30000.0 TO 50000.0 STEP 5000
160       PRINT USING"  ########",PRINCIPAL;
170       NEXT PRINCIPAL
180     PRINT
190     PRINT"YEARS     ";
200     PRINT"------    ------    ------    ------    ------"
210       FOR NOMONTHS=180 TO 300 STEP 12
220       Q=(1.0+(INTEREST/1200.0))**NOMONTHS
230       M1=(INTEREST/1200.0)*Q/(Q-1.0)
240       PRINT USING"###    ",NOMONTHS/12.0;
250         FOR PRINCIPAL=30000.0 TO 50000.0 STEP 5000
260         PRINT USING"  #####.##",M1*PRINCIPAL;
270         NEXT PRINCIPAL
280       PRINT
290       NEXT NOMONTHS
300     END
```

```
                  Program Output for
              Monthly Payments on a Loan

INTEREST:  6.50
                          - PRINCIPAL -
           30000     35000     40000     45000     50000
YEARS      ------    ------    ------    ------    ------
 15        261.33    304.89    348.44    392.00    435.55
 16        251.72    293.68    335.63    377.58    419.54
 17        243.34    283.89    324.45    365.00    405.56
 18        235.97    275.30    314.62    353.95    393.28
 19        229.46    267.70    305.94    344.19    382.43
 20        223.67    260.95    298.23    335.51    372.79
 21        218.51    254.93    291.35    327.76    364.18
 22        213.88    249.53    285.18    320.82    356.47
 23        209.72    244.67    279.63    314.58    349.53
 24        205.96    240.29    274.62    308.94    343.27
 25        202.56    236.32    270.08    303.84    337.60
```

PROGRAMMING SUGGESTIONS

If you have not had much experience with programming or have written only a few small programs from this book, you might want to pause here and try your hand at some personal applications using what you have learned up to this point. You might want to read ahead to the next chapter, but you should take some time to try some of the concepts you have learned in this chapter. You will find that you can build some very useful programs with what you have learned.

We'll leave the specific applications up to you, but here are some suggestions for items you might want to include in your programs:

1. Have the program interact with the user by asking his name and small amounts of information. Try to minimize the interaction, and keep it simple.
2. Echo back some of the interactive information.
3. Check some of the input data for consistency. For example, if a year is entered, is it greater than 1900 and less than 2000?
4. Get your application to work first and then think of ways to generalize it. For example, if you write a program to work with last year's gas bills, try to

rewrite it to work with any year's gas bills.

5. Use PRINT USING at least once in your program to place some results at specific parts of a line.
6. Do not go with grand ideas at first. Think small and get something running, and then add to it.

Review Questions and Exercises

1. Why does J cycle through the numbers 5, 4, and 3 in the statement

```
200 FOR J = 5 TO 3 STEP -1
```

2. Why is the following statement not likely to loop with J beginning at 1?

```
400 FOR J = I TO 10
```

3. Why do we prefer to a GOTO with a reference to a line name rather than to a line number? Why is the line name *STOP invalid?

4. What happens if K is 4 when the following statement is encountered?

```
900 ON K GOTO FIRST,SECOND
```

5. What is wrong with the IF statement

```
500 IF TIME = 1 OR 3 OR 5 OR 7 THEN GOTO ODD
```

6. Why does the compare in the following IF statement fail and cause the program to continue with the statement following the IF?

```
250 STATE$=" "
260 IF STATE$="    " THEN GOTO BLANK
270 PRINT STATE$
```

7. In the following program, why is the number of elements in the array MYDATA only 11, despite the fact that line 120 has reserved 51?

```
100 PRINT 300.4,297.6
110 GOTO SETUP
120 DIM MYDATA(50)
130 *SETUP:KSLOT=2000.0
140 PRINT "APRIL"
      ...
```

8. Why does the use of END in the following program line result in an incorrect statement?

```
150 IF END=45.0 THEN GOTO 400
```

9. When you are repeatedly modifying a program and using SAVE to save a the latest copy on disk, you must remember the name of the file each time. A way to avoid having to remember the name of the file, and to avoid worrying about mistyping the file name each time is to embed a few simple statements somewhere in the program like

```
500 LIST 510
510 SAVE "MYPROG.BAS
520 STOP
```

When you want to save, just GOTO line 500. Is there any danger in this technique of accidentally saving the file while the program is executing? Should any lines precede line 500? Does this technique suggest that you should be consistent in placing the SAVE code in each program? Would it be wise to start at 500 in one program, 900 in another, and 250 in another?

Summary

FOR-NEXT
: Allows a group of statements to be executed repeatedly for a specified number of times.
Testing of the loop takes place at the NEXT statement after the loop variable is incremented by the step size.
A step value of positive or negative size may be specified.
Loops may be nested one inside another.

Escape Key
: Used to terminate program execution when execution time is excessive.

GOTO
: Allows transfer of program control to a specified line.

ON-GOTO
: Permits transfer of control to one of many lines depending upon the value of an arithmetic expression.

Label or Line Name
: An alternate name for a line number.

IF
: Introduces logic into programs by permitting the execution of statements to be dependent upon the results of a logical expression.
Permits a number of relational operators (equal, greater than, etc.) and logical operators ("and", "or", etc.) in a logical expression.

CHAPTER 8

Debugging

It is possible to write some fairly complicated programs with the statements you have learned thus far. This increases the chances for errors of logic and other kinds to enter into the programs. Such errors will keep the programs from operating correctly. It is tempting to think that we are so good at writing programs that there won't be any problems. The modern Diogenes would find a search for the perfect programmer as difficult as the ancient Diogenes found his search for the honest man.

Figure 8.1 Help is on the way.

The process of uncovering programming errors and correcting them is called debugging. The errors uncovered are often referred to as bugs. Debugging is not always an easy task, and it may consume a great deal of the time it takes to complete a program successfully. In this chapter we will learn of some BASIC features that will help us to debug programs.

IMMEDIATE MODE

BASIC has an immediate mode of operation which permits us to execute statements independently of any program. Let's see how it works by scratching your work area and entering the error-free program

```
>>100 LET A=864
>>110 LET B=36
>>120 LET RESULT=A/B
>>130 PRINT "RESULT: ";RESULT
>>140 END
```

Use RUN to execute the program and to check the results against the expected output

```
RESULT: 24
```

Enter, without a line number

```
>>PRINT A,B
```

You should see

```
864           36
```

BASIC executes instructions without line numbers immediately; hence, the name immediate mode. This gives us the opportunity to use, in this case, the PRINT statement, to check to see if variables have the values we expect. We might have meant to set A to 0.864 instead of 864 and were surprised at the large result of 24. In a larger program it might not be so obvious that we set a variable to the wrong value.

Enter the following

```
>>PRINT A+B,A-B,"MORE RESULTS"
```

This should result in

```
        900            828            MORE RESULTS
```

You see that we are not restricted to simple PRINT statement.

In fact, any statement can be used in immediate mode. Try entering

```
        >>LET A=72
        >>GOTO 120
```

This produces

```
        RESULT:  2
```

If you print A, you will see that it is now 72, since we changed its value in the immediate mode. The GOTO is used here to recompute A/B at line 120.

STOPPING AND CONTINUING

Another useful statement for debugging is the STOP statement. It simply stops the execution of the program when it is encountered. Add the following line to our program and then enter RUN.

```
        >>105 STOP
```

This should produce

```
        ***105 Stop***
```

Entering

```
        >>PRINT A,B
```

produces

```
        864            0
```

This illustrates an important point. Entering RUN causes all program numeric variables to be initialized to 0 and string variables to be set to null strings. (An exception, which we will learn about in a later chapter, is COMMON variables that are used in chaining programs.) Since we stopped at line 105, B has not been assigned the value 36 at line 110. Hence, B is 0. Now enter

```
>>CON
```

This produces

```
RESULT: 24
```

CON is the continue command. It allows us to continue the program from the point where it stopped. This does not apply to the END statement. We cannot continue a program which has been stopped by an END. The CON is very valuable in debugging because it may be used in combination with STOP to continue a stopped program after displaying the contents of variables. The use of GOTO in immediate mode permits the same function as CON, but we must supply a line number or line label; this is not always conveniently done while trying to debug.

LISTING VARIABLES

Another feature of BASIC that is useful in debugging is the LVAR statement. It finds more use in the immediate mode than as an actual program statement. Using the previous example, now enter LVAR after the entry CON

```
>>LVAR
```

This should produce

```
A LFP 864.0
B LFP 36.0
Result LFP 24.0
```

LVAR, which stands for "list variables," produces a list of all variables, labels, function names, and procedure names. (Procedures will be discussed in a later chapter.) The current value of the variable is displayed along with its mode: LFP for long floating point; SFP for short floating point; and INT for integer. Labels are identified by LBL; other notations are used for functions and procedures. The list produced by LVAR may be placed on a file instead of being displayed at the terminal by using, for example, LVAR "PROGLIST.DBG". The string following LVAR specifies a file to which the results are written. The file can be printed for inspection using CDOS or CROMIX facilities.

While LVAR has some appeal, it is usually most useful in debugging programs that are fairly short. When the number of variables, labels, or names in a program becomes large, it is very difficult to weed out the results you really want to look at. Usually, printing just the variables you want displayed to the printer or terminal with PRINT is sufficient. LVAR is certainly useful for determining what variables exist in your program and what arithmetic modes (short, long, integer) are used by variables to store data. We will have more to say on mode in a later chapter.

STATEMENT TRACING

Another pair of statements, TRACE and NTRACE, are useful in debugging programs. They can be used equally well as program statements or in immediate mode. Usually, immediate mode is more suited for their use.

TRACE simply turns on the trace operation of BASIC, which lists each line number of each statement as it is executed. This can be very useful in tracing logic flow. Scratch the work area and enter

```
>>100 LET TOTAL=0
>>110 FOR NUMBER=85 TO 87
>>120 TOTAL=TOTAL+NUMBER
>>130 NEXT NUMBER
>>140 PRINT "SUM OF 85,86,87: ";TOTAL
>>150 END
```

Now enter TRACE and RUN as in

```
>>TRACE
>>RUN
```

This causes the program output

```
<100>
<110>
<120>
<130>
<110>
<120>
<130>
<110>
<120>
<130>
<140>
SUM OF 85,86,87 IS: 258
<150>
***150 End***
```

Each line number executed is shown between < and >. The FOR-NEXT loop is executed three times and the sequence <110> <120> <130> is seen three times in the output. The trace may be turned off by entering NTRACE, for no trace, as illustrated by

```
>>NTRACE
>>RUN
```

This produces

```
SUM OF 85,86,87 IS: 258
***150 End***
```

This is the normal output produced by BASIC; the trace is not operational because we entered NTRACE.

USING LIST AND ENTER TO CLEAN UP PROGRAMS

Although they are not specifically provided in BASIC for debugging purposes, two commands, LIST and ENTER, may sometimes help us overcome some idiosyncrasies of our programs caused by BASIC. LIST is the same LIST discussed earlier. Before stating how these commands are of help, let's see how they operate.

If we follow LIST with a string literal or string variable, the program in our work area will be listed to the file that is specified by the string. A file created in this manner and form cannot be loaded with LOAD. It can be brought back into the work area only with ENTER. SCR is usually used to clear the work area before using ENTER. A typical sequence of these commands, assuming there is something in our work area to begin with, is

```
LIST "MYPROG.LIS"
SCR
ENTER "MYPROG.LIS"
```

ENTER takes each line found in MYPROG.LIS that is created by the preceding LIST, and places it at the correct line number in the work area.

The question now is: What did LIST and ENTER just do for us? Actually, they did quite a bit. Using ENTER is nearly equivalent to having entered each line by hand directly into the work area; this has the effect of clearing away any loose ends in the program. The ENTER command causes BASIC to reanalyze each statement as it is entered, and in so doing, it sometimes clears up troublesome statements and programs.

How do statements and programs become troublesome? Usually, statements become troublesome when we modify them a great deal. The edit features, covered in an earlier chapter, occasionally introduce problems into statements. ENTER often clears them up. Programs become troublesome sometimes because we have modified them greatly and then saved them (using SAVE) repeatedly after modifying and executing them. Each time SAVE is used it saves everything: program statements, deleted line numbers, current variables and labels, and variables and labels previously deleted from our program. The use of LIST and ENTER as described sweeps away all of the excess debris from the work area and places the program statements in a fresh environment. Sometimes this can do wonders for straightening out balky and erratic-behaving programs. However, don't get too excited about this as a cure-all for program problems. The vast majority of bugs found in our programs will stem from other sources, which are usually of our own making, and these bugs will require some of the previously discussed features of BASIC to solve.

If you are interested in seeing how some debris accumulates in the work area, try the following exercise. Scratch your work area. Enter a LET at line 100, for example, LET A=1. Use RUN. An END statement is not needed in the work area. Use the LVAR command. Repeat this operation several times, changing the variable used in the LET. You will notice that each time LVAR is used, it lists all of the variables previously used. Use the LIST, SCR, and ENTER sequence discussed previously to clear up the one-line program work area and enter LVAR again. Are you left with just one variable when you use LVAR? You should be.

Before continuing, let us comment on what would happen if an ENTER is used without first scratching the work area. If statement lines already exist in the work area, they are not changed unless they correspond to lines in the program named in the ENTER. Later we will say more about this feature as a way of inserting or merging program segments into other programs. Note that the primary difference between LOAD and ENTER is that LOAD clears the work area before loading a program, and ENTER does not clear the work area.

USING LIST TO FIND A LINE NAME

In previous chapters, we mentioned that LIST is useful for listing program statements between two line numbers. It is also very useful for locating a line name. For example, suppose we have a line name of DO'TAXES in our program, and we want to locate the actual line where this line name occurs. To do this, we need only enter

```
LIST DO'TAXES
```

Try it out on a program that you have written that contains some line names.

Review Questions and Exercises

1. What is the difference between CON and RUN? When are CON and GOTO alike?

2. How do you turn off a trace?

3. Have a friend write a small workable program of 5 to 10 lines. Have him introduce a bug into the program. Debug the program using TRACE, CON, and LVAR. Are these commands helpful? Does inserting a PRINT at various points to print program variable values help? Is STOP a useful statement to insert somewhere in the program to help debug it?

4. If you write a program which has the last statement on line 4980, does adding the following statement help you to locate the last statement in your program when it is renumbered?

```
4990 *LAST:REM
```

   What happens when you enter LIST LAST after renumbering your program?

Summary

Immediate Mode
: BASIC statements may be executed immediately by entering a statement without a line number.
: Statements executed in immediate mode do not become part of the program.

Initialization
: RUN resets numeric values to zero and string variables to null strings.
: GOTO, in immediate mode, or CON do not re-initialize variables.

CON
: Continue command for continuing programs stopped by a STOP.

LVAR

Provides a way of listing the attributes and

Twelve data values are represented in the three DATA statements in lines 100 to 130. Line 140 contains a DIM statement which tells BASIC that a variable with the name of COST may contain up to 13 values. These values may be referenced by the notation COST(0), COST(1), COST(2), ..., COST(12), and they represent a table. We do not really need COST(0) and prefer to use COST(1) as the storage reference for the first data item. The numbers enclosed in parentheses following COST are called subscripts. Subscripts are used to refer to a specific element in the table. Lines 150 to 170 read each of the 12 data items into COST(1) to COST(12), using J as an index to subscript COST. Lines 190 to 210 loop through COST, using J as an index to subscript COST; each time COST(J) is added to the total, kept in TOTAL.

Although there are other ways of programming the above problem without using a table, the program does illustrate a way to use tabular structures.

The DIM statement serves as a way of telling BASIC how many elements to reserve in storage for a table or array. Any element may be used simply by referencing its subscript. In our example program, COST(J) references the j-th element in the COST array. Array elements may be used in any statement where a simple variable can be used. For example

```
PRINT COST(4),COST(3)
LET TAXES(J)=TAXES(J)+1
IF SALES(3)>200 THEN GOTO LARGE
```

are all legal statements. COST, TAXES, and SALES are arrays.

When reserving space with the DIM statement, or dimension statement, we must be careful not to overstate our needs. It is very easy to exceed the storage that is available to us by using too large a dimension. The number following the variable name in the DIM is referred to as the dimension of the array variable. Storage is shared by both program statements and variables, and we must reach a proper share for each. BASIC will tell us, with an error message issued at execution, when we do not have enough space for our program. We will address the problem of storage space in a later chapter.

dimension MYDATA for 21 values, ANGLES for 301, and COST for 49. BASIC uses what is called a zero origin for arrays, so we always get subscript zero. Hence, the numbers seen in a DIM statement must always be considered in light of this. If a dimension is not declared, then a variable with a subscript is automatically assumed to have a dimension of 10, which thereby reserves 11 elements. Incidentally, the reference COST, for example, is not the same as COST(0). That is, a variable and an array with the same name do not share the same storage locations. It is possible to have an array and a simple variable with the same name, although it is not good practice to do so.

It is worth noting that DIM is an executable statement. If it is not executed, it has no effect. If, for example, line 100 is not executed, then the arrays MYDATA, ANGLES, and COST all have the default dimension of 10.

TWO-DIMENSIONAL TABLES

Arrays such as the ones described above are called one-dimensional arrays. They essentially allow us to store a single column or row of data. Two-dimensional tabular data may be represented as well. For example, the table

Age	Weight
30	160
43	175
28	182
25	190

is of this type. It contains four rows and two columns, each corresponding to a dimension. Here is a program that reads this array data and prints it. Scratch your work area and enter it.

```
>>100 DATA 30,160
>>110 DATA 43,175
>>120 DATA 28,182
>>130 DATA 25,190
>>140 DIM AGEWGT(3,1)
>>150 FOR J=0 TO 3
>>160 FOR K=0 TO 1
>>170 READ AGEWGT(J,K)
>>180 PRINT USING "ROW # COLUMN # ",J,K;
>>190 PRINT USING "VALUE:###   ",AGEWGT(J,K);
```

```
>>200 NEXT K
>>210 PRINT
>>220 NEXT J
>>230 END
```

The data for the table are contained in the DATA statements. Line 140 declares the variable AGEWGT as a two-dimensional array with four rows and two columns. The subscripts for rows are 0 to 3, and for columns 0 to 1. It takes two FOR-NEXT loops to loop over two dimensions. The inner loop, from lines 160 to 200, controls the loop variable K, which is used to refer to column numbers 0 and 1. The outer loop, from lines 150 to 220, controls the loop variable J, which is used to refer to the row numbers 0, 1, 2, and 3. The data are read into the array AGEWGT at line 170, which is inside both loops. Line 180, inside both loops, prints AGEWGT and the row and column subscript used. The semicolon at the end of the PRINT USING statement prevents a carriage return from occurring. When a row of data has been read and printed on a line, the innermost loop, which controls the row variable K, is completed. Line 210 is then executed, causing a carriage return to occur. Hence, any output that follows is begun on the next line.

Now enter RUN and you should get

```
ROW 0 COLUMN 0 VALUE: 30   ROW 0 COLUMN 1 VALUE:160
ROW 1 COLUMN 0 VALUE: 43   ROW 1 COLUMN 1 VALUE:175
ROW 2 COLUMN 0 VALUE: 28   ROW 2 COLUMN 1 VALUE:182
ROW 3 COLUMN 0 VALUE: 25   ROW 3 COLUMN 1 VALUE:190
```

Statements similar to the loop and print control statements in the program are frequently found in programs that deal with two-dimensional tables. Thus it is worthwhile to study the program carefully to see what is happening.

Some people prefer to use 1 instead of 0 as the subscript in the first row or column of a table. This is easily accomplished by replacing lines with

```
140 DIM AGEWGT(4,2)
150 FOR J=1 TO 4
160 FOR K=1 TO 2
```

The new DIM reserves a 5 x 3 table. We do not use row 0 or column 0. For many users, the convenience of using subscript numbers that are familiar to them may offset the loss in storage that occurs because the entire table is not being used.

You might try modifying the program and experimenting with it. Keep a copy around by saving it with SAVE. For example, modify the PRINT USING to print just the value, without the row and column information. Try adding a line that prints the titles AGE and WEIGHT above each column.

HIGHER-DIMENSIONAL TABLES AND MAT

BASIC allows three-dimensional tables to be referenced as well. A DIM which reserves space for the three-dimensional table GEOGRAPH with 20 rows, 15 columns, and 5 planes is

```
DIM GEOGRAPH(19,14,4)
```

Higher-dimensional tables are not allowed.

Sometimes it is necessary to initialize all of the elements of an array or matrix to zero or some constant. This may be accomplished easily by the use of the MAT statement. For example

```
MAT COSTS=0
```

sets all of the elements of the array COSTS to zero. This is the only operation that is permitted on arrays. For the mathematicians, there are no matrix inversion or arithmetic matrix operations in Structured BASIC.

TABLES OF STRING DATA

There are times when we want to store string information in tables. BASIC does not allow us to do this in the same fashion as for numeric data. However, it is still possible to store string data in a tabular form.

For example, suppose we need to use the first three letters of each month as labels for a report that we want to produce, and for convenience we would like to keep these labels in a single string variable. Here is a sample program that prints the months across a print line

```
100 DATA "JAN","FEB","MAR","APR","MAY","JUN"
110 DATA "JUL","AUG","SEP","OCT","NOV","DEC"
120 DIM MONTH$(35)
130 FOR J=1 TO 12
140 READ MONTH$((J-1)*3,J*3-1)
150 NEXT J
160 FOR J=1 TO 12
170 PRINT USING "### ",MONTH$((J-1)*3,J*3-1);
180 NEXT J
190 PRINT
200 END
```

Since each of the 12 months is represented by three characters, we reserve 36 positions in the string variable MONTH$ at line 120. We will place all the labels in this variable. Lines 130 to 150 read each of the 12 labels into the next 3 consecutive positions of MONTH$ beginning with position 0. For example, when J is 1, "JAN" is placed in MONTH$((1-1)*3,1*3-1) or MONTH$(0,2). When J is 2, "FEB" is placed in MONTH$((2-1)*3,2*3-1) or MONTH$(3,5). The method of packing strings into substrings of a string variable is typical of the way related string data are kept in a single string variable.

This program gives us a way of generally accessing tabular data that is stored in a single string variable. When we are given the number of the month whose label is to be obtained, the substring notation, ((J-1)*3,J*3-1), which is used in lines 140 and 170, defines the substring positions needed. This idea is easily extended to other situations where the substrings are of a different length than 3.

HISTOGRAM EXAMPLE

One of the simpler and more useful programs that can be written in BASIC is a program to produce a histogram. The program listing is shown next along with its output. In this program, we produce a histogram for numbers between 0 and 65. Any number that is 65 or greater is simply counted and the count is printed. The idea presented here can be easily be expanded upon to produce histograms for other ranges of numbers.

To produce a histogram, we first must decide how we will collect the numbers. Each number between 0.0 and 65.0 is tabulated as belonging to an interval of length 5.0. For example, all numbers between 0.0 and 5.0 are counted in this interval. This is referred to as the first interval. The numbers from 5.0 to 10.0 are in the second interval. We will tabulate the number of values in each interval.

The mechanism for deciding which interval is used for a particular number is quite easy. We simply divide the number to be tabulated by 5.0 and find the resulting integer portion with the INT function. One is added to this result to give us an interval number. For example, consider the number 4.0. The integer portion of (4.0/5.0) is 0. Add one. The number belongs in the first interval. We use an array called URN to count the number of numbers in each interval. URN(1) refers to interval 1. URN(0) is not used.

With these comments in mind, look at the histogram program. The number of data points to be tabulated is read as NUMBER at line 160. The value 40 is read into NUMBER. Lines 170 through 220 form a loop which reads and tabulates each of the 40 numbers found in DATA statements in lines 120 to 150. VALUE contains the number to be tabulated. Line 190 computes the interval number according to the mechanism discussed previously. If VALUE is 65 or greater, then line 200 uses interval 14 to tabulate numbers 65 and above. Lines 230 to 300 print the bars for each of the 13 intervals. The statements from 260 to 280 print as many asterisks on a line as indicated by the contents of URN for each interval. Line 250 makes sure that an asterisk is not accidentally output for intervals whose count is 0.

Histogram Program

```
100   DIM URN(14)
110   DATA 40
120   DATA 44,18,22,15,17,36,29,40,66,29
130   DATA 18,80,22,17,44,18,14,80,5,43
140   DATA 15,71,92,74,15,24,18,39,55,17
150   DATA 7,91,18,24,47,49,77,61,15,81
160   READ NUMBER
170     FOR J=1 TO NUMBER
180     READ VALUE
190     INTERVAL=INT(VALUE/5)+1
200     IF VALUE>=65 THEN INTERVAL=14
210     URN(INTERVAL)=URN(INTERVAL)+1
```

```
 220      NEXT J
 230      FOR K=1 TO 13
 240      PRINT USING"###.# TO ###.# :",5.0*(K-1),5*K;
 250      IF URN(K)=0 THEN GOTO EMPTY
 260        FOR M=1 TO URN(K)
 270        PRINT"*";
 280        NEXT M
 290    *EMPTY : PRINT
 300      NEXT K
 310    PRINT : PRINT"NO. VALUES AT 65 OR BEYOND:  ";URN(14)
 320    END
```

Output from Histogram Program

```
  0.0 TO   5.0 :
  5.0 TO  10.0 :**
 10.0 TO  15.0 :*
 15.0 TO  20.0 :************
 20.0 TO  25.0 :****
 25.0 TO  30.0 :**
 30.0 TO  35.0 :
 35.0 TO  40.0 :**
 40.0 TO  45.0 :****
 45.0 TO  50.0 :**
 50.0 TO  55.0 :
 55.0 TO  60.0 :*
 60.0 TO  65.0 :*

NO. VALUES AT 65 OR BEYOND:   9
```

A PLOT PROGRAM

An interesting application involves plotting data in a scatter diagram fashion. Given a data point (3,4), we plot a point three units along an x-axis (horizontal axis) and four units along a y-axis (vertical axis). A typical example is a plot of weight versus height.

Next, we will discuss a fairly simple plot program which assumes that our x and y data fall in the range between 0.0 and 50.0. It is relatively easy to change the program to plot values over different ranges.

To produce a plot in the simplest fashion, assume that we have a grid of squares which will either be blank or contain a symbol that represents a plot point. Initially, the grid is filled with blanks. This grid is 35 squares wide and 20 squares high. Assuming that we want to plot 50 units on the horizontal axis, then each square width represents 50/35 or 1.42 units. Suppose the point we wish to plot has the coordinates (10,22). To find the distance along this axis for an x value of 10, we would use 15*(35/50) or 10.5. That is, 15 would be located in the square 10.5. For convenience, we would use 11. For the y value of 22, y would be located at square (20/50)*22 or 8.8, and we choose 9 for convenience. Our point at (15,22) would be placed at grid point (11,9).

The grid we would like to use is our terminal screen or printer. However, instead of plotting each point at the terminal, it is easier to put the point in a string in our program, which represents the grid, and then print the string when all the points are placed in it. The program that follows does just this by putting a "+" in the string variable GRID$ to represent a plot point. Although GRID$ contains a single one-dimensional string of characters, we can treat it as though it contains a row-and-column, two-dimensional string as mentioned earlier in this chapter. Since we tend to think of output in terms of lines and positions within a line, we can think of GRID$ in that way too. The first 35 elements contain the first line of the plot, and the second 35 elements contain the second line, etc. The first position in a line is the first element in a block of 35 elements. With this background in mind, let's look at the program more closely.

Line 100 reserves 701 elements for GRID$. The 700 reserves space for a 35*20 grid. The program does not use GRID$(0,0). In lines 110 through 130, GRID$ is set to blanks. The DATA statement at 140 indicates that there are 10 pairs of x and y coordinates given in the next DATA statements. (5,3) is the first pair. Line 170 reads the number of pairs into COUNT.

Lines 180 through 240 are the heart of the program. A pair is read into X and Y. Lines 200 and 210 compute the X position and Y line number that the point should occupy. At 220, P, the location of the point in GRID$ is computed.

Lines 250 through 360 place borders on the plot and print the data found in GRID$. Note that the computation of P is slightly different in line 290 than in line 220. The reason for the difference is that what is considered line 1 in GRID$ is really line 20 on the output screen.

Plot Program

```
100    DIM GRID$(700)
110      FOR J=1 TO 700
120      GRID$(J,J)=" "
130      NEXT J
140    DATA 10
150    DATA 5,3,18,25,7,4,9,11,22,30
160    DATA 6,8,45,45,30,40,40,47,38,44
170    READ COUNT
180      FOR J=1 TO COUNT
190      READ X,Y
200      XPOS=INT(35*X/50.0)+1
210      YLINE=INT(20*Y/50.0)+1
220      P=35*(YLINE-1)+XPOS
230      GRID$(P,P)="+"
240      NEXT J
250    PRINT"   Y"
260      FOR J=1 TO 20
270      PRINT"   |";
280        FOR K=1 TO 34
290        P=35*(20-J)+K
300        PRINT GRID$(P,P);
310        NEXT K
320      PRINT
330      NEXT J
340    PRINT"   |----------------------------------   X"
350    END
```

```
   Y
   |
   |                            +  +
   |                          +
   |                     +
   |
   |
   |
   |               +
   |
   |            +
   |
   |
   |
   |
   |
   |      +
   |    +
   |
   |   ++
   |
   |----------------------------------   X
```

Plot Program

AVOIDING STATEMENT REPETITION—GOSUB

Quite often in programming we find ourselves rewriting and repeating similar pieces of code or statements. This activity tends to waste our time and computer program storage. In order to overcome this difficulty, BASIC provides a way of defining code that is easily accessible from anywhere in a program and can be easily used over and over again. Such statements comprise what is known as a GOSUB, or GOSUB subroutine, and this code may be accessed with the GOSUB statement. Here is a simple example program which illustrates the concept

```
100 PRINT "QUARTERLY ELECTRICITY COST FOR 1981"
110 GOSUB HEADER
120 PRINT USING "###.## ",350.25,175.6,140.44,470.58
130 REM  OUTPUT GAS TITLE
140 PRINT "QUARTERLY GAS COSTS FOR 1981"
150 GOSUB HEADER
160 PRINT USING "###.## ",120.22,75.05,80.44,200.50
170 GOTO QUIT
180 *HEADER: PRINT "  Q1      Q2      Q3      Q4"
190 PRINT          "------ ------ ------ ------"
200 RETURN
210 *QUIT:END
```

This program prints the quarterly electricity and gas costs of a household in 1981. It is desirable to print a label above each cost printed that indicates the quarter to which it applies. To simplify the task of repeating the statement used to output the quarter labels, this task has been placed in a subroutine at lines 180 to 200. When the labels are needed, the GOSUB statement is used to go to the subroutine, execute the statements in the subroutine, and return to the next statement after the GOSUB statement. For example, when line 110 is executed, BASIC goes to the line name HEADER and continues executing there. Since HEADER corresponds to line 180, execution continues at 180. When line 200 is reached, the RETURN statement tells BASIC to return to the statement following the GOSUB, at line 120. The GOSUB is used again at line 150 to print the labels.

A GOSUB subroutine must be terminated by executing a RETURN statement. If a subroutine is executed that does not eventually execute a RETURN, BASIC will issue an error message and terminate your program. The opposite is true also. If a RETURN is executed without you having previously used a GOSUB statement, an error will occur.

In the previous program, if line 170 did not transfer control past the subroutine, the subroutine would accidentally be executed a third time. During the third execution, an error message would be generated at line 200 because the subroutine was not activated by a GOSUB statement.

In our previous illustration of a GOSUB, the subroutine was very small, and comprised of only a few lines. There are no limitations on the size of a GOSUB; some may be several hundred lines. We can pass different data into and out of a subroutine, as in the following example

```
100 LET INVALUE=10
110 GOSUB DEPTH
120 PRINT "SPEED: ";OUTVALUE*3.0
130 LET INVALUE=30
140 GOSUB DEPTH
150 PRINT "SPEED: ";OUTVALUE*3.0
160 GOTO QUIT
170 *DEPTH:IF INVALUE=10 THEN OUTVALUE=44.7
180 IF INVALUE<>10 THEN OUTVALUE=20.50
190 RETURN
200 *QUIT:END
```

INVALUE is set before executing the DEPTH subroutine. The subroutine checks the value of INVALUE and sets OUTVALUE as required. The main portion of the program uses OUTVALUE to print its results. This use of a GOSUB illustrates that parameters may be passed back and forth between the program using the GOSUB and the subroutine invoked by the GOSUB. When using a GOSUB in this manner, we must be very careful about changing variables inside the subroutine that are also used elsewhere in the program for other purposes. Such mistakes are quite often made.

GOSUB subroutines are very useful in many applications and should be considered when you find yourself repeating similar statements in your program. Here is a program that uses a subroutine called NSORT that sorts a one-dimensional array of numeric data into ascending order, i.e., from low to high values.

```
100    DATA 35,45,22,15,18,20,30,10,60,20,5,4,1,3,2
110    DIM NDATA(15)
120      FOR J=1 TO 15
130      READ NDATA(J)
140      NEXT J
150    U1=15
160    GOSUB NSORT
170      FOR J=1 TO 15
180      PRINT USING"## ",NDATA(J);
190      NEXT J
200    PRINT
210    STOP
500    REM SORTS NUMERIC ONE-DIMENSIONAL ARRAYS:NUMSORT
510    REM INPUT ARRAY IS NDATA. NDATA(0) IS NOT USED
520    REM U1 IS THE NUMBER OF ELEMENTS IN NDATA
530    REM U1, W1, W2, W3, W4 AND W ARE LOCAL VARIABLES
540   *NSORT : W1=1
550   *LESST1 : W1=W1+W1
560    W4=W1-1
570    IF W1>U1 THEN GOTO NEXIT
580    GOTO LESST1
590   *NEXIT : W4=INT(W4/2)
600   *DOWHIL : IF W4=0 THEN GOTO NQUIT
610    W3=U1-W4
620      FOR W2=1 TO W3
630        FOR W6=1 TO W2 STEP W4
640        W1=W2-W6+1
650        IF NDATA(W1+W4)>=NDATA(W1) THEN GOTO BRANCH
660        W=NDATA(W1)
670        NDATA(W1)=NDATA(W1+W4)
680        NDATA(W1+W4)=W
690        NEXT W6
700    *BRANCH : NEXT W2
710    W4=INT(W4/2)
720    GOTO DOWHIL
730   *NQUIT : RETURN
740    END
```

Lines 100 to 210 contain the program which uses the NSORT subroutine found in lines 500 to 740. Fifteen data values are read from the DATA statement into the array NDATA. After NSORT is used at line 160, the sorted values are found in NDATA and output as

```
 1  2  3  4  5 10 15 18 20 20 22 30 35 45 60
```

We will not explain the inner workings of NSORT. It is based on an algorithm by D. L. Shell, "Highspeed Sorting Procedures," COMMUNICATIONS OF THE ACM, Volume 2, No. 7 (1959), pp. 30-32. If you use it in your own programming, note that U1 is set to the number of values in the array and that NDATA(0) is not used.

Figure 9.1 Smooth sailing now.

Review Questions and Exercises

1. If DIM ABC(20) is found in a program, why does the ABC array have 21 elements?

2. Are COST and COST(0) the same variables? Why not?

3. Why does DIM SALES(10,10) reserve 121 elements?

4. Why is DIM MONTHS$(12,5) invalid?

5. Even though BASIC doesn't allow string arrays, is it possible to use string variables to store array-like data?

6. What statement must be used to exit from a subroutine?

7. Is it a good idea to put a STOP statement before a subroutine? Is a GOTO placed before a subroutine that branches around the subroutine useful? What happens if you accidentally enter a subroutine without using the GOSUB statement?

Summary

Array or Matrix Variable
- A variable which is used to contain data represented as a table.
- One-, two- or three-dimensional arrays are allowed.
- An element of an array variable is referenced by using the subscripts corresponding to the element.
- Arrays may be defined only for numeric data.

DIM
- Used to reserve storage for an array.
- Always reserves space for zero-th entry. That is, a zero origin is assumed for all dimensions.
- All one-dimensional arrays have a default dimension of eleven elements.

MAT
- Sets all the elements of a matrix or array to a constant.

GOSUB subroutine
: A set of program statements that may be executed repeatedly from other parts of a program.

GOSUB
: Transfers control to the GOSUB subroutine named in the statement.

RETURN
: Causes a GOSUB subroutine to return to the statement following the previously executed GOSUB statement.

CHAPTER 10
Odds and Ends

This chapter marks a point which is approximately two-thirds of the way through our learning about Structured BASIC and programming. The statements that we have learned already will allow us to write some very sophisticated and complex programs. Before extending our knowledge of BASIC further, we will try to bring together some odds and ends that will increase our ability to solve application problems with BASIC. These items, when added to what we have previously learned, represent a good fundamental knowledge of BASIC.

Figure 10.1 Not quite this odd.

SETTING SYSTEM PARAMETERS

BASIC allows us to have some control over system parameter settings. For example, when we use LIST to list lines in the work area, the statement and variable names are capitalized and the remaining portion of the names are in lowercase. We may change the corresponding system parameter so that everything is capitalized. System parameters are identified by numbers. Parameter 6 is associated with the uppercase control just mentioned. The parameter may be set to normal capitalization, or to complete capitalization. The statement which does this for us is SET. An example of its use is

```
150 SET 6,1
```

This example tells BASIC to set parameter 6 to a value of 1, which indicates complete capitalization. A value of 0 is used to set normal capitalization.

Here is a list of the more commonly used parameters which may meaningfully be set.

Parameter	Description and Values	Examples	Meaning
0	Controls page width of output statements. Default value is 80. Negative value inhibits carriage return and feed.	SET 0,132 SET 0,-1	132 columns Infinite--no carriage return
1	Tabbing field width used by the PRINT to separate fields. A field width of 20 is normal. Negative value or zero inhibits tabbing.	SET 1,20 SET 1,-1	20 cols/fld No tabs between flds
4	Positions current print column.	SET 4,70	Move next print to 70
5	Controls how long an INPUT statement will wait for input. Normally this feature is off, i.e., set to 0. Any value turns it on. Units are 10 per second.	SET 5,0 SET 5,20	Indefinite wait Wait 2 sec
6	Controls capitalization of names when a LIST is used. A 0 denotes first character of a name capitalization. A 1 denotes all characters are capitalized upon output.	SET 6,1 SET 6,0	All caps First cap
7	Numeric constants are displayed so that units of 1000 are evident.	SET 7,0 SET 7,1	No spaces 1 space btwn 1000s

Parameter	Description and Values	Examples	Meaning
8	When using LIST, indentation of such statements as the FOR-NEXT is controlled by this parameter. Normally it is SET to 0 indicating an indentation of 2 positions. A minus 2 provides no indentation. Values range from -2 to 5.	SET 8,5 SET 8,-1 SET 8,-2	Indent 7 Indent 1 Indent 0
9	Whether quotation marks input in response to an INPUT statement are to be ignored is controlled by this parameter. Normally, it is set to 0 to ignore them. Accept if set to 1.	SET 9,0 SET 9,1	Ignore " Obtain "
10	Set partition number. Also see USE in Chapter 14.	SET 10,6	Set to partition 6

Parameter 0 is often used in applications that wish to output more than 80 characters per line. Applications such as plotting, that do not want carriage returns inserted when a right margin is encountered, use this parameter with a negative setting.

This list does not contain the complete set of system parameters. It represents only the list of parameters that may meaningfully be set. All system parameters may be set by SET. Other parameters will be discussed shortly.

OBTAINING SYSTEM PARAMETERS

In addition to parameters which may be set and controlled by us, values of these parameters may be read by a program. There are only a few that it makes sense to try and obtain during the execution of a program. A particularly important one is parameter 3, which contains the error number of the last error produced during execution of a program. When BASIC detects an execution error, it always places a unique error number for the error in this parameter. All errors detected by BASIC and their error numbers may be found in Appendix C. The value of any parameter may be examined by use of the SYS function. For example, to obtain the value of parameter 3 we might use

```
ERRORNO=SYS(3)
```

The number enclosed in parentheses is the parameter number. Here is a list of some of the more important parameters which may be examined with SYS.

Parameter	Description	Example
2	Character last printed	A$=SYS(2)
3	Error number of last error detected during program execution	ERNO=SYS(3)
4	Current print column	PCOL=SYS(4)
5	Time remaining for INPUT	ITIME=SYS(5)
10	Current partition	PARNO=SYS(10)

When using SYS(2), the result is a string containing one character. Here is an example using SYS(2)

```
LAST$=SYS(2)
```

Again, all system parameters may be examined with SYS; however, there are very few applications where it makes sense to examine the uppercase listing parameter with SYS(6). SYS(3) is by far the most frequently used SYS function.

Several other parameters which have meaning with the keyed sequential access method (KSAM) facilities of BASIC may be set and read. See the 32K Structured BASIC Instruction Manual for a discussion of this topic.

RESPONDING TO EXECUTION ERRORS

As we have just seen in our discussion of system parameters, BASIC reacts to errors caused during the execution of a program by putting an error number in system parameter 3. It would be useful in some circumstances if the program could continue to operate and determine if the error is serious enough to stop. BASIC provides a statement for this purpose: ON ERROR. It has three forms which are illustrated here

```
ON ERROR STOP
ON ERROR GOTO ERRORCHK
ON ERROR GOSUB LOOKERROR
```

In the first case, the statement tells BASIC to stop whenever an error occurs and issue a message. This is the usual reaction by BASIC to an error. In the second case, BASIC is told to go to the statement at a line name of ERRORCHK and continue there. The final case is like the second one, but a GOSUB with the name LOOKERROR is used. Return from the GOSUB is made to the point following the statement that caused the error.

Not all errors can be controlled by use of the ON ERROR statement. Certain errors are considered fatal and cause BASIC to stop, regardless of any ON ERROR setting. An example of a fatal error is a transfer to a line number that does not exist. Only non-fatal errors react to the ON ERROR statement. An example of a non-fatal error is an arithmetic overflow or an arithmetic expression that produces a number which is larger than allowed by BASIC. Non-fatal errors are any errors which have an error number of 128 or greater.

Let's look at an example of how ON ERROR is used. Scratch your work area and enter

```
>>100 LET BAD=4/ANUMBER
>>110 PRINT "BAD: ";BAD
>>120 ON ERROR GOSUB RECOVER
>>130 LET NOTGOOD=22/ANUMBER
>>140 PRINT "NOTGOOD: ";NOTGOOD
>>150 GOTO QUIT
>>160 *RECOVER:PRINT "YOU JUST GOOFED"
>>170 RETURN
>>180 *QUIT:END
```

Both lines 100 and 130 contain invalid computations. They contain expressions which attempt to divide a number by 0. ANUMBER is 0 since it was not initialized. Enter RUN and you should get as output

```
Error 250 at line 100 -- Overflow/underflow
```

Since we have not yet told BASIC to do anything special about errors, it proceeds normally and stops with an error message. Now enter the continue command, CON, and this will produce

```
BAD: 0
YOU JUST GOOFED
NOTGOOD: 0
***180*** End
```

Line 110 printed the first line of output. We see that the variable BAD was assigned a value of 0. The next line, 120, tells BASIC to go to the GOSUB subroutine RECOVER when a non-fatal error occurs. Line 130 contains a non-fatal error, division by 0; BASIC jumps to RECOVER and prints the second line of output. The RETURN takes us back to line 140, which is the line following the error. The value of NOTGOOD is printed at line 140 and the program terminates successfully.

The process of treating errors as was just described is often referred to as trapping errors. The ON ERROR is used to set or not set the trap. It may not be necessary to have a trap in effect for all parts of a program. It is turned off by using ON ERROR STOP.

There is nothing to prevent us from doing whatever we want when we have trapped an error. We might issue our error message as in the example or ignore the message. Other possibilities are to examine system parameter 3 with SYS(3) and react to the error type or to correct an invalid variable value and continue processing. The last possibility is aided by the RETRY statement.

Appendix C contains a list of all fatal and non-fatal error messages used by BASIC. Take a look at the appendix to get an idea of the types of errors for which BASIC issues messages. Note in particular the non-fatal errors 200 through 208, and 250. Error message numbers 128 through 142 pertain to files, and they will be useful later when we learn more about file usage.

You might try using ON ERROR to recover from error 204 by constructing a simple program containing INPUT. You should use ON ERROR to trap and respond to the error. Issue your own message, and direct the user to input the correct number of items. Try to input too many data items to INPUT as a test of your program.

RECOVERING FROM ERRORS WITH A GOSUB

RETRY is used in place of a RETURN in a GOSUB when we want to transfer back to the statement causing the error. Presumably we will be able to correct the cause of the error in the GOSUB subroutine before attempting the RETRY, or we will find ourselves in an endless loop between the statement containing the error and the subroutine. In the previous example, we might put statements in the subroutine that ask for the value of ANUMBER so that a retry would be successful; also, we would replace RETURN with RETRY.

CONTROLLING THE ESCAPE KEY

There is another type of ON statement which is useful in applications where we want the user of a program to signal the program in some way by hitting the escape key. It is the ON ESC statement. Like ON ERROR, it comes in three varieties as seen in the following examples

```
ON ESC STOP
ON ESC GOTO EXAMINE
ON ESC GOSUB NEWMODE
```

If the first example is in effect, hitting the escape key will cause BASIC to stop at whatever line it is executing. The other two examples cause BASIC to transfer to a line or GOSUB subroutine. When the program transfers, it may, for example, ask the user for some specific item.

The normal reaction of BASIC to hitting the escape key may be disabled by use of the NOESC statement. When NOESC is in effect, BASIC ignores the hitting of the escape key. This may be of interest in applications which do not stop when the user hits the escape key. The normal response may be restored by use of the ESC statement. These two statements have the simple forms

```
NOESC
ESC
```

The NOESC statement is usually not used when a program is being developed or tested. The reason is that errors may exist that cause the program to go into an endless loop and the escape key could not then be used to stop the program. The reset switch on your computer would be used to stop everything in order to restart. Some care should be exercised in using NOESC.

PROTECTING PROGRAM LINES

There are some programs or portions of programs which a program developer may find valuable and not want to disclose to other users of the program. BASIC contains a command, NOLIST, which permits you selectively to designate lines which no one can list with LIST. Like LIST itself, NOLIST has several forms which are illustrated in the following examples

```
NOLIST
NOLIST 9000
NOLIST 4500,6000
NOLIST 500,
```

The first example makes it impossible to list any program lines. Example 2 makes it impossible to list line 9000, and example 3 makes it impossible to list lines 4500 through 6000. Finally, the last example prevents 500 to the end of the program from being listed. Caution should be observed in the use of this command. There is no way to restore the ability to list lines which have been designated as unlistable with NOLIST. Keep a separate program file containing completely listable lines, if you use NOLIST.

PROTECTING INPUT DATA

Some applications require that a user enter a password or identification that allows him access to the application. In order to prevent someone else from seeing a private code when it is entered, it is possible to have BASIC not echo each character as it is entered. When you enter a character at your terminal or console, it is BASIC that displays each character back to the terminal so that you can see what you have entered. The NOECHO statement turns off the facility which echos characters to the terminal. ECHO restores the facility to the normal mode. Here is an example program that shows how ECHO and NOECHO work. Scratch your work area and enter

```
>>100 NOECHO
>>110 PRINT "ENTER YOUR SECRET CODE"
>>120 INPUT CODE
>>130 ECHO
>>140 PRINT "ENTER YOUR NAME"
>>150 INPUT NAME$
>>160 PRINT "THANK YOU, ";NAME$;
>>170 PRINT ".  YOUR SECRET CODE IS ";CODE
>>180 END
```

Enter RUN. Responding with 500 as the secret code where shown by the arrow, and the name "WILLIAM" as in the following sequence produces

```
ENTER YOUR SECRET CODE
?                                      <--- enter 500
ENTER YOUR NAME
?WILLIAM
THANK YOU, WILLIAM. YOUR SECRET CODE IS 500
```

Our response to the first question mark that is produced by the INPUT at line 120 is not shown because NOECHO is in effect. The response, WILLIAM, to the next INPUT is echoed because line 130 turns the echo facility to its normal echo response.

SUBSTRINGS REVISITED

There are several different notational forms of referencing a substring. The method explained in an earlier chapter is the simplest form to understand and to use. Use both the beginning and ending position in a substring reference. We recommend using this approach whenever possible. There are times, however, when you will stumble into using other forms, and you may want to know what is happening. We will attempt to explain and simplify what is a very difficult set of notational and interpretational form differences for referencing substrings. With a little patience, you can understand and use the notation effectively in your programs.

Probably the best place to start is with the notion of a string variable's length and dimension. By dimension, we mean the current dimension of the string variable as established by a DIM statement. If a DIM statement was not used, the DIM of a string variable is 10. The length of a string variable is its length as determined by the Structured BASIC LEN function. The length is determined by counting characters beginning with the first position (0) and continuing until trailing null characters are encountered. A null character in the middle of a string variable is not a trailing null character. Now, let's examine a common reference form.

As we are about to discover, a simple reference to a string variable, such as NAME$, is actually a substring reference, even though the notation seems to suggest that it references the entire string in the variable. BASIC interprets such a form as referencing the first position of the variable (position 0) to some other final position in the variable. The final position is the length -1 (determined by LEN) or the dimension of the variable. (You don't have to apply LEN; BASIC does this for you as it tries to interpret the string variable reference.) The method used to find the last position is determined by whether the variable is used in an input or output statement. INPUT, READ, and PRINT are examples of such statements. In a later chapter, we will find PUT and GET as further examples. The LET statement is both an input and output statement. A variable on the left of the assignment is considered input, and one on the right is considered output. Here are two rules BASIC uses to determine the final position in the simple string variable reference

Rules for Final Position
in a
Simple String Variable Reference

```
Input:  dim(stvar$)
Output: LEN(stvar$)-1
```

The dim indicates that the current DIM is used by BASIC to determine the final position. As examples, consider the two statements

```
INPUT NAME$
PRINT CITY$
```

According to our rules, any data input into NAME$ in the INPUT statement is interpreted as a reference to position 0 through the last position of the string, 10, assuming that NAME$ was not used in a DIM. Observing the rule concerning output, the PRINT statement reference to CITY$ is interpreted as a reference to positions 0 through the last non-null character in CITY$ or the position determined by LEN(NAME$)-1.

With this simple form behind us, all other forms add refinements to it. There are really only two other notational forms for referencing a substring

```
STVAR$(e1)
STVAR$(e1,e2)
```

The symbols e1 and e2 are just arithmetic expressions. In their simplest form, e1 and e2 are just numbers.

When e1 is used, it refers to the first position of the substring. For example, the 6 in NAME$(6) indicates that the substring begins in position 6 of NAME$. When e2 is omitted, the last position is determined by one of the two rules stated for the simple reference. That is, e2 is determined from a consideration of the string variable's length (as determined by LEN) and its DIM. When e2 is used, its meaning is determined by whether e2 is positive or negative. If e2 is positive, it is the final position in the string. However, if e2 is negative, it is the number of positions, beginning with e1, in the substring. As examples

```
NAME$(4,6)  references positions 4 through 6
NAME$(8,-3) references positions 8 through 10
```

Although it is tempting to use the negative e2 form, this is an unusual feature that is not commonly found in most languages, and should probably be avoided.

What happens if we make a mistake in specifying either e1 or e2? What if e1 is negative? BASIC simply ignores both e1 and e2. For example, NAME$(-2,5) is treated as NAME$, but with one extremely important exception. If the form like NAME$(e1) is used and e1 is negative, the last position is always considered to be determined by the dimension regardless of whether the variable is used as input or output. This is very important in file statements because, as we will learn later, we often need a simple way of making sure all characters in a string variable are read and written.

What if e2 is positive, but refers to a position preceding e1, as in NAME$(4,2)? What happens if a reference such as NAME$(6,-3000) is given, where NAME$ has a dimension of 10. In both cases, since e2 is not meaningful, BASIC determines the final position as it did for the simple reference NAME$; nevertheless, the first position is taken as e1. What happens if e1 or e2 are positive and are larger than the dimension of the string variable? When you try to execute a statement with such a reference, BASIC will give you an error.

In summary, BASIC has a way of interpreting all valid notational substring references; positional references beyond the dimension of the string variable are illegal. Positional references before position 0 are taken as references to position 0. When an explicit reference is not made to the last position of a substring, BASIC determines the position from whether the statement in which the reference made is an input or an output statement.

NUMBERS AND STORAGE—THE LONG AND SHORT OF IT

If we become sophisticated enough in our programming needs, eventually we will become concerned about how much of our program is taken up by numeric constants and variables. After all, these items must be kept some place in the memory of the computer so that we can use them. They occupy memory, or storage, as we will call it for now. For most simple programs of a hundred or so lines, storage of these items will not be of much concern and we can ignore it. However, we will eventually be forced to deal with storage considerations, so let's learn some of the preliminaries. You may wish to read the following description lightly until you have mastered more of BASIC and have begun writing larger programs.

To start with, floating point numbers are broken into two forms: short and long floating point numbers. In the short form, only 6 significant digits are retained. In the long form, 14 digits are retained. This is the default floating point form. Short floating numbers require less storage than long floating numbers. In fact, a short floating number requires exactly one half the storage of a long floating number. (The fact that 6 is not one half of 14 may bother you. Two additional digits for the exponent are stored with the numbers. Hence, the numbers require 8 and 16 digits for storage which is a ratio of 1 to 2.) Integer numbers require less storage than short floating numbers. Floating numbers must keep some information about the size, in powers of 10, of the number they represent, whereas integer numbers do not. This additional information requires extra storage over an integer number.

OBSERVING THE EFFECT OF BASIC ON NUMBERS

The representation of a number, or constant, in a program is important to how it is stored internally in BASIC. BASIC attempts to alleviate this consideration for the programmer by storing numbers in their most compact form when possible.

As a brief illustration of how BASIC tries to use the minimum storage for data, consider the following program which was entered exactly as shown

```
100 LET A=4000
110 LET B=45000
120 LET C=12.34
130 PRINT A,B
140 END
```

Applying LIST to this program produces

```
100 Let A=4000
110 Let B=45000.0
120 Let C=12.34
130 Print A,B,C
140 End
```

We see that something has changed about the 45000 in line 110. It now contains a decimal point.

However, 4000 and 12.34 in the other two LET statements have not changed. BASIC keeps any number with a decimal as a floating number. It represents any integer number that is not greater than 10000 in magnitude as an integer. Hence, the listing indicates that 45000 is represented as a floating number; 4000 is an integer, so no decimal point is added. Since 12.34 has a decimal point, it is represented as a floating number. Both 12.34 and 45000 are considered short floating numbers because they do not contain more than 6 significant digits.

VARIABLE STORAGE

BASIC extends the concept of floating and integer numbers to variables. BASIC makes it possible to specify that a variable must keep data in a certain representation. For example, we might state that a variable should store only integer numbers, thereby requiring less space. This ability gives us control over storage size. The amount of storage taken by a variable is measured in bytes. A byte is a fundamental unit of data storage. The amount of storage occupied by variables of the three types is

Type	Storage
Long Float	8 bytes
Short Float	4 bytes
Integer	2 bytes

BASIC has several statements which allow us to control the representation and storage of data.

Although we will postpone the discussion of specifying variables as representing long, short, or integer storage to a later chapter, let's emphasize something that still may not be clear. First, the variable A in our example program in the preceding section occupies storage for a long floating number. Numeric variables default to long floating storage. Secondly, in our example program, 4000 will be kept internally in an area of storage for constants as an integer. When 4000 is placed into A during the execution of line 100, the variable A still occupies the space for a long floating number despite the fact that an integer constant is placed in it. In fact, the integer is first converted from its integer representation to its floating form when it is stored in A. The point we wish to emphasize is that constants occupy storage according to a set of rules concerning how the constants are written in the program, and variables occupy storage according to a different set of rules. The rules which apply to variables will be discussed in a later chapter. The BASIC mode statements also influence the rules for both constants and variables, and these statements will be discussed later.

MISCELLANEOUS SYSTEM FACILITIES

For BASIC users who are familiar with assembly language and the inner workings of the computer, BASIC provides several statements and functions which may be of interest. Here is a list with a brief description of their purpose.

Function	Description
ADR	Returns the address of a variable.
INP	Reads data from an input port.
PEEK	Reads the contents of a memory location.
TYPE	Determines whether an arithmetic expression is short float, long float, or integer.
USR	Executes an assembly language subroutine.

Statement	Description
OUT	Writes to an output port
POKE	Stores into a memory location

These functions and statements are described in more detail in the Structured BASIC Instruction Manual. KSAM numeric sorting conversion functions are described in more detail in the 32K Structured BASIC Manual. Some examples of their use follow

```
MEMLOC=PEEK(%FAB4%)
POKE %C004%,128
DUMMY=USR(%C100%,Z,1,18)
OUT 1,34
CHARHEX=INP(3)
ADDRESS'OF'Z=ADR(Z)
VALTYPE=TYPE(32.0/2)
VALTYPE=TYPE(ABC)
```

PEEK reads the contents of the memory location at the hexadecimal number %FAB4% into MEMLOC. POKE places the value 128 (decimal) into memory location %C00$%. USR executes an assembly language program at location %C100%, and passes it the parameters Z, 1, and 18. OUT sends the value 34 to port 1, and INP reads a character from port 3. ADR returns a two-byte integer corresponding to the address of the variable Z. TYPE is discussed in more detail in the next paragraph.

The TYPE function is particularly useful in determining the mode of an expression or simple variable. It returns the following values

Value	Expression Type
1	Integer
2	Short Floating
4	Long Floating

Try some of the following uses of TYPE to see what you get

```
LET ZIP=TYPE(3)
LET ZIP=TYPE(5/2)
LET ZIP=TYPE(22.0/4)
LET ZIP=TYPE(2*A)
LET ZIP=TYPE(44.444)
LET ZIP=TYPE(A+B/3)
```

A FEW MORE ODDS AND ENDS

As indicated in Chapter 7, a colon (:) can be used with the IF statement to place multiple statements after the IF that are executed when the logical expression in the IF is true. The colon has a similar use elsewhere in BASIC. Multiple statements may be placed on any line if they are separated by a colon. For example

```
150 PRINT : PRINT : PRINT "MARCH DATA"
```

In this example, the three PRINT statements are executed from left to right when line 150 is encountered. Although it is appealing to use several statements on a single line, we suggest that you minimize such use. Your programs will generally be easier to read and to debug if you use one statement per line.

It is possible to concatenate strings with the + operator in the LET statement. By concatenate, we mean the ability to combine several strings into a larger string. For example

```
250 LET CITY$="BOSTON"
260 LET STATE$="MASS."
270 LET TITLE$=CITY$+", "+STATE$
```

Assuming TITLE$ has a dimension great enough to hold the result, TITLE$ will contain the concatenation of CITY$, ", ", and STATE$

```
BOSTON, MASS.
```

You cannot use the concatenation operation to concatenate a variable to itself because an incorrect result will be obtained. The following will result in "ABC" being placed in A$ regardless of the the previous contents of A$

```
400 A$=A$+"ABC"
```

The correct result can be obtained by using

```
400 B$=A$+"ABC"
410 A$=B$
```

Review Questions and Exercises

1. What does SET 0,-1 do? What does SYS(3) refer to? If SYS(3) returns a value of 10, what error has occurred according to the descriptions of errors given in the appendix?

2. If you have not used ON ERROR, what is the ON ERROR response given by BASIC?

3. How does a fatal error differ from a non-fatal error? What is the range of error numbers for a fatal error?

4. If NAME$ contains "ABCDEFG", why does NAME$(4) reference "EFG"?

5. How many significant digits are retained by BASIC for short and long variables? How many bytes of storage do such variables require? Does an integer variable use only one byte of storage?

6. What value does TYPE(3.4442) return? What does it mean?

Summary

SET
: Statement which sets system parameters that control page width, field tabbing, etc.

SYS
: Function which returns the current settings of system parameters.

Error Numbers
: Numbers corresponding to BASIC error messages.
: SYS(3) produces the error number of the last error detected during program execution.

Fatal Error
: An error which causes BASIC to stop execution.

Non-Fatal Error
: An error which causes BASIC to stop executing unless an ON ERROR statement is in effect which specifies otherwise.

ON ERROR
Allows transfer to program statements which examine error causes and react as necessary.

RETRY
A special return from a GOSUB that returns to a statement which was in error when the GOSUB was activated.

ON ESC
Allows transfer to program statements which react to the activation of the escape key.

NOESC-ESC
Disable and restore the normal response of BASIC to the activation of the escape key.

NOLIST
Command which permits lines to be selectively specified as unlistable.

NOECHO-ECHO
Disable and restore the normal response of BASIC to echo characters back to the terminal entered in response to an INPUT statement.

ADR, INP, PEEK, TYPE
Special functions for detailed system work.

OUT, POKE
Special statements for detailed system work.
Numbers are regarded as integer, short floating, long floating, or hexadecimal by BASIC and each type occupies a different amount of storage.

Short, Long, and Integer Variable Storage
Numeric variables may have storage attributes so that they occupy different amounts of storage.

Subscripts
String variables may be subscripted in a variety of ways. The safest approach is to use ABC$(beginpos,endpos).

CHAPTER 11
Files

Until now, our concerns about storage have been with storage in the memory of the computer where our programs reside during execution. Variables and program statements occupy this storage when a program is executed in the work area. There is another type of storage which lies outside of memory. It is the storage that is available on your floppy disk or hard disk. As we know, when we save or load programs, we are actually utilizing disk storage where the program files reside. We may also place data files on our disk storage. The topic of data files will be addressed in this chapter.

FILE STRUCTURES AND ACCESSING FILES

There are three types of data file structures allowed in BASIC: sequential, random access, and keyed sequential access. File structures of the last type are manipulated by the extensive keyed sequential access method (KSAM) statements available only with 32K Structured BASIC. KSAM will not be discussed in this book.

Sequential data files are files which are written and read in a sequential manner. Data are placed on the file in a fixed sequential order and is retrieved in the same order. Random data files are files which are not written in any fixed sequential order. Data are read from them in an order that is dependent on the application and on the way the data are written. Date are placed on the file in some specific order, which is usually application-dependent, and are retrieved in a way that is meaningful to the application.

When we use a sequential file structure, we might write the numbers 7.4, 3.2, and 5.3 sequentially to a file. In order to retrieve them, we would have to read 7.4 first, 3.2 second, and 5.3 third, i.e., sequentially. With a random file, we might structure it so that every group of five numbers is related in some way. For example, each group of numbers might be the test scores for individuals in five different classes. It may be necessary for our application to examine the fifth number of every group. In the example, this corresponds to the scores of all students in the fifth class. In another application, a different combination of the five might be examined. Assume that the fifth item is of interest here. If we access the file as a random structure, we would read and examine the fifth, tenth, and fifteenth numbers on the file without reading any of the others. It may be advantageous to access the fifth number in each group of five in some order which is application-dependent. We might, for example, read the tenth, fortieth, fifth, and then the ninetieth number, and so on through the file. The access may appear to be random; hence, the name. Random access structures are used in applications in which the data involved have some characteristic that permits us to compute quickly its location in a file. Sequential structures are used in the bulk of application work that requires files.

Despite our emphasis on the order in which data are read and retrieved from the two file structures, it is certainly possible to use a random structure as if it were sequential and to use a sequential structure as if it were random. This will become clearer later when we learn more about how data is read and written. Usually we think of a file as either representing a random or a sequential structure and operate on it accordingly. Let us begin our discussion of the file statements available in BASIC by first finding out how to create a file on disk.

WORKING WITH FILES

In order to place data in a file, the file must have a name and it must exist on disk. We may create a directory entry by using the CREATE statement as in

```
CREATE "MYDATA.DAT"
```

This statement simply creates a directory entry with the name MYDATA and extension DAT on the current disk, in CDOS, or current directory, in CROMIX. To create this same file on the C disk we would use

```
CREATE "C:MYDATA.DAT"
```

There is nothing magical about DAT as an extension; we can use pretty much what we want. Of course, extensions such as COM, BAK, and REL have specific meanings to the operating system, and we should avoid using such extensions. The name of the file can be contained in a string variable, as in

```
CREATE FILENAME$
```

where FILENAME$ contains, say, "MYDATA.DAT". Once a file is created, there is no need to create it again; if you attempt to do so, BASIC will issue an error message.

Now let's do some work with the file. In order to use it, we must tell BASIC that we wish to use it. This is accomplished with the OPEN statement. For example

```
OPEN \4\"MYDATA.DAT"
```

tells BASIC that MYDATA.DAT is to be used and that it is to be associated with the file number 4. The choice of 4 is arbitrary, but the numbers 1 through 8 must be used. The file number, rather than the file name, is used to reference the file in read and write statements for the file. The file number may be an expression or a variable, and the file name may be a string variable so we might use

```
OPEN\FNO\FNAME$
```

where FNO has been assigned 4 and FNAME$ has been assigned the string "MYDATA.DAT".

Note very carefully that we use the backward slash to surround the file number in the OPEN and not the forward slash that is used for division. The backward slash is used in this capacity in all file statements that we will encounter.

When we have finished reading or writing data to a file, the CLOSE statement is used to prevent further access to the file. Examples of the CLOSE are

```
CLOSE\5\
CLOSE\FNUM\
```

Only a file number is used. A file name is not needed.

When a file is closed by CLOSE, the file cannot be read or written without causing BASIC to generate an error message. A file must be opened before it can be accessed (read or written).

Before continuing, let us mention a few miscellaneous things about OPEN and CLOSE. Only 8 files can be open at any one time in BASIC. All files that are open may be closed simultaneously by using CLOSE without a file number reference, or by using CLOSE for file 0, as in

```
CLOSE
CLOSE\0\
```

It is probably worth your while to review some of the error messages that are associated with files. Take a look at the appendix on error messages and note that there are a number of file errors that can be trapped by the ON ERROR statement. See error numbers 128 through 142. Looking for error number 134 is a very useful activity in applications in which the user of an application is to supply a file name. If the file does not exist, perhaps the user mistyped the name; an OPEN will generate this error. By responding to it with an ON ERROR, you can ask the user to specify an existing file.

READING AND WRITING SEQUENTIAL FILE STRUCTURES

Scratch your work area and enter the following program

```
>>100 CREATE "MYDATA.DAT"
>>110 OPEN\4\"MYDATA.DAT"
```

```
>>120 PUT\4\200.22,55.6,77.4
>>130 CLOSE\4\
>>140 OPEN\4\"MYDATA.DAT"
>>150 GET\4\A,B
>>160 GET\4\C
>>170 PRINT A,B,C
>>180 CLOSE\4\
>>190 END
```

Enter RUN and you should see the output

```
200.22          55.6            77.4
```

(If you have trouble executing the program and need to execute it a second time after correcting your errors, you may be stopped at line 100 with a message. The message would result from trying to create MYDATA.DAT a second time. Just delete line 100 and use RUN again. You may have to enter CLOSE in immediate mode to make sure the file is closed before attempting to open it again.)

Let's see what the program is doing. You should be familiar with lines 100 and 110. The PUT statement on line 120 writes data on file number 4. The data consist of three numbers: 200.22, 55.6, and 77.4. Line 130 contains a CLOSE statement, which tells BASIC that no more data are to be placed on the file. We then open the file again at line 140. This causes BASIC to reposition itself to the beginning of the file. The GET statement at line 150 reads the first two data items on the file into A and B. The next GET on line 160 reads the third data item into C. The result of getting or reading the file is seen in the output of the PRINT. The final CLOSE at line 180 closes the file to further use.

Incidentally, if we had not closed and reopened our file, by omitting lines 130 and 140, BASIC would have been positioned after the third item written when we attempted to get A and B. Because there is nothing after the third item, data will be read erroneously. In some instances BASIC will generate an error message. This will be discussed later in connection with detecting an end of file.

It is not necessary to close and reopen a file to cause BASIC to reset itself at the beginning of a file. Use of the following statement is sufficient

```
PUT\4,0,0\
```

The 4 is the file number and should be replaced by whatever file number corresponds to the file you wish to position at its beginning. The two zeroes have a meaning which will be discussed later in this chapter.

PUT AND GET PRECAUTIONS

In our example program, we have introduced two new statements: GET and PUT. We will discuss them more fully next.

A GET simply reads the next data items on a file into the variables listed in the GET statement. The case where we read past any legitimate data that we have placed on the file will be discussed later in this chapter in connection with testing for an end of file condition.

A PUT simply writes data items to a file in the order they are listed in the PUT. Items contained in the list may be arithmetic expressions, numbers, strings, and variables. Here are some examples

```
PUT\5\20.5,X,Y,Z
PUT\3\25.5+15.2/6,AVALUE,"TUCSON"
PUT\2\NAME$,"BOSTON",44.5
```

The file number reference in a PUT or GET may be a variable or arithmetic expression.

For reasons of consistency in reading and writing files in BASIC, it is advisable not to use arithmetic expressions or numbers in the list of a PUT. It is best to use variables to write data to files. Assign numbers or expressions to a variable, and write the variable to the file. If you do place a number in the list of a PUT, make sure it is a decimal number (use a decimal point). In our example programs, we have taken care to write decimal numbers, and not integers, to files. The reason for these cautions will become apparent later as you acquire a firm understanding of how numbers and variables are stored internally to BASIC. The idea behind these precautions is to assure that the type of number (integer, float) written is the same as the type of variable used to read them.

When reading strings, it is important to read exactly the number of characters in the string into a string variable. Consider the following simple program

```
100 DIM ALPHA$(5)
110 OPEN\2\"MYDATA.DAT"
120 PUT\2\"ABCD",44.5
130 CLOSE\2\
140 OPEN\2\"MYDATA.DAT"
150 GET\2\ALPHA$,A
160 PRINT ALPHA$,A
170 CLOSE
180 END
```

This program writes "ABCD" and 44.5 to the file and then reads the items into ALPHA$ and A. It will not work satisfactorily. The string "ABCD" is written as four characters. When we attempt to read it into ALPHA$ at line 150, BASIC sees that ALPHA$ is defined as 6 (=5+1) characters and tries to read six characters from the file. It will succeed by taking some of the non-character representation of 44.5, which follows it on the file. The fifth and sixth positions of ALPHA$ will contain garbage (undecipherable or unusable characters). When A is read, part of it is already in ALPHA$ and the result placed in A will also be garbage. We can remedy this situation by using DIM ALPHA$(3) in line 100. That is, the dimension of the string variable is made to match the length of the string on the file. Another way of getting around the difficulty with reading and writing string data is discussed in the next section.

One other precaution should be mentioned. When data are written to a file using PUT, they are placed on the file in an internal computer format which can be read by the GET. It is not possible to list these files with CDOS or CROMIX commands such as the CDOS TYPE command. The CDOS DUMP utility program can be used to list files. See the CDOS Instruction Manual provided by Cromemco for details on how to use DUMP. DUMP may not be accessed from BASIC.

USING SIMPLE STRING VARIABLES WITH PUT AND GET

A particularly important observation was made in Chapter 10 about how BASIC treats an unsubscripted (simple) string variable differently in input and output situations. Upon output, the length (LEN) of the variable is considered, but, upon input, the dimension is considered. This can result in a rather unusual result. For example, consider the program

```
100 DIM BETA$(5)
110 OPEN\2\"MYDATA.DAT"
120 LET BETA$="XYZ"
130 PUT\2\BETA$
140 CLOSE\2\
150 OPEN\2\"MYDATA.DAT"
160 GET\2\BETA$
170 CLOSE
180 END
```

Line 130 places the three-character string "XYZ" on the file because BETA$ is used as output and the length of BETA$ is 3. At line 160, the GET causes six characters to be placed in BETA$ because BETA$ is used as input and it has a dimension of 5. The three extra characters read from the file will probably not make sense when they are printed.

To make sure that we always GET and PUT the same number of characters with a string variable, we can use the beginning and ending substring reference approach that was recommended earlier in connection with using substring references. In some cases, it becomes awkward to remember or to specify the ending position. An easy way to keep our reference to simple string variable consistent is to use a single negative subscript. In lines 130 and 160, we should replace BETA$ with BETA$(-1). As we discussed in Chapter 10, this always causes BASIC to input and output as many characters as are associated with the dimension of the variable. The PUT at line 130 would place six characters on the file: "XYZ" followed by three null characters. The GET at line 160 would read all six characters.

OPENING FILES

Let's return to the OPEN statement for awhile. There are two parameters other than the file number that the OPEN statement uses. The first is the record size for the file. It is specified in the following examples

```
OPEN\4,256\"MYDATA.DAT"
OPEN\6,512\"AFILE.DAT"
```

The 256 and 512 in the respective OPEN statements tell BASIC the record size. For most sequential file applications, this is not an important number. If a number is not given, BASIC assumes 128. The importance of this parameter will become clearer when we consider random access structures later in this chapter. For sequential access, 128 is a reasonable choice.

The second of the two parameters tells BASIC whether you plan to read, write, or both read and write on the file during its use in the program. If neither is specified, then both read and write are assumed. This was the case in our previous examples. It is not often that applications both want to read and write into a file. Usually we either read or write. This is true in most cases where we are dealing with a sequentially structured file. Examples of how this parameter is used in the OPEN are

```
OPEN\5,128,1\"SOMEFILE.OLD"
OPEN\4,100,2\"MASTER.SAL"
OPEN\7,256,3\"OLDMAST.INV"
```

The third item between the backward slashes is a code which designates the file mode

Mode	Meaning
1	Read only
2	Write only
3	Both read and write (default)

It might be tempting to always use mode 3, read and write, but there is a reason to use the other modes. If you are using the read mode and accidentally try to write on the file, BASIC will issue a message and stop. The opposite is true if you are using the write mode. Thus, these modes can be used to protect your files from being inadvertently damaged.

Numeric variables and arithmetic expressions may be used to define the file number and parameters used in the OPEN. The following are valid forms of the OPEN

```
OPEN\INFILE'NO,RECSIZE,RDONLY\ MASTER$
OPEN\6,SIZE,DUAL\ "PAYROLL.MAR"
OPEN\FILENO+2,256,WRONLY\ NEWMAST$
```

RECORDS

A fundamental concept that is necessary to understand when working with random access structures is that of a record. BASIC views a file as consisting of a series of records. A record is a group of related items that occupy a fixed number of positions. For example, a record might consist of the name, address, and age of an individual or a record may consist of only one item. Each item in a record may be thought of as occupying a field of fixed length within the record.

BASIC essentially is unconcerned with the concept of a record when processing sequential structures. It does not need to know where a record begins or how long it is. We get to the beginning of the each successive record by reading a fixed number of items. In a random access structure, we may want to go directly to the tenth record without reading anything in between. If BASIC knows the exact length of a record, it can position itself in the file without reading anything. It simply skips as many positions as needed from the beginning of the file, enough positions to cover nine times the record length in our example.

How do we compute the length of a record? A record's length is determined by the type of data we write into it. Assuming we only write records with variables or strings (as advised earlier), we can use the following guide

Item Type	Length
Long float	8
Short float	4
Integer	2
String	String length

If our record consists of six long floating point fields and one string of 12 characters, the record length is 60. This is the record size value we should use to open the file in the OPEN statement. In the terminology often used in BASIC, we would say the record has a length of 60 bytes. A byte is a fundamental unit of storage. A single alphabetic character occupies 1 byte of storage.

In working with random access structures, BASIC allows us to position a read or write operation at the beginning of a record or at any position within the record. In the next section, we will see how this is accomplished.

ACCESSING RANDOM FILE STRUCTURES

Our preceding discussion and program examples were concerned with using file structures in a sequential fashion. Repeated use of PUT or GET advanced us sequentially through a file. By the introduction of a few parameters that were not previously discussed, we can proceed through a file in whatever order best suits our application.

As an example, suppose we have a record that contains three long floating point items, i.e., a record length of 24 (or 24 bytes). The following program writes three records of data and then reads the data of the second record and prints it at the terminal. Scratch your work area and enter it.

```
>>CREATE "RANDOMF.DAT"
>>100 OPEN\1,24\"RANDOMF.DAT"
>>110 PUT\1,1\44.4,55.5,66.6
>>120 PUT\1,0\11.1,22.2,33.3
>>130 PUT\1,2\77.7,88.8,99.9
>>140 GET\1,1\A,B,C
>>150 PRINT A,B,C
>>160 CLOSE\1\
>>170 END
```

We use the immediate mode to create the file RANDOMF.DAT. Line 100 opens the file as file number 1, and defines each record as containing 24 bytes. This record size is sufficient to accommodate three long floating numbers. Since a file mode parameter is not given in the OPEN, the default is both read and write mode. On lines 110 through 130, three data items are put into records 1, 0, and 2. Note that the records are not written in the sequential order 0, 1, and 2. (BASIC numbers records from 0.) The record number follows the file number in each PUT. Line 140 is a GET that points to record 1, and is designated by the 1 following the file number. The three data items are read into A, B, and C using the GET. Line 150 displays the values read. The output from this program is

```
        44.4            55.5            66.6
```

When using GET and PUT in the manner just described, these forms are referred to as the random access forms of PUT and GET. Sometimes we call this manner of using them the random access method of accessing file data.

It is possible to refine our ability to read or write at specific points in a file. Replace line 140 in the program with

```
>>140 GET\1,1,8\B
```

and re-execute the program. You will find the output is now

```
0                55.5             0
```

A and C are not read and are 0. The third value between the backward slashes in the GET indicates that we want to read beginning at the eighth position in the record (BASIC uses 0 as the first position in a record). This position is the beginning of the field containing the second number in the record, or 55.5. BASIC leaves its pointer on the next item in the file at the position immediately following the last item read. This pointer is kept internal to BASIC, and it is referred to as the file pointer. A file pointer exists for each active file.

If we were to have the statement

```
145 GET\1\C
```

following the line 140 change that we just made, this statement would cause 66.6 to be read into C. A record number and record position are not needed here because BASIC remembers the last position read in the file from the current position of the file pointer. After the statement

```
140 GET\1,1,8\B
```

is executed, the file pointer is at record 1 and position 16 (B is a long variable requiring 8 bytes of storage). The reading of B moves the pointer 8 bytes to position 16. When C is read at line 145, the next number, 66.6, is moved into C.

Some valid examples of the GET statement in its random access form are

```
GET\3,20,2\A,B,C
GET\1,10,40\NAME$
GET\1,RECNUMBER,TOTAL'FLD\TOTAL
```

The PUT statement can be used to place data anywhere within a record by giving the values for the record number and the position within the record. Some valid forms are

```
PUT\3,9,2\33.45,Z,TOTAL
PUT\2,0,28\INTEREST,SALES
PUT\4,RECNO,NAME'FIELD\"SMITH, JOHN"
```

The PUT moves the file pointer to the position following the last data item placed on the file.

When PUT is used without a list of data items in its random access form, it positions BASIC at a specific record and location within the record. The next read or write will begin at the position specified. As we stated in connection with sequential file operations, specifying a record number and record position of 0 (for example, PUT\3,0,0\) is a useful way of positioning the next read or write operation at the beginning of a file. In this particular form, a PUT may even be used to position a file which is in read-only mode; nothing is actually written to the file.

When using a statement of the form

```
GET\1,5\ZIP
```

in which the position within the record is omitted, a value of 0 is assumed. The same is true for PUT.

USING INPUT AND PRINT WITH FILES

As we mentioned earlier, GET and PUT operate on files with data kept in an internal computer format. There are some applications where we would like to produce and manipulate data in its external form, i.e., the natural form in which we expect to see numbers written. The external form of the number 44.77 is just 44.77. Its internal computer representation is slightly more complex and not suitable for interpretation by most users. The external format is sometimes referred to as the ASCII format.

The output of the PRINT (or PRINT USING) statement is in an external format. The PRINT statement can be used to write data in the same form to files. Similarly, the INPUT statement can be used to read the external form on a file. INPUT, PRINT, and PRINT USING operate exactly the same way when they are used to read and write data from and to the terminal. Some example forms are

```
INPUT\4\ABC,CITY$
INPUT\3,2,1\A,B,C
INPUT\6,1\LINE$
PRINT\2\A,B,C
PRINT\6,0,10\LINE$
PRINT\2,4\NAME$,WGT,SIZE
PRINT USING\2\"TOTAL: ####.##",SUM
```

Like GET and PUT, the file number, record number, and position within a record are specified between a pair of backward slashes.

The output written on a file by PRINT is most suitable for printing or display rather than as file input to a program. Files created by PRINT are easily displayed and printed with commands like the CDOS TYPE command. It is not recommended that files created by PRINT be read with either GET or INPUT. Additional details on reading PRINT produced files may be found in the Structured BASIC Instruction Manual.

Basically, INPUT reads strings of ASCII characters, the characters you see on your keyboard, and expects each string to be terminated by a carriage return and line feed character. This is similar to its expectations when you enter data for an INPUT which reads the data from the terminal instead of a file. When you depress the return key on your keyboard, a carriage return and a line feed are transmitted to BASIC, along with any other ASCII characters you entered on the line. The string of ASCII characters entered at the terminal is interpreted in the same way as when they come from a file. In either case, the string of ASCII characters cannot exceed 132 characters.

Perhaps one of the more common uses of INPUT with files is to read text prepared by an edit program (not the edit command in BASIC), or to read a file created by LIST. A program that reads a file created by LIST and counts the number of lines in the program is shown next.

```
100 DIM NAME$(15),LINE$(131)
110 PRINT"ENTER PROGRAM NAME";
120 INPUT NAME$
130 OPEN\1\NAME$
140 LINES=0
```

```
150 ON ESC GOTO ENDFILE
160 *BEGIN:INPUT\1\LINE$
170 LINES=LINES+1
180 GOTO BEGIN
190 *ENDFILE:PRINT
200 PRINT"PROGRAM: ";NAME$
210 PRINT"NUMBER OF LINES: ";LINES
220 CLOSE\1\
230 END
```

If you use this program, you must enter the name of a file that has been created with LIST as the response to the prompt at lines 110 and 120. The ON ESC statement in line 150 is used to detect the end of file. This topic is discussed in more detail later. The INPUT at line 160 reads from the file named. Each line in the file is a string of ASCII characters that has been terminated with a carriage return and line feed character. Each line is read into LINE$. When the program tries to read beyond the last line, an escape interrupt is generated and the ON ESC causes the program to transfer to line 190.

It is tempting to replace the INPUT from the file in the preceding program with a GET. The result would be quite different. Instead of reading data to the next carriage return and line feed into LINE$, the next 132 characters of data would be read. GET reads as much data into a string variable as the variable can hold. The result produced would be a count of the number of 132 character blocks in the program, and not the number of lines.

An application where PRINT is useful is in connection with writing data directly to a printer without placing it on the terminal screen. This is especially important in CROMIX, since it has no other way of simply writing to the printer. Writing directly to the printer is accomplished by opening a special file called $LP (do not use lowercase, i.e, $lp), and then directing all printed output to the file number associated with $LP. For example

```
400 OPEN\5\"$LP"
410 PRINT\5\"SEND ME TO THE PRINTER"
```

FILE EXAMPLE—LITTLE LEAGUE BASEBALL RECORDS

It is probably a good idea to take a look at an example of file usage. For this purpose, suppose we would like to keep statistics on the number of times at bat and the number of hits per player for a Little League baseball team of 12 players. To establish our recordkeeping system, we will place the data for each player on a record in a file on the computer. We will update the player's record after each game.

A simple approach to manipulating this data on a file is to use a file record for each player. A record will consist of the player's name, the number of times at bat, and the number of hits. We will allow room for a 20-character name. If we write times at bat and hits as long floating variables, they will each occupy 8 bytes in a record. A record, therefore, needs to be at least 36 bytes. Hence, we choose 36 bytes as a record size. For convenience, we will use jersey numbers as a way of referring to the players. The team has jersey numbers from 1 to 16 with numbers 6, 7, 12, and 13 missing. Record 1 will correspond to the player with jersey number 1, and so forth.

Assume that we are in the middle of the season when we decide to begin our task. First, we must initialize a data file with the current data on the players.

A program for initializing our data file, TEAMSTAT.DAT, follows. Lines 100 to 210 contain DATA statements for each player. A player's jersey number, name, number of times at bat, and number of hits are contained in each statement. Line 220 indicates that data for the last player preceded this DATA statement. At 500, NAME$ is defined to allow 20 characters. Variables are set in lines 510 and 520 for our file number and record size. The use of these variables in file statements will help clarify file statement parameter lists. Although we could create the file TEAMSTAT.DAT outside the program by using CREATE in immediate mode, we use it as a statement in line 530. Line 540 opens the file in write-only mode. The loop from 560 to 580 writes null string data into every name field, and 0 values into every record from 1 to 16. This assures us that the records for which there are no players will contain some recognizable data. Actual data will be placed in records for existing jersey numbers later in the program. Note that the variable ZERO is used instead of 0.0 in keeping with our precautions about not using literal data in PUT statements. Lines 590 to 620 read the data from the DATA statement and put the data in the record number corresponding to the player's jersey number. At line 620, NAME$(-1) is used to make sure all 20 characters of NAME are written. When the IF at line 600 determines that the jersey number just read is zero, the program terminates by going to QUIT, whereupon the file is closed.

Program to Initialize TEAMSTAT.DAT File

```
100   DATA 1,"BOBBY FELLER",40,12
110   DATA 2,"MIKE MANTLE",55,18
120   DATA 3,"JOHNNY SEAT",55,17
130   DATA 4,"NOLAN ROLAN",44,12
140   DATA 5,"ALAN CALINE",62,22
150   DATA 8,"CROOKS ROBINSON",47,17
160   DATA 9,"SPARKY MYLE",30,8
170   DATA 10,"ROLLIE THUMBS",25,8
180   DATA 11,"YOGGIE BEARA",49,17
190   DATA 14,"ROBIN MOUNT",54,18
200   DATA 15,"FERNANDO VENEZUELA",38,14
210   DATA 16,"ROCKY HENDERSON",35,12
220   DATA 0,"",0,0
500   DIM NAME$(19)
510   FILENO=1
520   RECSIZE=36
530   CREATE"TEAMSTAT.DAT"
540   OPEN\FILENO,RECSIZE,2\"TEAMSTAT.DAT"
550   ZERO=0.0
560     FOR J=1 TO 16
570     PUT\FILENO,J\NAME$(-1),ZERO,ZERO
580     NEXT J
590 *MOREDATA : READ JERSEY'NO,NAME$,AT'BATS,HITS
600   IF JERSEY'NO=0 THEN GOTO QUIT
610   PUT\FILENO,JERSEY'NO\NAME$(-1),AT'BATS,HITS
620   GOTO MOREDATA
630 *QUIT : PRINT
640   PRINT"TEAMSTAT.DAT FILE CREATED. READY FOR USE"
650   CLOSE\FILENO\
660   END
```

Now we will consider the next program shown, which allows us to update the number of times at bat and number of hits for each player. When executed, the program will prompt us for a jersey number. Responding with a 0 terminates the program and lists all the players' statistics, including batting averages. When a jersey number for nonexistent players is given, a prompt for a correct number is given. Giving a valid jersey number causes the current statistics for the corresponding player to be listed, and a prompt for the new data (times at bat and hits for the latest game) is issued. Supplying this data causes the record to be updated and the new statistics to be printed. A prompt for a new jersey number is given when the update is complete.

In the code for the program, line 530 opens the file in both read and write mode. This is very important for updating records, since the data must be read and then written. Lines 540 to 570 process the jersey number. The jersey number is checked at line 560. If the number is 0, then a branch to QUIT, line 720, is made. If a number larger than 16 is accidentally entered, line 570 issues a warning and branches to prompt for a new jersey number. If the number is valid, line 580 is reached, and the data from record number JERSEY'NO is read into NAME$, AT'BATS, and HITS. If NAME$ is null (line 590) there is no player for the jersey number. A message is issued to that effect, and a branch to the jersey number prompt code is made. Upon finding a non-null name, the statistics for the player are listed in 620 to 630. Lines 640 to 660 request the player's data for the latest game. At line 670, the new statistics are computed. When the new statistics from lines 680 to 690 are displayed, the statistics are written back into the record at line 700. Line 710 takes us back to the jersey number prompt. When a jersey number of 0 is finally entered, lines 720 to 830 are used to print the statistics for the entire team. Note that the loop in this code causes all 16 records to be read. When a record containing data for a non-existent player is read, line 790 skips the printing of the data for that player.

Program to Update Player Records

```
500   DIM NAME$(19)
510   FILENO=1
520   RECSIZE=36
530   OPEN\FILENO,RECSIZE,3\"TEAMSTAT.DAT"
540 *ASK : PRINT
550   INPUT"ENTER JERSEY NO.-  ",JERSEY'NO
560   IF JERSEY'NO=0 THEN GOTO QUIT
570   IF JERSEY'NO>16 THEN PRINT"INVALID NO." : GOTO ASK
580   GET\FILENO,JERSEY'NO\NAME$,AT'BATS,HITS
590   IF NAME$="" THEN PRINT"NO SUCH PLAYER" : GOTO ASK
600   PRINT
610   PRINT"PLAYER:  ";NAME$
```

```
620    PRINT USING"AT-BATS:  ### HITS: ### ",AT'BATS,HITS;
630    PRINT USING"AVERAGE:  ###.##",1000*HITS/AT'BATS
640    PRINT
650    PRINT"ENTER GAME AT-BATS & HITS (0'S IF SKIP) ";
660    INPUT G'AT'BATS,G'HITS
670    AT'BATS=AT'BATS+G'AT'BATS : HITS=HITS+G'HITS
680    PRINT USING"AT-BATS: ### HITS: ### ",AT'BATS,HITS;
690    PRINT USING"NEW AVERAGE: ###.## ",1000*HITS/AT'BATS
700    PUT\FILENO,JERSEY'NO\NAME$(-1),AT'BATS,HITS
710    GOTO ASK
720 *QUIT : PRINT
730    PRINT"            TEAM STATISTICS"
740    PRINT
750    PRINT"NAME                 AB  H  AVG"
760    PRINT"----------------  --- -- ---"
770      FOR J=1 TO 16
780      GET\FILENO,J\NAME$(-1),A,H
790      IF NAME$="" THEN GOTO NOPLAYER
800      AVG=1000.0*H/A
810      PRINT USING"################",NAME$;
820      PRINT USING" ### ## ###",A,H,AVG
830    *NOPLAYER : NEXT J
840    CLOSE\FILENO\
850    END
```

The following example shows how the record for Johnny Seat, jersey number 3, is updated. He had 4 at bats and 2 hits in the latest game. The arrows at the right show where data were entered in response to prompts from the program. In the team summary, note that only Seat's data have been updated.

Example Execution of the
Baseball Update Program

```
ENTER JERSEY NO.-  3                                  <---  Enter 3

PLAYER:  JOHNNY SEAT
AT-BATS:   55 HITS:  17 AVERAGE:  309.09

ENTER GAME AT-BATS & HITS (0'S IF SKIP)  4,2          <---  Enter 4,2

AT-BATS:  59 HITS:  19 NEW AVERAGE: 322.03

ENTER JERSEY NO.-  0                                  <---  Enter 0
```

```
             TEAM STATISTICS

NAME                 AB  H  AVG
----------------    --- -- ---
BOBBY FELLER         40 12 300
MIKE MANTLE          55 18 327
JOHNNY SEAT          59 19 322
NOLAN ROLAN          44 12 273
ALAN CALINE          62 22 355
CROOKS ROBINSON      47 17 362
SPARKY MYLE          30  8 267
ROLLIE THUMBS        25  8 320
YOGGIE BEARA         49 17 347
ROBIN MOUNT          54 18 333
FERNANDO VENEZUELA   38 14 368
ROCKY HENDERSON      35 12 343
```

DETECTING THE END OF FILE

The topic of detecting the end of file is an interesting one. We will see that there are several ways to determine the end of file.

When reading a file in a sequential manner, we would like to know when we have read the last data item. There is no simple facility in BASIC that indicates that we have come to the end of file data. It is usually a good idea to write a count of the items on the file at the beginning, so that the programs know how many items to read. An example program that illustrates this idea is

```
100 OPEN\1\"MYFILE.DAT"
110 GET\1\TOTAL
120 COUNT=0
130 *BEGIN:IF COUNT>=TOTAL THEN GOTO ENDFILE
140 GET\1\ANUMBER
150 PRINT ANUMBER
160 COUNT=COUNT+1
170 GOTO BEGIN
180 *ENDFILE:PRINT "END OF JOB"
190 END
```

Line 110 reads the first number from the file MYFILE.DAT. We assume this number contains the total number of data items on the file. COUNT is initialized to 0 and is used to keep track of how many numbers we read from the file. When COUNT exceeds TOTAL, line 130 will transfer to the ENDFILE line name. Lines 140 through 170 read numbers from the file, print the numbers, and repeat these operations.

We might expect that BASIC would produce an error message if we wrote five items to a file and then attempted to read these five items plus two more. In most cases, we can read past the last item written on a file because BASIC always creates files in blocks of 128 bytes, which are called sectors. As we learned earlier in our discussion of records, five long floating point numbers take up 40 bytes. The remaining bytes in the sector often contain zeroes or null characters. (They probably won't if BASIC has re-used a discarded sector. BASIC grabs discarded or free sectors on the disk to add to your file as they are needed.) When the two extra items are read, they will contain zeroes. It is, therefore, very important to be able to tell where the end of file is by the means described previously. BASIC will produce an error message when we try to read past the physical end of file, the end of the last sector, and we can use that to our advantage in some applications. This possibility is discussed shortly.

As an aside, the fact that BASIC allocates file space on disk in terms of sectors rather than records may be confusing. However, as we mentioned earlier, the concept of records is used for random file structures and access methods. The concept of sectors is common to both.

A simple but effective way of finding the end of file is to write an unusual or unique data value as the last item on the file. For example, the last value written might be 1.0E+50. This value is so large that it is unlikely that it would serve as the end-of-file item in many applications. When reading the file, the program checks to see if this value is read. Some care must be exercised in choosing an unusual value.

Another way of determining whether we have read past the last data item is to have the program that wrote the file record place information in the file about the location of the last item. The IOSTAT function that is discussed later in this chapter is able to tell us the location of the last item when it is written. If we place this information at the beginning of our file, it can be used when the file is read. Since this information is not known until the last item is read we should leave space for it at the beginning of the file so we do not have to write the entire file again. Before attempting such a scheme, you should become very familiar with PUT and GET as they are used in random access methods, and the concepts of records and sectors. Sectors are discussed in connection with the IOSTAT function.

Another method of detecting the end of file is applicable when we sequentially read records using random access methods. If we attempt to read an item that is beyond the physical end of the file, BASIC will generate an error message and stop. This fact can be used to our advantage when we use the ON ERROR statement that was discussed in an earlier chapter. When BASIC attempts to read past the last item on a file, error number 138 is generated and 138 is put in system parameter 3. If we are using ON ERROR, the program will transfer to a portion of the program that handles the end of file. The system function SYS(3) allows us to examine whether 138 is in fact the error. A program that illustrates these concepts is

```
100 OPEN\1\"MYFILE.DAT"
110 ON ERROR GOTO EXAMINE
120 RECNO=0
130 *BEGIN: GET\1,RECNO\A
140 PRINT A
150 RECNO=RECNO+1
160 GOTO BEGIN
170 *EXAMINE:IF SYS(3)=138 THEN GOTO ALLDONE
180 PRINT "NOT TERMINATED BY AN END OF FILE"
190 *ALLDONE: CLOSE\1\
200 END
```

This program keeps getting the the first item in each record in the file and printing it. RECNO keeps track of which record we are to read. Eventually the end of file will be reached and it will transfer to line 170 since we are using ON ERROR. It is unlikely that any error other than 138 will be encountered, so the check at line at 170 is superfluous.

The method for detecting an end of file for files written by a text editor or the LIST command is different than we just described. We assume that such files are to be read with INPUT (see the earlier discussion of INPUT). Such files contain a special character at the end, which causes BASIC to produce an escape interrupt. This interrupt can be detected by the use of the ON ESC statement, as was shown in the program example concerning the use of INPUT to read files.

BASIC FILE COMMANDS

The BASIC command ATTRIB, or ATRIB, which was discussed in Chapter 2, is applicable to data files as well. Two other commands, which have applicability to either program or data files, are RENAME and ERASE. These commands are similar to the CDOS REN and ERA commands. They were discussed in Chapter 2 also, but we will explore them further here. We want to stress that they can be used as statements as well as commands and that string variables can be used to define the files referred to in these statements. ATRIB can be used as a statement too, but it probably has a less frequent use in the statement form than ERASE and RENAME.

RENAME simply changes the name of a file from an old name to a new name. Some examples are

```
150 RENAME "OLDFILE.TXT","NEWFILE.TXT"
170 RENAME OLDNAME$,"MASTER.DAT"
```

Note that string variables may be used.

ERASE simply removes a file from the directory so that it no longer can be accessed. Some examples of the ERASE command are

```
120 ERASE "INVMASTER.DAT"
290 ERASE "C:OLDPAYRL.DAT"
400 ERASE MASTER$
```

Like the RENAME command, string variables may be used.

A FILE FUNCTION

There are instances when we would like to know where we are in the file. That is, we would like to know the record and the position within that record where BASIC is positioned for its next read or write. We cannot tell which record BASIC is positioned at directly, but we can use the IOSTAT function to help determine the position. If we give the IOSTAT function both a file number and an information request code as its arguments, IOSTAT will return a value that gives us information about the file. There are three possible codes. The first code, 0, is illustrated in the example

```
LET FSTATUS=IOSTAT(5,0)
```

which requests the file status of file number 5. The 0 is a code that requests the file status. FSTATUS will be set to one of the following three values

Status Value	Meaning
0	File is OK.
1	Positioned at the physical end of file, i.e., the end of the last sector.
2	Positioned at a record which was never written by a random access write.

The second code, 1, returns the sector number where BASIC is positioned in the file. While BASIC does view files in terms of records, it also views them internally as sectors. A sector is just 128 bytes or characters of file data. A file with 64-byte records would have its first two records contained in the first sector of the file. Specifying a code of 1 in IOSTAT as in

```
LET FSECTOR=IOSTAT(5,1)
```

sets FSECTOR to the sector number at which BASIC is positioned within file 5. Sector numbers begin with sector number 0.

The third code, 2, returns the position within the sector where BASIC is positioned in the file. Like the record position, the first position is numbered 0. Specifying a code of 2 in IOSTAT as in

```
LET FSECTPOS=IOSTAT(5,2)
```

sets FSECTPOS to the position number at which BASIC is positioned within file 5.

In summary, IOSTAT has the following meanings

```
IOSTAT(fileno,0)   -   return status (0, 1, or 2)
IOSTAT(fileno,1)   -   return sector number
IOSTAT(fileno,2)   -   return byte position in sector
```

where fileno is a file number, variable, or expression.

USING GET FOR TERMINAL INPUT

An interesting use of the GET statement occurs when it is used to obtain data from the terminal. An example of this use of the GET statement is

```
100 PRINT "Hit any key to continue"
110 GET\0\A$(0,0)
120 END
```

File number 0 designates the console or terminal. When this code is executed, the message at line 100 will be displayed, and the program will pause at line 110 until any key is depressed upon the keyboard. This use of GET is a very handy mechanism for causing a pause in the output of a program. This ability is used to give the person who is interacting with the program a chance to read the output before continuing. The reader of the output can thus proceed at his or her own pace.

Review Questions and Exercises

1. If a program reads the fifth and then then tenth record in a file, what access method is it most likely to be using?

2. What is missing in the statement OPEN\2\?

3. Why does CLOSE/3/ result in a syntax error?

4. Is it a good idea to use PUT\3\22? Why is it preferable to use PUT\3\22.0?

5. The definition of a record length is important to random access of files. Why? If a record is to be written with three long floating variables, why is the choice of 24 bytes as a record length reasonable? How much record space would be wasted if we choose a length of 30?

6. Which record does the statement PUT\5,25\ refer to? Which record does PUT\5,0\ refer to?

7. Which byte in record 8 does GET\6,8,20\ refer to? How about GET\6,8,0\?

8. What is a read-only file? What code in an OPEN specifies such a file?

9. PUT\5,10,0\ positions a file at record 10, even if the file is a read only file. How do you position a file at the first byte in the first record? Do you use PUT\5,0,0\ or PUT\5,1,1\?

10. BASIC has no specific statement that allows us to detect an end of file. Name several methods that can be used to detect an end of file.

Summary

Sequential File Structure
A file organized in such a way as to make it easy to read from or write to it in a sequential fashion with BASIC sequential access methods.

Random File Structure
A file organized in such a way as to make it easy to write or read it in a random, or application-dependent way, with BASIC random access methods.

Sequential Access Method
A GET or PUT statement that does not reference a record number.

Random Access Method
A GET or PUT statement that references a record number.

Records
A group of data items which represent related items.
Records have lengths measured in bytes.

End of File
Applicable to sequential file strucutes.
The occurrence of going beyond the last data item on a file.

CREATE
Creates a directory entry for a file with a specified name.

OPEN
Makes a file accessible through GET and PUT statements.
Files may be opened in three modes: read only, write only, or both read and write.
Defines the size of a record for use of a file as a random file structure.
Repositions BASIC to the first data item in the file.

CLOSE
Makes a file inaccessible to other file operation statements.
Allow for the closure of individual files or all files simultaneously.

GET
Reads data from a file.
Sequential access form does not require a

record number.
Random access form requires a record number.
Position number within a record may be specified for the random access form.

PUT
: Writes data to a file.
Sequential access form does not require a record number.
Random access form requires a record number.
Position number within a record may be specified for the random access form.
When used without a list of data items in its random access form, it positions BASIC at a specific record and location within the record.

File Number
: Number associated with a file when the file is opened.
Used to reference the file in read, write, and close operations.

Sector
: Fundamental unit of file space allocation on disk.
128 bytes of disk space.

File Pointer
: Internally, BASIC keeps track of where the next read or write is to begin with a file pointer.
The file pointer is stated in terms of a record number and the position within the record number.
Changed by using a sequential or random access method operation.
Set to the beginning of a file when the file is opened.

INPUT
: Functions exactly like the INPUT to read data from the terminal except that it expects its input from a file.
May be used as a random access method.
End of data is read when a carriage return and line feed character are found in the input.

PRINT
: Functions exactly like the PRINT to print data at the terminal except that it places its output into a file.
May be used as a random access method.
Files produced by PRINT are better suited for

display or printing than as files to be read by other statements.

RENAME

Renames a file.
Command form is more widely used than its statement form.

ERASE

Erases a file.
Command form is more widely used than its statement form.

IOSTAT

System function which provides information about the status of a file or the location where BASIC is pointing for the next read or write.

CHAPTER **12**

Modern Programming Structures

In an earlier chapter we learned about control statements such as FOR-NEXT, IF, and GOTO. We mentioned that the GOTO is a controversial statement in programming. GOTO derives most of its negative reputation from the fact that it is often used to excess, thereby creating programs which are hard to debug and modify. When it is used excessively, the resulting programs tend to resemble spaghetti in their appearance because transfers are made up, around, and down in the program. Programs tend not to flow generally from the first statement to the last when GOTO is used frequently; they lack a certain structure, which makes them difficult to read and understand.

These situations and others are remedied by program statements which generally provide a simpler structure to the programs and make them easier to modify and debug. Basically, these new statements are related to the IF and FOR-NEXT statements that were previously discussed. They have become increasingly popular in programming languages in recent years and will be discussed in detail in this chapter.

GROUPING STATEMENTS TOGETHER—DO-ENDDO

A particularly simple pair of statements that help to add structure to our programs is the DO-ENDDO pair. They do nothing more than provide a beginning and an end for a group of related statements. Their use is illustrated by the following section of a program

```
200 DO
210 PAGENO=PAGENO+1
220 PRINT "                              PAGE ";PAGENO
230 PRINT
240 PRINT "NAME      ADDRESS              PHONE"
250 PRINT "-------   ---------------   ----------"
260 LINENO=4
270 ENDDO
280 PRINT USING "#######",NAME$;
```

When line 200 is executed, nothing really happens and execution of the following statements is made in the expected order. The execution of the ENDDO has no special effect. In this form, DO-ENDDO are not very useful, but when it is used with the IF statement, it becomes very useful. (Another place where these statements are effectively used is with the variable stacking and recursive capabilities of BASIC.) The DO-ENDDO can be used to define a block of statements that are executed when the logical expression of an IF statement is true. For example

```
200 IF LINENO>66 THEN DO
210 PAGENO=PAGENO+1
220 PRINT "                              PAGE ";PAGENO
230 PRINT
240 PRINT "NAME      ADDRESS              PHONE"
250 PRINT "-------   ---------------   ----------"
260 LINENO=4
270 ENDDO
280 PRINT USING "#######",NAME$;
```

When line number 200 is executed and LINENO is greater than 66, then the group of statements surrounded by DO and ENDDO will be executed. If LINENO is not greater than 66, then the statement following the ENDDO will be executed next.

It appears that the only advantage gained here is that we avoided coding a GOTO and a line name for the GOTO transfer. Another advantage becomes apparent if we use LIST on this program section. Using LIST would show that statements 210 through 270 are indented, which makes them stand out as a related group of statements. This highlighting of related statements is very effective when a program contains many statements and we are trying to modify it.

A simple form of grouping related statements for execution when only the logical expression in an IF statement is true was discussed in an earlier chapter. It is restated here in the example

```
IF ANUM=45 THEN A=4 : B=B+10 : PARTNO$="ABC-4"
```

If ANUM is 45, then the three assignment statements following the THEN are executed; otherwise, control is passed to the next line.

TWO-PART LOGICAL GROUPS—ELSE

Another statement which is related to the IF-DO-ENDDO statements is the ELSE. Here is an example of its use

```
150 IF LINENO>66 THEN DO
160 LINENO=0
170 ELSE
180 LINENO=LINENO+1
190 ENDDO
```

When LINENO is greater than 66, execution begins at the line following the IF-THEN-DO. It continues to the ELSE, where transfer is then made to the statement following the ENDDO. If the logical expression is false, transfer is made to the statement following ELSE and continues through the ENDDO. ELSE, therefore, acts as a divider within the DO-ENDDO to divide statements which are executed when the logical expression is true and when it is false. Only one ELSE is allowed to divide a DO-ENDDO group.

It is perfectly legal to place the group of IF-THEN-DO-ELSE-ENDDO statements inside DO-ENDDO groups. For that matter, any statement is legal inside a DO-ENDO group.

Like the FOR-NEXT statements, the DO-ENDDO statements always come in pairs. Some care should be exercised in making sure that the DO and its ENDDO are correctly placed in the program. If an ENDDO is encountered without having been preceded by a DO, BASIC will generate an error message. If DO is executed, but the corresponding ENDDO is not executed before the program terminates with an END, BASIC will issue an message that a stack error has occurred.

LOOPING UNDER LOGIC CONTROL—WHILE-ENDWHILE AND REPEAT-UNTIL

A very effective statement pair for controlling loops and for determining whether or not a group of statements should be executed are the WHILE and ENDWHILE statements. Like DO-ENDDO they group a related set of statements together. However, the statements grouped together by WHILE-ENDWHILE are executed only while a logical expression in the WHILE is true; the group of statements is executed repeatedly while the expression is true. Scratch your work area and enter the program

```
>>100 COUNT=0
>>110 WHILE COUNT<3
>>120 PRINT "COUNT IS ";COUNT
>>130 COUNT=COUNT+1
>>140 ENDWHILE
>>150 PRINT "FINAL COUNT: ";COUNT
>>160 END
```

Using RUN we find this program produces

```
COUNT IS 0
COUNT IS 1
COUNT IS 2
FINAL COUNT: 3
```

When the program begins, COUNT is 0, so the expression COUNT<3 in the WHILE is true. The statements between WHILE and ENDWHILE are executed, and the program starts with the WHILE again to determine whether the expression is still true. The previous execution of the WHILE loop changed COUNT to 1 so the expression is true, and the loop is executed again. When COUNT becomes 3, the loop is not executed, and execution continues at line 150.

If line 100 set COUNT to 5 to begin with, then the WHILE loop would never be executed because the expression is false. Some care should be exercised in making sure that the logical expression in a WHILE is eventually met; otherwise, the program will loop indefinitely. The ability of the WHILE to prevent the execution of a loop when the logical expression is false makes the WHILE-ENDWHILE a particularly useful structure. Proper use can prevent numerous mistakes from creeping into our programs.

A statement pair that works something like WHILE-ENDWHILE is the REPEAT-UNTIL structure. Instead of checking a logical expression at the beginning of its loop, it checks a logical expression at the end of its loop. This causes the loop defined by REPEAT-UNTIL to always be executed at least once. In a sense, the REPEAT-UNTIL and WHILE-ENDWHILE are opposites of one another. The REPEAT-UNTIL program structure is illustrated by the example program

```
100 COUNT=0
110 REPEAT
120 PRINT "COUNT IS ";COUNT
130 COUNT=COUNT+1
140 UNTIL COUNT>=3
150 PRINT "FINAL COUNT: ";COUNT
160 END
```

This is the same program that was used in the WHILE example except that WHILE and ENDWHILE have been replaced by REPEAT and UNTIL. The loop is repeated until COUNT becomes 3 or greater; in this instance, the loop will terminate when COUNT is 3. Note that the condition tested is the opposite of that in the WHILE-ENDWHILE example. The program produces the same output as in the previous example

```
COUNT IS 0
COUNT IS 1
COUNT IS 2
FINAL COUNT: 3
```

The major difference is that if line 100 is changed to set COUNT to 3, the loop is executed exactly once before going to statement 150. Making this change in the preceding WHILE-ENDWHILE example would cause the WHILE loop not to be executed.

The warning in an earlier chapter that you should not transfer into the middle of a FOR-NEXT loop applies to the loop structures discussed in this chapter as well.

Review Questions and Exercises

1. If VALUE is 10 and TOT is 15, why is TOT still 15 when BASIC continues to the line following 400?

```
400 IF VALUE=24 THEN PRINT VALUE : TOT=TOT+1
```

2. END and ENDDO are not the same. What is the difference? What statement is an ENDDO used with?

3. What is wrong with the following use of ELSE

```
100 IF NUMBER=20 THEN COST=100
110 ELSE
120 COST=200
130 ENDDO
```

How do you correct line 100? Where does COST=100 belong?

Summary

DO-ENDDO
: Group similar statments together in a single unit.
Most effective when used with an IF statement.

IF
: All statements occurring after the THEN and on the same line as the IF are executed when the logical condition is true.

WHILE-ENDWHILE
: Defines a loop structure which is executed only while a logical condition is true.
The defined loop structure is not executed if the logical condition is false when the loop is encountered.
The logical expression is evaluated at the top of the loop.

REPEAT-UNTIL
: Defines a loop structure which is not executed until a logical condition is true.
The defined loop structure is executed once regardless of whether the logical condition is true or false when the loop is begun.
The logical expression is evaluated at the bottom of the loop.

CHAPTER 13
Writing Large Programs

As we increase our skills in programming and tackle larger applications, the progams we write become bigger and they often cannot be accommodated in the work area available. In this chapter, we will consider some features of BASIC that allow us to write programs that are larger the work area.

FIGURING OUT HOW MUCH SPACE IS AVAILABLE

We can get a very good idea of how much space is left in our work area by using the FRE function. Although this function does not actually require an argument, it nevertheless must be supplied one; so we write it as FRE(0) whenever we use it. The argument of 0 has no specific meaning to the function. This function simply returns to us the number of bytes or characters not occupied by our program. Let's try this function. Scratch your work area and enter

```
>>100 DIM TABLE(200)
>>110 PRINT "HELLO"
>>120 END
```

Now enter the following statement in immediate mode as

```
>>PRINT FRE(0)
```

BASIC will respond with 19712. (If you are working with 32K BASIC and have installed it with less than its full complement of features, you may get a significantly larger number. See the appendix on the installation of 32K BASIC. In any other case, you should get something close to 19712.) This is approximately the size, measured in bytes, of the work area that remains for your use. It is not a completely accurate number because BASIC allocates space for its own use in segments or blocks. Unused space in these blocks is not reflected by FRE. Furthermore, the DIM statement allocates space when it is executed; hence, we have not accounted for the space allocated to TABLE yet. Now enter RUN and then re-enter PRINT FRE(0) so as to produce the sequence

```
>>RUN

HELLO
***120 End***

>>PRINT FRE(0)

17920
```

This time, FRE tells us that the available space is 17920 bytes. Some of the difference is accounted for by the 200 elements allocated to TABLE, or about 1600 bytes. TABLE is a long floating point variable and each element occupies 8 bytes. The remaining difference is accounted for by the extra space BASIC needs to acquire when it begins execution.

We see from our exercise that there is a difference between the space used when the program is simply loaded into the work area and when it is executed.

CONTROLLING STORAGE—MODE CHANGES

As mentioned in earlier chapters, the amount of storage required to store a number depends upon the representation of the number. Integer numbers require two bytes, which is the least amount of storage; short floating numbers require four bytes; long floating numbers require eight bytes. With BASIC, we can specify the type of numbers that a variable is to store in several ways. Hence, we can refer to a variable as an integer, short float, or long float variable.

Integer variables may store integer valued numbers which range from -32768 to +32767. Contrast this with a statement in an earlier chapter, which stated that integer constants range from -10000 to +10000. Both statements are true. As stated in earlier chapters, a statement such as

```
LET INTNUM=30000
```

in which INTNUM is an integer variable, converts 30000, a short float number, to an integer.

The simplest way to specify the type of variables used by a program is with the IMODE, SFMODE, and LFMODE statements. They refer, respectively, to integer, short float, and long float mode. When you start writing a program in a work area, BASIC assumes that all variables and computations will be made in long float mode. Switching to another mode may save considerable space. However, some caution should be observed in that the space saved is mostly obtained in connection with numeric array variables. See a related discussion in the section concerning the use of INTEGER, SHORT, and LONG.

The mode may be changed by using one of these three statements. However, when one of the instructions is executed, the mode change does not come into effect until a RUN is executed afterwards. The use of these statements in your program creates a slightly awkward problem, however. How is it possible to use RUN after the execution of the mode change statement? A solution is given in the Structured BASIC Instruction Manual which is quite difficult to remember. A very effective and easy-to-remember solution is not to use the mode statements as part of the program, and to use them in immediate mode instead. We'll elaborate on this idea in the next few paragraphs.

Let's change the mode of a program from its default mode of long float to short float. Scratch your work area and enter the program

```
>>100 LET A=200.0
>>110 LET B=0.4
>>120 PRINT A,B
>>130 END
```

Verify that A and B are in long float mode by using LVAR, as in the sequence

```
>>LVAR
```

This produces

```
A LFP 0.0
B LFP 0.0
```

Now enter the SFMODE and LVAR as in the sequence

```
>>SFMODE
>>LVAR
```

This produces

```
A LFP 0.0
B LFP 0.0
```

We see that A and B are still in long float mode despite the use of SFMODE. Now enter RUN, which produces the following sequence

```
>>RUN

200.0                   0.4
***130 End***
```

Entering LVAR produces

```
A SFP 200.0
B SFP 0.4
```

All variables are now in short float mode.

If the program is to be saved, we need to be somewhat careful to make sure that we do not create some unwanted debris. It is probably advisable to place a STOP statement as the first line of code in the program when you are changing mode as we just described. When RUN is entered after using SFMODE, the program will stop immediately, but all variables will be in the new mode. Now, delete the STOP and use SAVE. This procedure minimizes the debris saved with the program file. If you actually let the program run without first stopping it, debris created by the execution would be saved with a subsequent SAVE.

MODE CHANGES FOR TRIGONOMETRIC FUNCTIONS

Two other mode change statements that we have not discussed are DEG and RAD. Normally, the arguments and return values of trigonometric functions are assumed to be in radians. DEG changes the mode to degrees instead of to radians, and RAD does just the opposite. RAD is the default mode for trigonometric functions. We do not recommend the use of DEG or RAD because most other high-level languages assume a radian mode with trigonometric functions. As a result of using them, you may lose some portability in transferring programs from one language to another, if you ever need to.

CONTROLLING STORAGE OF SPECIFIC VARIABLES

We can selectively specify the variable type for any variable instead of using the mode change statements to change all the variable types simultaneously. This is accomplished by using the SHORT, LONG, and INTEGER statements. It is preferable to use these statements over the mode change statements when trying to control storage. Some examples of their use are

```
INTEGER MONTH,DAY,YEAR,ID(5)
SHORT COST,TOTAL(100)
LONG MILEAGE(5,20),PIVOT,ANGLE
```

MONTH, DAY, and YEAR are declared as simple integer variables. ID is a one-dimensional array of integers. An attempt to store a number in these variables outside the range -32768 to 32767 will cause an overflow error. Integer variables occupy 2 bytes of storage. The array ID will occupy 12 bytes.

COST is declared as a simple short floating variable. TOTAL is a one-dimensional array of short floating numbers. Short floating variables occupy 4 bytes of storage. TOTAL, an array, will occupy 404 bytes.

MILEAGE is a two-dimensional array of long floating numbers. PIVOT and ANGLE are declared to be simple long floating variables. Long floating variables occupy 8 bytes of storage. MILEAGE will occupy 1008 bytes (6*21 elements of 8 bytes).

It is very important to realize that SHORT, LONG, and INTEGER are executable statements; they have no effect until they are executed. If we are working in long float mode, the default mode, a declaration of the sort

```
500 SHORT INCOME
```

does not cause INCOME to be reserved as 4 bytes of storage until line 500 is actually executed. If INCOME is used before line 500 is encountered, it will utilize 8 bytes of storage until line 500 is executed.

Actually, BASIC is somewhat deceptive in the amount of storage saved by employing SHORT, LONG, and INTEGER. Simple, non-array variables always occupy 8 bytes of storage, which is the storage required for a long floating variable. When any of these three statements is used, BASIC simply uses 2, 4, or 8 bytes of the 8-byte area for the variable. This is not true for array variables that are declared as one of the three types. Array elements occupy exactly the specified number of bytes for the type. This means that we can save space by declaring an array to a type requiring less space, but, for simple variables, no real space is gained.

To see that simple variables take 8 bytes, try using SHORT and LONG to declare the same variable successively and print the contents of the variable after each new declaration. For example

```
100 SHORT A
110 A=25
120 LONG A
130 PRINT A
140 SHORT A
150 PRINT A
160 END
```

The variable A will not change its value, and is maintained at a fixed location of 8 consecutive bytes in memory. Try some similar experiments with arrays.

MIXED MODE ARITHMETIC AGAIN

In an earlier chapter, we introduced the topic of mixed mode arithmetic. Now that SHORT, LONG, and INTEGER have been introduced, it is appropriate to discuss this topic further.

Some care should be taken when variables of different types are placed in the same arithmetic expression. Consider the program section

```
190 SHORT A,B
200 LONG RESULT
210 A=2.0: B=3.0
220 RESULT=A/B
```

The value placed in RESULT is 0.66666700000000. Short arithmetic operations are used to evaluate the expression, because A and B are short, and this results in a value of 0.666667. The result is then placed into the long floating variable RESULT. BASIC performs its arithmetic in the form of the longest variable encountered in an expression. If A or B had been long variables, the division operation would have used long arithmetic. The occurrence of division in an arithmetic expression involving mixed types of data and variables causes the majority of such quirks.

USING DELREM TO INCREASE SPACE

Probably the easiest thing we can do to reduce the size of our programs in the work area is to remove REM statements. This is simplified by the DELREM command, which deletes REM statements located at a specified line or range of lines. Like many other BASIC commands that involve line numbers, it has four forms, which are illustrated in the following examples

```
DELREM
DELREM 2500
DELREM 400,920
DELREM 900,
```

The first example deletes all REM statements; the second deletes only REM statements found at line 2500; the third deletes REM statements from 400 to 920; and the last deletes all from 900 to the end of the program. Once a REM statement is deleted by this command, it cannot be restored unless you re-enter it. Space is not actually freed by performing the command. Hidden debris and phantom line numbers remain. Space may actually be acquired by using the LIST-SCR-ENTER procedure for removing debris, which was discussed in an earlier chapter.

USING LIST AND ENTER TO COMBINE PROGRAMS

The LIST and ENTER commands can be used to help us construct larger programs. These commands can be used to produce programs that are of a reasonable size, and to eliminate the re-entry of statements necessary to create these programs.

As we learned in an earlier chapter, LIST may be used to place program lines in a file and they may be brought into the work area with ENTER. This procedure can be more selective than we previously discussed. In addition to the forms of LIST discussed, the following are example forms of LIST

```
LIST "SECTIONA.LIS",200
LIST "PARTFOUR.LIS",400,450
LIST SUBPROG$,2150,
```

The first example places just line 200 on a file SECTIONA.LIS. The final example places lines 400 through 450 on PARTFOUR.LIS. The final example places lines 2150 through the end of the program on a file specified by the string variable SUBPROG$.

The ENTER command places only the program lines it finds on a file into the work area, without disturbing any other lines in the work area. By using a combination of LIST, ENTER, and other commands, programs can be manipulated and constructed. Thus, without retyping statements, we can divide a program into smaller programs, or combine smaller programs into larger programs with the aid of ENTER and LIST.

Let's see how we might combine two programs into one program with the aid of these commands. Consider the following program, which is actually a

GOSUB subroutine

```
9000 *FINDIT
9010 FOR NBPOS=0 TO LEN(TEXT$)-1
9020 IF TEXT$(NBPOS,NBPOS)<>" " THEN GOTO EXIT
9030 NEXT NBPOS
9040 NBPOS=-1
9050 *EXIT: RETURN
```

This subroutine, FINDIT, scans each character of a string, TEXT$, and returns the position of the first non-blank character in TEXT$ in the variable NBPOS. If TEXT$ contains all blanks, NBPOS is set to -1. Suppose these lines are the only lines in the work area, and that we have saved them with LIST on the file FINDIT.LIS. (If they were part of some other statements in the work area, we could use LIST "FINDIT.LIS",9000,9050 to place them on a file.) Now suppose we want to combine the lines in FINDIT.LIS with the following program

```
100 DIM TEXT$(99)
110 INPUT "ENTER SOME TEXT:  ",TEXT$
120 GOSUB FINDIT
130 PRINT "FIRST NON-BLANK IS AT:  ";NBPOS
140 END
```

This program uses the FINDIT subroutine. Assuming no other lines are in the work area except 100 through 140, how can we combine these lines with the lines in FINDIT.LIS? We enter the command ENTER "FINDIT.LIS". The result is that we would now find the following lines in the work area

```
 100 DIM TEXT$(99)
 110 INPUT "ENTER SOME TEXT:  ",TEXT$
 120 GOSUB FINDIT
 130 PRINT "FIRST NON-BLANK IS AT:  ";NBPOS
 140 END
9000 *FINDIT
9010 FOR NBPOS=0 TO LEN(TEXT$)-1
9020 IF TEXT$(NBPOS,NBPOS)<>" " THEN GOTO EXIT
9030 NEXT NBPOS
9040 NBPOS=-1
9050 *EXIT: RETURN
```

Try your hand at combining FINDIT.LIS with the other statements using this process. These operations are quite useful, and they are well worth the effort made to understand them.

CHAINING PROGRAMS TOGETHER WITH RUN

RUN can be used as a program statement. It helps us to construct programs which are too large to fit in our work area. Often it is possible to simply break a program into two or more pieces and then execute one after the other to produce the desired application results. Rather than entering RUN at the completion of a program, we can have the program do this for us. We simply code RUN as the last executable statement in the program. For example, consider the following two programs, PROGA.BAS and PROGB.BAS

PROGA.BAS

```
100 PRINT "HELLO"
110 RUN "PROGB.BAS"
120 END
```

PROGB.BAS

```
100 PRINT "I AM FINE"
110 END
```

Entering RUN "PROGA.BAS" produces

```
HELLO
I AM FINE
```

After printing "HELLO" at line 100, program PROGA.BAS executes PROGB.BAS at line 110. The work area is cleared and PROGB.BAS is loaded and executed. If we were to use LIST, we would find PROGB.BAS in the work area. Any number of programs may be chained in this manner.

PASSING DATA BETWEEN PROGRAMS—OVERLAYS, FILES, AND COMMON

If we are faced with the problem of passing data from one program to another, we can do this by writing the data to a file and then reading in the next program. This is a very popular way of chaining programs together when a large amount of data are shared by each program.

Another approach to this problem is to use ENTER as a program statement to overlay or replace part of a previously executed program. Here are two programs that use this concept: OLAYA.BAS and OLAYB.LIS

Program OLAYA.BAS

```
200 LET WEIGHT=100
210 PRINT "BLOCK WEIGHS ";WEIGHT;" POUNDS"
220 ENTER "OLAYB.LIS"
300 END
```

Program OLAYB.LIS

```
200 GOTO OTHER
230 *OTHER:PRINT "WEIGHT IN OUNCES: ";WEIGHT*16
```

OLAYA.BAS is a program file created by SAVE, and OLAYB.LIS is a file created by LIST. Entering RUN "OLAYA.BAS" causes OLAYA.BAS to be loaded into the work area and executed. When line 220 is reached, ENTER causes OLAYB.LIS to be inserted into the work area, and the resulting work area looks like

```
200 GOTO OTHER
210 PRINT "BLOCK WEIGHS ";WEIGHT;" POUNDS"
220 ENTER "OLAYB.LIS"
230 *OTHER:PRINT "WEIGHT IN OUNCES: ";WEIGHT*16
300 END
```

Line 200 has been replaced with a GOTO and line 230 is inserted into the program. After executing the ENTER as indicated, BASIC begins execution with the first line in the program area. This is now a GOTO, which transfers control to the new program code at line 230. WEIGHT is still set to 100 because the work area was neither scratched nor reloaded. Hence, the entire execution sequence beginning with the RUN produces the output

```
BLOCK WEIGHS 100 POUNDS
WEIGHT IN OUNCES: 1600
***300 End***
```

While overlaying programs is a useful procedure, it can be troublesome when used excessively or without care. The debris that we referred to earlier can be collected quickly, and it is often difficult to visualize which statements are really in the work area after each overlay. This technique is better suited for small amounts of program overlaying.

Usually, when smaller amounts of data need to be passed between programs, it is done in a special program area called the Commmon area. This is a method that is employed by many programming languages and it is very widely used. It is particularly effective when many programs must share the same data.

Only array variables and strings may be passed between programs using this technique. This does not prevent us from passing data values of simple (non-array) variables between programs. By placing the value in an appropriate array variable, the value may be used by all programs using the Common area.

A Common area is defined by the use of DIM, INTEGER, SHORT, and LONG statements and the COMMON statement. The COMMON statement terminates the definition of a Common area. Only DIM, INTEGER, SHORT, and LONG statements which precede the COMMON statement are considered when defining the area. Among the INTEGER, SHORT, and LONG statements, any array variables found, plus those in DIM statements, define the Common area. The area is organized in the order that array variables occur in the statements.

Let's try an example to get a better understanding of how all of this works. Scratch your work area and enter the following program

```
>>100 DIM TABLE(2),NAME$(15)
>>110 SHORT COST(0),INCOME(1)
>>120 COMMON
>>130 TABLE(0)=0 : TABLE(1)=10 : TABLE(2)=20
>>140 COST(0)=33
>>150 INCOME(0)=1000.1
>>160 NAME$="DAVID"
>>170 RUN "PROGTWO.BAS"
>>180 END
```

Now enter

```
>>SAVE "PROGONE.BAS"
```

This saves the preceding program on the file PROGONE.BAS. Scratch your work area and enter the next program

```
>>100 DIM MYTABLE(2)
>>110 DIM MYNAME$(15)
>>120 SHORT COST(0),INCOME(1)
>>130 COMMON
>>140 PRINT "NAME IS: ";MYNAME$
>>150 PRINT "INCOME IS: ";INCOME(0)
>>160 END
```

Now enter

```
>>SAVE "PROGTWO.BAS"
```

This saves the preceding program on the file PROGTWO.BAS. Enter

```
>>RUN "PROGONE.BAS"
```

and you should see the result

```
NAME IS: DAVID
INCOME IS: 1000.1
```

If we study the two programs, we see that each defines the Common area with the following organization and description

Common Area

Program 1	Program 2	Bytes
TABLE(0)	MYTABLE(0)	8
TABLE(1)	MYTABLE(1)	8
TABLE(2)	MYTABLE(2)	8
NAME$	MYNAME$	16
COST(0)	COST(0)	4
INCOME(0)	INCOME(0)	4
INCOME(1)	INCOME(1)	4

The declarative statements have defined an area of exactly 52 bytes in each program that has been set aside for Common. The first 8 bytes are occupied by the zero-th element of a long floating variable array. In PROGONE.BAS, these bytes are referenced by TABLE(0), and in PROGTWO.BAS, these bytes are referenced by MYTABLE(0). Continuing down the table, it is easy to see the correspondences. There is no reason why the names in each program must be different when referencing the same portion of Common. This is illustrated by COST and INCOME, which are used in both programs.

It is easy to see how we got the results shown by the execution of the two programs. NAME$ is set to "DAVID" in PROGONE.BAS, but this corresponds to

MYNAME$ in PROGTWO.BAS. INCOME(0) is set to 1000.1 in PROGONE.BAS, but this corresponds to INCOME(0) in PROGTWO.BAS. When PROGTWO.BAS is run, the Common area is preserved. The values of the variables in it have not been changed.

When defining the Common area within a program, the area is structured on a variable-by-variable basis in whatever order the array variables are encountered. This is done without any regard for the structure that was imposed by any preceding programs. An advantage to this method is that we can sometimes restructure Common to our advantage. For example, in one program it may be convenient to reference a portion of Common containing a 20-character string as FULLNAME$. In another program, it may be more convenient to reference the same portion as two 10-character strings, FIRSTNAME$ and LASTNAME$. Here is a simple illustration

PROGRAM1.BAS

```
100 DIM FULLNAME$(19)
110 COMMON
120 FULLNAME$="WILLIAM   JOHNSON"
130 RUN "PROGRAM2.BAS"
140 END
```

PROGRAM2.BAS

```
100 DIM FIRSTNAME$(9),LASTNAME$(9)
110 COMMON
120 PRINT LASTNAME$,FIRSTNAME$
130 END
```

If we were to run PROGRAM1.BAS the result produced would be

```
JOHNSON         WILLIAM
```

The two names have been reversed in the output from PROGRAM2.

It is important to make sure that integer, short, long, and character variables share locations with variables of a similar type; otherwise, your programs may produce strange results. If two programs reference the same area with a variable of a different type, with one variable as short and the other variable as integer, one of the two programs will eventually run into trouble. The internal representation of each type is different and the number of bytes referenced is different.

There are several qualities of Common that are important to note. When RUN is used, variables which are in Common are not initialized. Variables outside of Common are initialized as described in an earlier chapter. Because BASIC does not initialize variables in Common, you should make it a practice to do so. Many subtle bugs can be avoided by carefully managing the initialization of variables. Another important point about Common is that successive programs which use Common must contain the COMMON statement; otherwise, the Common area previously established will be lost.

Review Questions and Exercises

1. What does FRE(0) do?

2. What does SFMODE do? When a mode change command is given, what additional statement must be executed before the mode changes are effective?

3. Which of the following two statements actually saves storage space?

```
SHORT COST,TOTAL,SALES
SHORT RAINFALL(200),TEMPERATURE(40)
```

4. Why is LET A=33.3/22 an example of mixed mode arithmetic?

5. What command is useful for combining or merging programs?

6. Can one program run another program? What statement permits us to chain programs?

7. Is it possible to use a simple (non-array) variable in Common?

8. Why is the length of Common 26 bytes in this example?

```
100 SHORT PRESSURE(1)
110 DIM ANGLE(0),WIDTH(0),MONTH$(5)
120 COMMON
```

9. Suppose you are using Common to transfer data between two programs, and an error occurs in the second program and it stops. You correct the problem, and want to continue from the start of the second program. What happens to the Common area if you use RUN? Why is it better to use GOTO to transfer to the first statement in this case?

Summary

FRE
A function which tells us the number of bytes remaining in our work area and program.

IMODE, SFMODE, LFMODE
Arithmetic mode commands which change the storage attributes of numeric variables.
The actual mode change of variables does not occur until a RUN is given, following the entry of a mode change command.

DEG,RAD
Commands which change the default mode of trigonometric functions between radians and degrees.

SHORT, LONG, INTEGER
Statements which declare the storage attributes of numeric variables.
Variables whose attributes are given with these statements are not changed by the mode change commands.
Attributes assigned by these statements are not effective until the statements are executed.
Variables may be dimensioned with these statements.

DELREM
A command which deletes REM statements.
Phantom line numbers are created.
Creates program debris.

COMMON
A statement which defines an area that is preserved when a new program is brought into the work area.
Variables declared in COMMON must be arrays or string variables.

ENTER-LIST
May be used to manipulate, combine, and merge programs into desirable forms.

OVERLAYS
Placement of program lines into the work area of an executing program with ENTER in such a way as to modify portions of the work area while executing the program.

RUN-CHAINING
RUN may be used as a statement to link or chain programs together.

CHAPTER 14
Procedures

In an earlier chapter we learned about subroutines in connection with the GOSUB. In this chapter we will learn about Procedures. They are closely related to the subroutine concept of the GOSUB and have a number of important uses.

ADVANTAGES OF PROCEDURES

Procedures provide a number of advantages over subroutines. First, they may define variables which are completely local to the Procedures. This means that the main program and a Procedure may contain the same variable names, but the names occupy different storage locations. This permits us to develop Procedures which may be used by more than one application, and be free from worrying about whether variable names conflict in such a way that we accidentally change a variable in a Procedure or vice versa.

Procedures may be used with arguments, and data may be passed back and forth between the program using the Procedure and the Procedure itself. By looking at the argument list, we can know exactly what constitutes input and output from the Procedure. This adds a little discipline to our programming, which is generally beneficial. Use of arguments also permits us to write recursive Procedures. These special Procedures are useful in only a few applications.

Perhaps the most important advantage of Procedures is that they give us another way of manipulating storage. Procedures may reside on a library outside of the program and be brought into the work area as they are needed. In a similar manner, they may be removed from a work area after they have been used, thus providing space for another Procedure or for additional storage. For example, we may have two Procedures in our library. One produces plots of data, and the other prints tables in some specified format. Our application may never need to have both of these Procedures in the work area at the same time. It can call them as needed and remove them after each use so that the minimum amount of storage is used. That is, they are never brought into the work area at the same time and so storage is not required for both.

FORM OF A PROCEDURE

A Procedure is defined by a PROCEDURE and ENDPROC statement. An example of a very simple Procedure is

```
1000 PROCEDURE .SUMHEAD
1010 PRINT
1020 PRINT "                    SUMMARY REPORT"
1030 PRINT
1040 ENDPROC
```

This Procedure simply prints a header for a report. The Procedure is named .SUMHEAD. All Procedure names begin with a period, and the name following the period follows the conventions used for naming numeric variables. When referring to a Procedure by name in the text, we will just use the name without the period. This Procedure is executed when a CALL statement referencing the Procedure is used. For example

```
500 CALL .SUMHEAD
```

The CALL statement is similar to the GOSUB.

Let's work with a slightly more complex Procedure. Scratch your work area and enter the program

```
>>100 DIM MYDATA(15)
>>110 DATA 5
```

```
>>120 DATA 14,19,30,17,22
>>130 READ N
>>140 FOR J=1 TO N
>>150 READ MYDATA(J)
>>160 NEXT J
>>170 CALL .SUMIT (N, MAT MYDATA; SUM, SUMSQ)
>>180 PRINT "SUM: ";SUM;" AND SUM OF SQUARES: ";SUMSQ
>>190 STOP
>>200 PROCEDURE .SUMIT (COUNT, MAT NUMBERS)
>>210 DIM NUMBERS(15)
>>220 S=0.0
>>230 SSQ=0.0
>>240 FOR K=1 TO COUNT
>>250 S=S+NUMBERS(K)
>>260 SSQ=SSQ+S*S
>>270 NEXT K
>>280 EXITPROC (S,SSQ)
>>290 ENDPROC
>>300 END
```

Make sure you put the space after SUMIT in lines 170 and 200, and the space after EXITPROC in line 280; otherwise, BASIC will generate a syntax error for those lines.

Enter RUN and you should produce

```
SUM: 102 SUM OF SQUARES: 22058
```

The Procedure SUMIT defined on lines 200 through 290 finds the sum and the sum of squares of numbers found in an array. The program found on lines 100 through 190 places five data values in the array MYDATA. The zero-th element of MYDATA is not used.

At line 170, SUMIT is called to compute the required sums that are printed in line 180. The call contains an argument list. A semicolon divides the list into two lists of variables. The first list contains the input variables N and the matrix MYDATA. The output results of SUMIT are placed in the output variables SUM and SUMSQ in the second list. As we will discover shortly, these two lists are not necessarily divided into separate lists for input and output variables. However, in many cases, we can think of the two lists as functioning in such a manner. An alternate name for argument lists is parameter lists, and we will use both terms interchangeably.

The PROCEDURE statement at 200 contains a list of input variable names used in the Procedure.

These are artificial, or dummy, names. They represent names that are used in forming the statements contained in the Procedure, and variables that are available may be affected through the CALL statement. For example, when called at line 170, the data for COUNT are taken from N, and the data for the array NUMBERS are taken from MYDATA. The names listed in the CALL and PROCEDURE must be in the same order as their intended use. CALL .SUMIT (MAT MYDATA, N; SUM, SUMSQ) would not be acceptable.

The EXITPROC statement at line 280 is used to return the output variable data to the output variables in the second list in the CALL. The value contained in S is placed in SUM, and the value contained in SSQ is placed in SUMSQ. A STOP is placed at line 190 to prevent the Procedure from being executed without a CALL.

Array and string variables may not be included in the output list of an EXITPROC. If array elements or strings are to be returned as output to the calling program through the argument list, they must be included in the first argument list. In this case, the array and the string variables must be explicitly declared with a DIM in the main program and the Procedure. Dimensions must match. For example, the program

```
100 DIM NUMBERS(5), MYSTRING$(10)
110 CALL .SIMPROC (MAT NUMBERS,MYSTRING$)
120 PRINT NUMBERS(4), MYSTRING$
130 STOP
140 PROCEDURE .SIMPROC (MAT MATRIX, ASTRING$)
150 DIM MATRIX(5), ASTRING$(10)
160 MATRIX(4)=22
170 ASTRING$="HELLO"
180 ENDPROC
190 END
```

produces

```
22              HELLO
```

If a single, non-array variable is to be used as output, its value must be output through the second list included in a CALL.

The STOP at line 130 is used to prevent the PROCEDURE statement from being executed without the CALL. A GOTO to skip around the Procedure would be another acceptable way to avoid executing the

Procedure without use of a CALL. It is not advisable to jump into the middle of a Procedure. BASIC will generate a stack error.

When using arrays in the argument list of the CALL and in the PROCEDURE, the prefix MAT must be used to identify the items as arrays. Character strings may be used in the argument lists for a Procedure.

Our SUMIT Procedure introduces all the procedural statements in BASIC except ERRPROC. ERRPROC is used to recover from errors which occur in a Procedure. It is discussed in the Structured BASIC Instruction Manual.

GLOBAL AND LOCAL VARIABLES

The SUMIT Procedure just discussed is fairly typical of Procedures. Our use of J inside SUMIT creates no problems because J is not used outside of SUMIT. If J had been used in the program outside of SUMIT, which is termed the calling program, a possible conflict could develop. The variable J could be changed in the Procedure. Its new value might cause a problem when the calling program continued execution from the point of the CALL. For example, if in the calling program, J contains a cost but was used in the Procedure as a loop variable, the cost might be inadvertently changed.

The variables J and K in our example are referred to as global variables because they refer to the same storage locations whether they are used inside or outside the Procedure. It is possible to make variables local to a Procedure with the LOCAL statement. That is, we may specify that variables have names and locations that are only known to the Procedure in which they are declared local. If we insert

```
205 LOCAL J
```

into the Procedure, we declare J as local. Now if J is used outside of SUMIT, it is effectively a different J than the J inside of SUMIT. This is of tremendous advantage when we write Procedures. We need not worry about choosing variable names in our Procedures that might be the same as those used in the calling program. Use of local variables prevents variables from getting changed accidentally in the calling program. Some care must be exercised when declaring array variables and string variables as local variables; the proper dimensions must be specified with a DIM statement or default values will be applied. Variables used in the parameter list of the PROCEDURE and EXITPROC statement are taken as local variables and need not be declared as such.

In order to emphasize the difference between local and global variables, consider the program

```
100 SCORE=12
100 POINTS=4
120 CALL .DOIT
130 PRINT "SCORE=";SCORE;" POINTS=";POINTS
140 STOP
150 PROCEDURE .DOIT
160 LOCAL SCORE
170 SCORE=200
180 POINTS=20
190 ENDPROC
```

This program produces the output

```
SCORE=12 POINTS=20
```

SCORE and POINTS are used inside and outside the Procedure. SCORE is used as a local variable inside the Procedure. It is a different SCORE inside. Hence, changing its value does not change the value of SCORE in the calling program. POINTS, however, is a global variable, and when its value is changed at line 180, it is changed in the calling program.

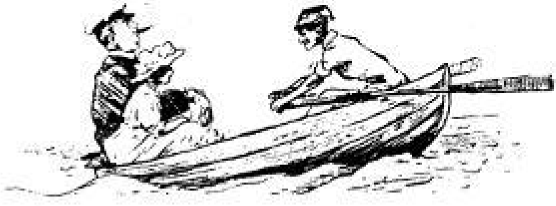


Figure 14.1 Just a little way to shore, folks.

As another example, consider the following program outline (indentations have been used to make some procedural relations clearer)

```
100 A=1.0
110 B=44.0
120 CALL .FIRST
130 CALL .HOUSE
    ...
170 STOP
200 PROCEDURE .FIRST
210   LOCAL A
220   A=4.0                                  FIRST
230   B=16.0
240   ENDPROC
300 PROCEDURE .HOUSE
310   LOCAL H
320   H=45.0
330   CALL .KITCHEN
340   EXITPROC
350   PROCEDURE .KITCHEN
360     A=15                                               HOUSE
370     H=200                               KITCHEN
380     EXITPROC
390     ENDPROC
400   ENDPROC
410 END
```

This program, which we will call MP (Main Program), uses three Procedures: FIRST, HOUSE, and KITCHEN. KITCHEN is nested within HOUSE. We can make the following statements about the variables A, B, and H

A in the MP is global to HOUSE and KITCHEN
A in FIRST is local to FIRST
B in the MP is global to FIRST, HOUSE, and KITCHEN
H in HOUSE is local to HOUSE
H is HOUSE is global to KITCHEN

We see that the terms local and global are relative.

It is not necessary to declare every variable in a Procedure as a local variable, but this is often a smart way to operate. If you do not, you run the risk of forgetting which variables are being used for this purpose when you use the Procedure in other programs, and this may lead to the kind of problems we just discussed. On the other hand, there is nothing wrong with using a global variable in a Procedure. This is a perfectly acceptable way of passing input or output values between a Procedure and the calling program.

If you plan to use your Procedures with the partitioning features of BASIC that are described later in this chapter, you will not have to worry quite as much about declaring variables local. This topic will be discussed later in connection with using Common and partitions.

RECURSION AND STACKING

The BASIC statement LOCAL, which permits us to define local variables, permits the stacking of variables, and this capability opens the possibility of writing recursive Procedures. However, stacking and recursion are concepts that most applications do not use, and they will not be discussed in any detail here. These concepts are discussed in the Structured BASIC Instruction Manual. We have included an interesting and fun example program which uses these concepts extensively in Appendix B.

MATCHING ARGUMENTS

We mentioned earlier that variables and data must be passed through the argument list to the Procedure in the correct order, as defined by the PROCEDURE and any EXITPROC statement. It is not necessary to match the storage attributes of the variables in the CALL and PROCEDURE statements. BASIC will take care of any mismatch in attributes. It will pass the correct values into the Procedure, and return values properly to output variables in the calling program.

When arrays or strings are involved in the arguments, it is very important to make certain that the dimensions and string lengths match. This is especially true when a two-dimensional array is used. When one-dimensional arrays or strings are used, make sure that the dimensions and lengths of variables in the Procedure are the same as the corresponding variables in the calling program.

As an aside for users who are familiar with languages like FORTRAN or PL/I, declaring variables appearing in the list of a PROCEDURE or EXITPROC statement as having particular storage attributes has no effect. INTEGER, SHORT, and LONG are executable statements and have no effect until they are actually executed.

BUILDING PROCEDURE LIBRARIES

As stated earlier, one of the advantages of Procedures is that they can be placed in libraries which are independent of the program. A program may bring Procedures into a work area when it needs them and remove them to conserve space as needed. This means that we can control the size of our programs during execution by selecting the minimum program configuration needed to solve our application problem. Using Procedures in this manner may be thought of as an alternative to the chaining and overlay methods discussed in a previous chapter.

A library consists of modules. A module is a single Procedure or a group of Procedures kept on a program file in SAVE form. A module must be created from a file consisting of Procedures that have been saved using the SAVE command.

There are two ways to create library modules. One method simply uses a file containing a single Procedure or several Procedures. This file must be created by using the SAVE command. In this form, the library consists of a single module. The other method requires the use of a BASIC program file called LIBBUILD.LIS which is included only with 32K BASIC software. Structured BASIC does not contain this program. The advantage of LIBBUILD is that several modules may be placed in a single library. We will discuss LIBBUILD in more detail in a later section.

Scratch your work area and enter each of the following two modules

Two Procedures for module 1

```
>>SCR
>>100 PROCEDURE .FATHER
>>110 PRINT "WE ARE USING FATHER"
>>120 ENDPROC
>>130 PROCEDURE .MOTHER
>>140 PRINT "WE ARE USING MOTHER"
```

```
>>150 CALL .CHILD
>>160 ENDPROC
>>SAVE "PARENTS.BAS"
```

A Procedure for module 2

```
>>SCR
>>100 PROCEDURE .CHILD
>>110 PRINT "CHILD HAS BEEN CALLED"
>>120 ENDPROC
>>SAVE "CHILDREN.BAS"
```

The first set of two Procedures, FATHER and MOTHER, is saved in a file called PARENTS.BAS, and the third Procedure, CHILD, is in a file called CHILDREN.BAS. We will use these two modules as individual libraries in the examples to follow in the next section.

PARTITIONS

Before using our libraries, PARENTS.BAS and CHILDREN.BAS, in a program, let's learn how BASIC views the work area for the purpose of working with Procedure libraries.

BASIC artificially divides the work area into 8 partitions, which are numbered from 0 to 7. When we normally enter a program, the program statements are placed in partition 0. When a Procedure is called from our program and it resides on a library file, the module containing the Procedure is placed in another partition, unless it is already in a partition. The Procedure is then executed and control is returned to the partition that made the call, which is partition 0 in this case. There are ways of specifying into which partition the next module brought into the work area should be placed. However, if we do not specify the partition, BASIC will place the module into the highest numbered partition that is available. A partition is available if it is not locked. Partition locking will be discussed later.

Now, scratch the work area and enter the following program, which makes use of the two libraries we built earlier.

```
>>100 LIBRARY "PARENTS.BAS"
>>110 CALL .FATHER
```

```
>>120 LIBRARY "CHILDREN.BAS"
>>130 CALL .MOTHER
>>140 END
```

Entering RUN produces

```
WE ARE USING FATHER
WE ARE USING MOTHER
CHILD HAS BEEN CALLED
```

The LIBRARY statement at line 100 tells BASIC that the next CALL which does not find the referenced Procedure should use the library PARENTS.BAS to find the Procedure. BASIC searches each partition in order from 0 to 7 to find a Procedure that is referenced in a CALL. When line 110 is executed, BASIC cannot find the Procedure FATHER in any partition. It finds it in module 1 of PARENTS.BAS and places it in partition 7. FATHER is executed. Line 120 opens a new library, CHILDREN.BAS. When a new library is opened, any previously open library file is closed. Simply using LIBRARY without a library reference closes any open library file. Upon executing line 130, BASIC finds that MOTHER is in partition 7 with FATHER and executes MOTHER. There was no need to reload the two Procedures of module 1. In the course of executing MOTHER, MOTHER calls CHILD, which is not in any partition. Again BASIC goes to the library; this time it goes to CHILDREN.BAS, and loads module 2 into partition 6, which is the next lowest available partition. CHILD is executed, producing the last of the three output lines. Control is returned to MOTHER, and MOTHER returns control to the program in partition 0. Line 140 terminates the program.

Use LIST to see what is left in your work area after executing our example program. Surprise! It is just what we entered a moment ago: lines 100 to 130. Where are the Procedures that were loaded from the libraries? These are in partitions 6 and 7. A USE command allows us to see these partitions. Enter

```
>>USE 7
>>LIST
```

This produces

```
100 Procedure .Father
110   Print"WE ARE USING FATHER"
```

```
120   Endproc
130 Procedure .Mother
140   Print"WE ARE USING MOTHER"
150   Call .CHILD
160   Endproc
```

Try experimenting with USE to look at partitions 6, 0, and 5. Try USE "CHILD". It finds the partition containing CHILD. What happens when you add or modify statements in the partition? BASIC permits you to make the changes.

An important point is that all variables in a partition are local to that partition. Another partition does not know the values of those variables, and cannot change them. However, within the partition, variables are global to one another between Procedures. That is, if two Procedures in the same partition have a similarly named variable, the variable is global. As a consequence, either Procedure can change its value.

The example just considered only illustrates the partitioning concept. As of yet, it may not be clear that we can use partitioning to our advantage in minimizing the storage required for a program. Before seeing how this can be accomplished, let us look at how partitions are locked.

LOCKING A PARTITION

In our preceding discussion, we mentioned that BASIC places a module in the next available partition. A partition is available when it is not locked. A partition becomes locked when it is involved in a nested call. This is exactly what happens when MOTHER is called in our example. The partition containing MOTHER is locked because it needs CHILD. The module containing CHILD is placed in the next available partition, which is 6. This form of locking is called an automatic lock. When control is transferred out of a partition that has been automatically locked, the partition is automatically unlocked. We may manually lock a partition and force BASIC not to use it with the LOCK statement. Examples of the two forms of the LOCK statement are

```
LOCK 4
LOCK "MOTHER"
```

The first form locks partition 4. The other form locks whatever partition contains the Procedure MOTHER. An UNLOCK command is available in the same two forms as the LOCK. It has the opposite effect of LOCK.

From what we have learned, it should be become clear that, through the proper use of general partitioning facilities and the locking mechanism, we can use partitions to minimize the storage occupied by our executing programs. For example, consider the use of a library called ANIMALS.BAS, which contains a single Procedure, HORSE, that uses no other Procedures. Suppose HORSE is called after MOTHER. Our program might look like

```
100 LIBRARY "PARENTS.BAS"
110 CALL .FATHER
120 LIBRARY "CHILDREN.BAS"
130 CALL .MOTHER
140 LIBRARY "ANIMALS.BAS"
150 CALL .HORSE
160 END
```

When line 160 is executed, the module containing HORSE will be loaded into partition 7 and executed. Partition 7 is not locked because we have not manually locked it, and the automatic locking mechanism does not apply. When BASIC searches for an unlocked partition, it begins at partition 7 as previously mentioned. Since partition 7 does not contain any Procedures HORSE needs or any which use HORSE, partition 7 is not automatically locked and it is used to load the module. We have reused partition 7 and replaced it with the space occupied by the former module with a new module. Partition 6 will be unchanged. If we wanted to free more space in the program we could clear partition 6 with a CLEAR statement, which we will learn about next.

SCRATCHING A PARTITION

When a module is loaded into a partition, it takes up space until it is removed. We may recover the space used by a partition by using the CLEAR statement. CLEAR acts like the SCR command on a partition. Examples of the two forms of the CLEAR statement are

```
CLEAR 3
CLEAR "TALLY"
```

The first form clears partition 3. The second form clears the partition containing the Procedure TALLY. Try experimenting with CLEAR and LVAR on each partition. Try printing the available space with FRE(0).

CLEAR is generally used to free space taken by a module residing in a partition. There is no need to clear a partition simply to have BASIC load a new module into it. When BASIC reuses an unlocked partition, it clears the partition first and then loads the module into it.

CREATING A LIBRARY WITH LIBBUILD

The LIBBUILD program is an interactive program which allows you to create libraries of Procedures. (It is only available in the 32K version of Structured BASIC, and is not available for the C-10.) It is a fairly large program which is kept on file in LIST form. It is so large that you may not be able to enter it into the delivered configuration of BASIC that you have been using. There are two ways to solve this problem. The first way is to reconfigure your BASIC interpreter by using the BASICGEN program delivered with your software. This is easily done by following the instructions in the appendix for configuring BASIC, and by configuring BASIC to a form which does not use the KSAM features. The second way is to remove all of the remark statements in the LIBBUILD program using the Cromemco SCREEN editor or WRITEMASTER programs. Use the edited version with BASIC. Keep a copy of the original.

In order to use LIBBUILD, we enter

```
>>ENTER "LIBBUILD.LIS"
>>RUN
```

and the program will respond with a menu of the form

```
32K STRUCTURED BASIC Library Utility
```

```
Select a function:

V -- View PROCEDURE names within a SAVEd program
I -- display Index of a library file
A -- Add a SAVEd program to a library file
D -- Delete a module from a library
C -- Create a new library file
Q -- Quit

function ? --->>>
```

The last line is a prompt to enter one of the function symbols: V, I, A, D, C, or Q.

LIBBUILD is fairly straightforward to use, and we will not discuss it in any further detail. However, if you want to try it out, enter the two modules that we created earlier, PARENTS.BAS and CHILDREN.BAS, as individual libraries into a library named EXAMLIB with LIBBUILD. When you are finished, use the I function of LIBBUILD, and you should obtain the index

```
  ( 2 k bytes used )

Module number 1
         .FATHER
         .MOTHER
Module number 2
         .CHILD
```

Note that in your directory you have created a file called EXAMLIB. It contains two modules with the above Procedures. Use the following program to get exactly the same results we obtained earlier using PARENTS.BAS and CHILDREN.BAS as individual libraries

```
100 LIBRARY "EXAMLIB"
110 CALL .FATHER
120 CALL .MOTHER
130 END
```

COMMON FOR PROCEDURES

Earlier, in connection with local variables, we mentioned that all variables in a partition are local to it. Since all variables in a Procedure are local to the partition, how do we pass data back and forth between partitions? One way was previously mentioned: use the argument list of a CALL statement. However, when we have a lot of data to share, it is easier to use a special Common area for Procedures. The BEGINCOMMON and ENDCOMMON statements are used to define such a COMMON, and we will call it method 2 for defining Common. In an earlier chapter, we discussed another way of defining Common with the COMMON statement, and it will be referred to as method 1. Method 1 does not give us a way of passing data through a Common area defined in a partition.

The Common area established by method 2 may be used by both partitions and programs. That is, method 2 permits partitions to share data. An entirely new program brought into the work area can also make use of the method 2 Common defined by any previous program in the work area. This second approach is quite similar to method 1. Usually, we think of method 2 as a way of sharing data between partitions. Method 2 is also similar to method 1 in that only array and string variables may be included in Common.

The main program, which is found in partition 0, reserves the space for Common method 2. A BEGINCOMMON is not needed in partition 0, and it will generate an error message if it is used. BASIC assumes that one is present before the first line of partition 0. An ENDCOMMON marks the point at which variables are no longer to be included in Common. Any array or string variable reference between the beginning of the main program and the ENDCOMMON causes these variables to be included in Common. Recall that in method 1, variables are placed in Common only if they are explicity defined in DIM, SHORT, LONG, and INTEGER statements. In method 2, any reference to array variables or strings places the variable in Common. The variables are placed in the method 2 Common area in the order they are referenced between the beginning of the main program and the ENDCOMMON. If an ENDCOMMON is not found in the main program, the first CALL made in the main program defines the ENDCOMMON. Once the space is reserved by BASIC when it encounters the ENDCOMMON, the size of the area is fixed unless it is extended by a subsequently executed program that is chained (by use of RUN) to the previous program. If an ENDCOMMON is placed as the first line of the main program, no Common area is reserved.

It is usually best to define all the variables in Common with DIM, SHORT, LONG, and INTEGER statements at the beginning of the main program, and they should be immediately followed by an ENDCOMMON. The placement of the ENDCOMMON in this way prevents variables from accidentally being included in Common. For example, if a PRINT statement with a reference to a string variable occurs before the ENDCOMMON, that string variable would reserve space in Common. In most instances, this would probably be undesirable.

In order that a Procedure may share the Common reserved in the main program, the Procedure must use the BEGINCOMMON and ENDCOMMON statements. If an ENDCOMMON is not used, the ENDPROC takes the place of ENDCOMMON. A BEGINCOMMON must be used if the Procedure needs access to the Common area. In a Procedure, the two statements do not reserve any space, but they do define how the Procedure views the Common area. The main program has already reserved the space. The effect of these two statements in the Procedure is simply to lay out or define how the Procedure views the area. In Procedures, variables are defined and allocated space in Common in the order they are encountered. If a Procedure attempts to use more space in Common than was previously established, BASIC will generate an error message. Restructuring of an area is possible in the same way as discussed for method 1.

In Procedures, it is wise to include only declaration statements such as DIM, INTEGER, SHORT, and LONG between the two statements. If another statement is included which contains a reference to a subscripted variable or string, the referenced variable is put into Common as well.

To illustrate Common method 2 for Procedures consider the program

Main Program - in Partition 0

```
100 DIM TABLE(4),NAME$(15)
110 ENDCOMMON
120 TABLE(1)=11.1
130 TABLE(4)=44.4
140 NAME$="GRETA"
150 CALL .SHOW
160 CALL .TELL
170 STOP
180 END
```

SHOW and TELL Procedures - in another partition

```
100 PROCEDURE .SHOW
120 BEGINCOMMON
130 DIM MATRIX(4),NAME$(15)
140 ENDCOMMON
150 FOR J=1 TO 4
160 PRINT MATRIX(J),
170 NEXT J
180 PRINT
190 PRINT "USER'S NAME: ";NAME$
200 ENDPROC
300 PROCEDURE .TELL
310 PRINT "NOTICE: ";TABLE(1),TABLE(4)
320 ENDPROC
```

When the main program is executed with RUN, it produces

```
11.1          0            0         44.4

USER'S NAME: GRETA
NOTICE: 0       0
```

TABLE and MATRIX match the same data in the Common area. TABLE is used in the main program, and MATRIX is used in the Procedure. NAME$ references the same data in either partition. J is a local variable to the partition containing the Procedure. A reference to J in the main program would be to a different J. Notice in particular in the Procedure TELL that a BEGINCOMMON and ENDCOMMON have not been used. This Procedure does not use Common. Hence the array TABLE is not part of Common, and, therefore, references to TABLE(1) and TABLE(4) are to an array outside of Common. The values of this array are all 0.

Review Questions and Exercises

1. Which character is used as the first character in a procedure name?

2. Why is it a good idea to precede a PROCEDURE statement with a STOP or GOTO statement?

3. What statement is used to execute a Procedure?

4. In the following program, why is AREA 15.0 when line 110 is executed? Is AREA a local or global variable?

```
100 CALL .RECTANGLE (3.0,5.0)
110 PRINT AREA,SIDE1,SIDE2
120 STOP
130 PROCEDURE .RECTANGLE (SIDE1,SIDE2)
140 AREA=SIDE1*SIDE2
150 ENDPROC
160 END
```

5. In question 4, why are SIDE1 and SIDE2 both 0 at line 110? What makes SIDE1 and SIDE2 local variables?

6. Instead of using AREA as a global variable in the program in question 4, we return the area of the rectangle through the argument list.

```
100 CALL .RECTANGLE (3.0,5.0;AREA)
110 PRINT AREA,AREA.RECT
120 STOP
130 PROCEDURE .RECTANGLE (SIDE1,SIDE2)
140 AREA.RECT=SIDE1*SIDE2
150 EXITPROC (AREA.RECT)
160 ENDPROC
170 END
```

Why is the value of AREA.RECT zero at line 110?

7. How many partitions does BASIC allow? What number corresponds to the first partition? Which partition contains the main program?

8. When are the Common areas defined by the use of COMMON and BEGINCOMMON-ENDCOMMON the same? Are they the same in the following main program and procedure, where the procedure is found in partition 7, for example?

Main Program (Partition 0)

```
100 SHORT ABC(19)
110 COMMON
120 ABC(5)=44.22
130 CALL .OUTSIDE
140 END
```

OUTSIDE Procedure (Partition 7)

```
100 PROCEDURE .OUTSIDE
110 BEGINCOMMON
120 SHORT ABC(19)
130 ENDCOMMON
140 PRINT "FROM OUTSIDE";ABC(5)
150 ENDPROC
```

What happens if COMMON is changed to ENDCOMMON?

Summary

Procedure
: Similar to a GOSUB subroutine except that it is more flexible.

PROCEDURE-ENDPROC
: Statements which define the beginning and the end of a Procedure.
When an argument list is included it refers to the first list in a CALL.

EXITPROC
: Use to return output values to simple (non-array) variables in a CALL statement.
Values are returned to the second argument list in the CALL which executed the Procedure.

CALL

Used to execute a Procedure.
May contain zero-, one-, or two-argument (parameter) lists.
The first and second lists are separated by a semicolon.
References to arrays must be preceded by MAT.
The second argument list may not contain array references.
The second argument list is used to return data to simple (non-array) variables.
The first list may contain only 1. simple input values and variables; 2. input array variables; and 3. output array variables.

ERRPROC

Used to recover from errors within a Procedure.

Local Variable

A variable with the same name as a previously used variable; its name is known only within the Procedure in which it is used.
Variables in a partition are local.
Variables named in a PROCEDURE or EXITPROC are local.

LOCAL

Explicitly defines a variable as local.
Stacks values in a way that can be used with recursive Procedures.

Global Variable

A variable whose name and value are known inside and outside a Procedure.

Partitions

A work area may be divided into eight partitions of arbitrary size which are numbered from 0 to 7.

Module

A collection of Procedures on a file created by a SAVE.

Library

Contains one or more modules which may be automatically brought into the work area by a reference to a Procedure that is contained in one of the modules.
The simplest library is single module.

LIBRARY

Opens a library which is searched when a Procedure reference cannot find the Procedure in any partition.

USE

Command which allows access to a partition.

LOCK

Permits a partition to be locked so that BASIC cannot load a new module into it.

CLEAR

Scratches a partition.

LIBBUILD Program

A special program that allows you to construct libraries containing several modules.

BEGINCOMMON-ENDCOMMON

Define a common area which may be used by Procedures occupying different partitions. May be used to establish a common that is shared by programs that are chained together with RUN.

APPENDIX A

Generating BASIC

When BASIC is purchased as a 32K version (occupying 32K of memory), it may be necessary to reduce the size of the BASIC interpreter by removing the capabilities that are not needed in your application. (This capability is not available in the C-10 personal computer version of BASIC, which is a 26K version of BASIC.) The extra space gained can be used to increase the size of programs, arrays, and strings. Many applications do not use, for example, the KSAM facilities of BASIC. By generating a version of BASIC without the KSAM facilities, we may gain 6500 bytes for program purposes.

BASIC is generated by a program called BASICGEN which is supplied with your BASIC software. In this appendix, we will give a brief description of how to generate BASIC with BASICGEN. A fuller treatment is contained in the 32K Structured BASIC Instruction Manual Addendum.

To generate BASIC, load a disk containing BASICGEN, and some other BASIC generation programs which will be mentioned shortly, onto a drive and enter

```
BASICGEN filename
```

where filename is the name of the file onto which the new version of BASIC is to be placed. Examples are

```
BASICGEN B:BASIC
BASICGEN NEWBASIC
BASICGEN RUNTIMEB
```

You do not need to specify an extension of COM. The following list of additional BASIC generation programs must be on the same disk with BASICGEN

B1.SBR	B2.SBR	B3.SBR
B4.SBR	B5.SBR	
C1.SBR	C2.SBR	C2A.SBR
C3.SBR	C3A.SBR	C4.SBR
C5.SBR	C6.SBR	C7.SBR
C8.SBR	C9.SBR	C9A.SBR
C9B.SBR		
BASLIB.SBR	SBASIC.SBR	SBASICIO.SBR

If these programs are not present along with BASICGEN.COM, BASIC will issue an error message and quit. If you are using CDOS, make sure that you are using the same master drive as the drive containing these programs.

When you execute BASICGEN, it will prompt you with a series of 10 questions to which you should respond with a Y or N for YES and NO, respectively. When you have completed all 10 questions, BASICGEN will then generate the new BASIC file. An abbreviated summary of the questions follows.

BASICGEN Question	Bytes Saved
Will this be an interactive version...?	5500
Do you wish KSAM file access capability?	6500
Do you wish the full text of error mess...?	1350
Do you wish editing capability?	770
Do you wish to include the PRINT USING...?	900
Do you wish to allow user defined functions?	190
Do you want to include the LOG and EXP...?	890
Do you wish to include the square root...?	175
Do you wish to include the trigonmetric...?	510
Do you wish to include the HEX, VALC,...?	400

Questions 8 and 9, concerning the square root and trigonmetric functions, will not be asked if you respond to question 6 (LOG and EXP) negatively.

APPENDIX B

An Example of Recursion

In the chapter on Procedures, we mentioned that we would include an example of writing a recursive procedure in BASIC. The example given here generates Hilbert curves of various orders. It is adapted from an algorithm in the book ALGORITHMS + DATA STRUCTURES = PROGRAMS by Nicklaus Wirth, Prentice-Hall, 1976. We offer the program with little comment and suggest that readers interested in the details see the cited book.

Hilbert curves of order 1 and 2 are shown here.

```
                 Hilbert Curves

          Order 1               Order 2

                           *********   *
                           *       *   *
                           *       *   *
                           *       *   *
          *****            *****   *****
          *                    *
          *                    *
          *                    *
          *****            *****   *****
                           *       *   *
                           *       *   *
                           *       *   *
                           *********   *
```

Notice that the order 2 curve consists of four basic parts; each part looks like an order 1 curve with a connector, rotated into a different position. A

Hilbert curve of order 3 uses a Hilbert curve of order 2 rotated into four similar patterns.

The program shown generates Hilbert curves of any order. It is set up to draw curves to order 4. The four procedures A, B, C, and D are used recursively to generate the four parts of the curve. The curves are placed in PLOT$ and displayed after each curve is generated. H0 governs the width of the plot and should be some power of 2, as indicated in the listing. Since PLOT$ is dependent upon H0, H0 should be chosen to keep PLOT$ from being too large to fit in the work area, and, at the same time, it should be a convenient page width. No attempt to erase the previous curve is made as each new curve is generated, so curves are superimposed over the same area as each new one is generated. The variable I does all the work as a recursive variable. It is very interesting to write the Plot procedure so that it places the curves on the screen as they are being generated. The result is a rather snake-like drawing.

A Program to Generate Hilbert Curves

```
100    Integer I,J,K,H,X,Y,X0,Y0,N,H0,Xold,Yold
110    Rem HILBERT CURVES FROM ORDER 1 TO N
120    Set 0,-1 : Rem Set BASIC page width
130    N=4 : Rem N is highest order curve
140    H0=64
145    Rem H0 is page width and H0=2**K for some K>=N
150    Dim Plot$(H0*H0)
160    I=0 : H=H0 : X0=Int(H/2) : Y0=X0
170      For J=1 To H0*H0 Step 8
180      Plot$(J,J+7)="        "
190      Next J
200      Repeat
210      Rem PLOT HILBERT CURVE OF ORDER K
220      K=K+1 : H=Int(H/2)
230      X0=X0+Int(H/2) : Y0=Y0+Int(H/2)
240      Xold=X0 : Yold=Y0
250      X=X0 : Y=Y0 : Call .Setplot
260      Call .A (K)
270      Call .Pout
280      Until K=N
290    Print
300    Stop
400  Procedure .Pout
410    Print
```

```
 420   Print"SUPERIMPOSED HILBERT CURVES TO ORDER ";K
 430   Print
 440     For J=1 To H0
 450     Print
 460     Print Plot$((J-1)*H0+1,J*H0);
 470     Next J
 480   Endproc
 500 Procedure .A (I)
 510   Integer I,Newi
 520   Newi=I : Local I : I=Newi
 530   If Newi>0 Then  Do
 540     Call .D (I-1) : X=X-H : Call .Plot
 550     Call .A (I-1) : Y=Y-H : Call .Plot
 560     Call .A (I-1) : X=X+H : Call .Plot
 570     Call .B (I-1)
 580     Enddo
 590   Endproc
 600 Procedure .B (I)
 610   Integer I,Newi
 620   Newi=I : Local I : I=Newi
 630   If Newi>0 Then  Do
 640     Call .C (I-1) : Y=Y+H : Call .Plot
 650     Call .B (I-1) : X=X+H : Call .Plot
 660     Call .B (I-1) : Y=Y-H : Call .Plot
 670     Call .A (I-1)
 680     Enddo
 690   Endproc
 700 Procedure .C (I)
 710   Integer I,Newi
 720   Newi=I : Local I : I=Newi
 730   If Newi>0 Then  Do
 740     Call .B (I-1) : X=X+H : Call .Plot
 750     Call .C (I-1) : Y=Y+H : Call .Plot
 760     Call .C (I-1) : X=X-H : Call .Plot
 770     Call .D (I-1)
 780     Enddo
 790   Endproc
 800 Procedure .D (I)
 810   Integer I,Newi
 820   Newi=I : Local I : I=Newi
 830   If Newi>0 Then  Do
 840     Call .A (I-1) : Y=Y-H : Call .Plot
 850     Call .D (I-1) : X=X-H : Call .Plot
 860     Call .D (I-1) : Y=Y+H : Call .Plot
 870     Call .C (I-1)
 880     Enddo
 890   Endproc
1000 Procedure .Plot
```

```
1010    Dim Plotchar$(8) : Plotchar$=" +.*X@#%!"
1020    Integer Size
1030    Local J
1040    If Xold<>X Then  Do
1050      Size=Sgn(X-Xold)
1060        For J=Xold To X Step Size
1070        Plot$((Y-1)*H0+J,(Y-1)*H0+J)=Plotchar$(K,K)
1080        Next J
1090      Else
1100      Size=Sgn(Y-Yold)
1110        For J=Yold To Y Step Size
1120        Plot$((J-1)*H0+X,(J-1)*H0+X)=Plotchar$(K,K)
1130        Next J
1140      Enddo
1150    Xold=X : Yold=Y
1160    Endproc
1170 Procedure .Setplot
1180    Xold=X : Yold=Y
1190    Endproc
```

APPENDIX **C**

BASIC Error Messages

All the error messages that BASIC issues are listed except those for KSAM. When the text of a message seems incomplete, we have added comments in parentheses. When you try to interpret an error message at the terminal, list the statement and compare it with the message.

Fatal Errors

No.	Message (Meaning*)
1	Syntax
2	Using Syntax (PRINT USING format incorrect)
3	Number of Arguments (No. args in fun. call incorrect)
5	Illegal Statement
6	Print Item Size (Exceeded page width - See SET)
7	Too Many Gosubs
8	Expression Too Complex (E.g., too many parentheses)
9	Return, No GOSUB Active
10	Next Without For
12	User Function not Defined
13	Invalid Dimensions given
14	Goto or Gosub Non-existent Line
15	Subscript Value (Subscript value out of range)
16	Number of Subscripts (Wrong number of subscripts)
17	Duplicate definition of label or function
19	Use of undefined line label
20	Run Time Stack Improperly Nested (E.g., WHILE-ENDO)

* Descriptions in () are added to clarify the message, and are not part of the message.

Fatal Errors (Continued)

No.	Message (Meaning)
21	Attempt to Go Back to Altered or Deleted Line
22	DIM Would Overflow Top of ... COMMON (Already reserved)
23	Bad Begincommon/Endcommon Sequence
24	String/Numeric Expression Mismatch (E.g., NUM="HI")
71	No Such Procedure Available
72	Bad Arguments to a Procedure CALL/ENDPROC
73	No Free Partitions to Load Procedure/Module into
74	Invalid Procedure Library
99	FEATURE NOT IMPLEMENTED
101	End of Statement/End of Line (See Cromemco)
102	Out of Memory

Non-Fatal Errors (User Trappable)

No.	Message (Meaning)
128	File Not Found (file not in directory)
129	Filename (Illegal file name)
130	Invalid Command for Device
131	File Already Open
132	File Not Open
133	File Number (Invalid range for file number)
134	Cannot Open File (or does not exist)
135	No File Space
136	File Mode Error (E.g, writing on a read only file)
137	Cannot Create File (File already exists)
138	File Read: No Data (Past last sector)
139	File Write (Writing to protected disk or file)
140	File Position/Status (Record no. out of range)
141	No Channels Available (Too many files open)
142	Cannot Close File (File missing from disk)
143-190	(KSAM errors - KSAM not discussed in this book)
200	Invalid Hex Number
201	Integer Overflow
202	Function Argument Value (E.g., SQR(-2.4))
203	Invalid Input (E.g., INPUT string into numeric)
204	Input (Too many items for INPUT)
205	Not Dimensioned
206	No Data Statement (Not enough data in DATA stmts)
207	Data Type Mismatch (READ reading wrong data type)
208	Number Size (Number too big or small for BASIC)
209	Line Length (ENTER file line more than 132 chars)
210	Input Timeout (See SET statement)
250	Overflow/Underflow
251	Errproc Return from a Procedure

APPENDIX **D**

ASCII Character Codes

Dec	Hex	Code	Dec	Hex	Code	Dec	Hex	Code	Dec	Hex	Code
000	00	NUL	032	20	Space	064	40	@	096	60	`
001	01	SOH	033	21	!	065	41	A	097	61	a
002	02	STX	034	22	"	066	42	B	098	62	b
003	03	ETX	035	23	#	067	43	C	099	63	c
004	04	EOT	036	24	$	068	44	D	100	64	d
005	05	ENQ	037	25	%	069	45	E	101	65	e
006	06	ACK	038	26	&	070	46	F	102	66	f
007	07	BEL	039	27	'	071	47	G	103	67	g
008	08	BS	040	28	(	072	48	H	104	68	h
009	09	HT	041	29	)	073	49	I	105	69	i
010	0A	LF	042	2A	*	074	4A	J	106	6A	j
011	0B	VT	043	2B	+	075	4B	K	107	6B	k
012	0C	FF	044	2C	,	076	4C	L	108	6C	l
013	0D	CR	045	2D	-	077	4D	M	109	6D	m
014	0E	SO	046	2E	.	078	4E	N	110	6E	n
015	0F	SI	047	2F	/	079	4F	O	111	6F	o
016	10	DLE	048	30	0	080	50	P	112	70	p
017	11	DC1	049	31	1	081	51	Q	113	71	q
018	12	DC2	050	32	2	082	52	R	114	72	r
019	13	DC3	051	33	3	083	53	S	115	73	s
020	14	DC4	052	34	4	084	54	T	116	74	t
021	15	NAK	053	35	5	085	55	U	117	75	u
022	16	SYN	054	36	6	086	56	V	118	76	v
023	17	ETB	055	37	7	087	57	W	119	77	w
024	18	CAN	056	38	8	088	58	X	120	78	x
025	19	EM	057	39	9	089	59	Y	121	79	y
026	1A	SUB	058	3A	:	090	5A	Z	122	7A	z
027	1B	ESC	059	3B	;	091	5B	[	123	7B	{
028	1C	FS	060	3C	<	092	5C	\	124	7C	|
029	1D	GS	061	3D	=	093	5D	]	125	7D	}
030	1E	RS	062	3E	>	094	5E	^	126	7E	~
031	1F	US	063	3F	?	095	5F	Under	127	7F	DEL

FF=form feed, ESC=escape, LF=line feed, CR=carriage return
DEL=rubout, under=underscore.

Control Codes

Dec	Hex	Cntrl	Dec	Hex	Cntrl	Dec	Hex	Cntrl	Dec	Hex	Cntrl
000	00	@	008	08	H	016	10	P	024	18	X
001	01	A	009	09	I	017	11	Q	025	19	Y
002	02	B	010	0A	J	018	12	R	026	1A	Z
003	03	C	011	0B	K	019	13	S	027	1B	[
004	04	D	012	0C	L	020	14	T	028	1C	\
005	05	E	013	0D	M	021	15	U	029	1D	]
006	06	F	014	0E	N	022	16	V	030	1E	^
007	07	G	015	0F	O	023	17	W	031	1F	Under

Index

The letter S before a number indicates the entry corresponds to a summary at the end of a chapter.

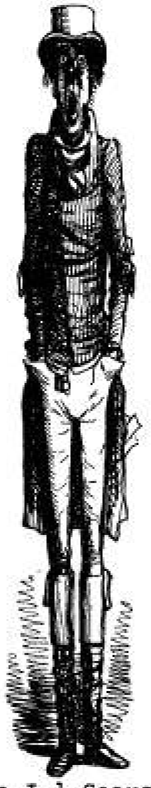


Figure I-1 Search me!

Figure I-2 At last!

AN INTRODUCTION TO STRUCTURED BASIC FOR THE CROMEMCO C-10
by Wayne T. Watson

This book assumes no prior programming experience and stresses interaction with the microcomputer. It is an introduction to the Cromemco™ Structured BASIC, or Structured BASIC, language, which is available for use on Cromemco microcomputers. All programs and program examples included in the book have been run to ensure that they are accurate. These programs should be applicable to the reader's own computer needs. The first 10 chapters provide a foundation to the most commonly used concepts in BASIC, and the last 4 chapters offer more advanced material on files and programming structures.

About the Author

Wayne T. Watson is the owner/president of The Software Hill, a computer software company specializing in statistical and business forecasting applications. One of its first products is IFDAS—Interactive Forecasting and Data Analysis System, a statistical software package. He has been active in the area of software reliability and has held positions concerned with software development for the last ten years. His articles have appeared in *Communications of the ACM* and *The American Statistician*.

MACMILLAN PUBLISHING COMPANY
866 Third Avenue, New York, N.Y. 10022

ISBN 0-02-424580-1