decsystem10

BASIC CONVERSATIONAL LANGUAGE MANUAL

This manual reflects the software as of version 17D.

Additional copies of this manual may be ordered from: Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754 Order code: DEC-10-LBLMA-A-D

digital equipment corporation · maynard. massachusetts

1st Printing December 1968
2nd Printing (Rev) May 1969
3rd Printing September 1969
4th Printing (Rev) January 1970
5th Printing (Rev) September 1970
6th Printing (Rev) August 1971
7th Printing (Rev) February 1972
Update Pages May 1972
Update Pages September 1972
Update Pages January 1974
Update Pages March 1974

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

ILLUSTRATION

TABLES

PREFACE

WHY BASIC? BASIC is a problem-solving language that is easy to learn and conversational, and has wide application in the scientific, business, and educational communities. It can be used to solve both simple and complex mathematical problems from the user's Teletype® and is particularly suited for time-sharing.

In writing a computer program, it is necessary to use a language or vocabulary that the computer recognizes. Many computer languages are currently in use, but BASIC is one of the simplest of these because of the small number of clearly understandable and readily learned commands that are required, its easy application in solving problems, and its practicality in an evolving educational environment.

BASIC is similar to other programming languages in many respects; and is aimed at facilitating communication between the user and the computer in a time-sharing system. As with most programming languages, BASIC is divided into two sections:

a. Elementary statements that the user must know to write simple programs, and
b. Advanced techniques needed to efficiently organize complicated problems.

As a BASIC user, you type in a computational procedure as a series of numbered statements by using common English syntax and familiar mathematical notation. You can solve almost any problem by spending an hour or so learning the necessary elementary commands. After becoming more experienced, you can add the advanced techniques needed to perform more intricate manipulations and to express your problem more efficiently and concisely. Once you have entered your statements via the Teletype, simply type in RUN or RUNNH. These commands initiate the execution of your program and return your results almost instantaneously.

SPECIAL FEATURES OF BASIC - BASIC incorporates the following special features:

a. Matrix Computations - A special set of 13 commands designed exclusively for performing matrix computations.

®Teletype is the registered trademark of Teletype Corporation.

b. Alphanumeric Information Handling - Single alphabetic or alphanumeric strings or vectors can be read, printed, and defined in LET and IF...THEN statements. Individual characters within these strings can be easily accessed by the user. Conversion can be performed between characters and their ASCII equivalents. Tests can be made for alphabetic order.

c. Program Control and Storage Facilities - Programs or data files can be stored on or retrieved from various devices (disk, DECtape, card reader, card punch, high-speed paper-tape reader, high-speed paper-tape punch and line printer). The user can also input programs or data files from the low-speed Teletype paper-tape reader, and output them to the low-speed Teletype paper-tape punch.

d. Program Editing Facilities - An existing program or data file can be edited by adding or deleting lines, by renaming it, or by resequencing the line numbers. The user can combine two programs or data files into one and request either a listing of all or part of it on the Teletype or a listing of all of it on the high-speed line printer.

e. Formatting of Output - Controlled formatting of Teletype output includes tabbing, spacing, and printing columnar headings.

f. Documentation and Debugging Aids - Documenting programs by the insertion of remarks within procedures enables recall of needed information at some later date and is invaluable in situations in which the program is shared by other users. Debugging of programs is aided by the typeout of meaningful diagnostic messages which pinpoint syntactical and logical errors detected during execution.

CHAPTER 1
INTRODUCTION

This chapter introduces the user to PDP-10 BASIC and to its restrictions and characteristics. The best introduction lies in beginning with a BASIC program and discussing each step completely.

1.1 EXAMPLE OF A BASIC PROGRAM

The following example is a complete BASIC program, named LINEAR, that can be used to solve a system of two simultaneous linear equations in two variables

$$ax + by = c$$
$$dx + ey = f$$

and then used to solve two different systems, each differing from the above system only in the constants c and f. If ae - bd is not equal to 0, this system can be solved to find that

$$x = \frac{ce - bf}{ae - bd} \qquad \text{and} \qquad y = \frac{af - cd}{ae - bd}$$

If ae - bd = 0, there is either no solution or there are many, but there is no unique solution. Study this example carefully and then read the commentary and explanation. (In most cases the purpose of each line in the program is self-evident.)

```
1Ø      READ A,B,D,E↵
15      LET G=A*E-B*D↵
2Ø      IF G=Ø THEN 65↵
3Ø      READ C,F↵
37      LET X=(C*E-B*F)/G↵
42      LET Y=(A*F-C*D)/G↵
55      PRINT X,Y↵
6Ø      GO TO 3Ø↵
65      PRINT "NO UNIQUE SOLUTION"↵
7Ø      DATA 1,2,4↵
8Ø      DATA 2,-7,5↵
85      DATA 1,3,4,-7↵
9Ø      END↵
```

NOTE

All statements are terminated by pressing the RETURN key (represented in this text by the symbol ↵). The RETURN key echoes as a carriage return, line feed.

1.2 DISCUSSION OF THE PROGRAM

Each line of the program begins with a line number of 1 to 5 digits that serves to identify the line as a statement. A program is made up of statements. Line numbers serve to specify the order in which these statements are to be performed. Before the program is run, BASIC sorts out and edits the program, putting the statements into the orders specified by their line numbers; thus, the program statements can be typed in any order, as long as each statement is prefixed with a line number indicating its proper sequence in the order of execution. Each statement starts after its line number with an English word which denotes the type of statement. Unlike statements, commands are not preceded by line numbers and are executed immediately after they are typed in. (Refer to Chapter 9 for a further description of commands.) Spaces and tabs have no significance in BASIC programs or commands, except in messages which are printed out, as in line number 65 above. Thus, spaces or tabs may, but need not be, used to modify a program and make it more readable.

With this preface, the above example can be followed through step-by-step.

```
10      READ A,B,D,E
```

The first statement, 10, is a READ statement and must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing a program, it causes the variables listed after the READ to be given values according to the next available numbers in the DATA statements. In this example, we read A in statement 10 and assign the value 1 to it from statement 70 and, similarly, with B and 2, and with D and 4. At this point, the available data in statement 70 has been exhausted, but there is more in statement 80, and we pick up from it the value 2 to be assigned to E.

```
15      LET G=A*E-B*D
```

Next, in statement 15, which is a LET statement, a formula is to be evaluated. [The asterisk (*) is used to denote multiplication.] In this statement, we compute the value of AE - BD, and call the result G. In general, a LET statement directs the computer to set a variable equal to the formula on the right side of the equal sign.

```
20      IF G=0 THEN 65
```

If G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 20, if G is equal to zero.

```
65      PRINT "NO UNIQUE SOLUTION"
70      DATA 1,2,4
80      DATA 2,-7,5
85      DATA 1,3,4,-7
90      END
```

If the computer discovers a "yes" answer to the question, it is directed to go to line 65, where it prints NO UNIQUE SOLUTION. Since DATA statements are not executable statements, the computer then goes to line 90 which tells it to END the program.

```
30      READ C,F
```

If the answer is "no" to the question "Is G equal to zero?", the computer goes to line 30. The computer is now directed to read the next two entries, -7 and 5, from the DATA statements (both are in statement 80) and to assign them to C and F, respectively. The computer is now ready to solve the system

$$\begin{aligned} x + 2y &= -7 \\ 4x + 2y &= 5 \end{aligned}$$

```
37      LET X=(C*E-B*F)/G
42      LET Y=(A*F-C*D)/G
```

In statements 37 and 42, we instruct the computer to compute the value of X and Y according to the formulas provided, using parentheses to indicate that C*E - B*F is calculated before the result is divided by G.

```
55      PRINT X,Y
60      GO TO 30
```

The computer prints the two values X and Y in line 55. Having done this, it moves on to line 60 where it is reverted to line 30. With additional numbers in the DATA statements, the computer is told in line 30 to take the next one and assign it to C, and the one after that to F. Thus,

$$\begin{aligned} x + 2y &= 1 \\ 4x + 2y &= 3 \end{aligned}$$

As before, it finds the solutions in 37 and 42, prints them out in 55, and then is directed in 60 to revert to 30.

In line 30, the computer reads two more values, 4 and -7, which it finds in line 85. It then proceeds to solve the system

$$\begin{aligned} x + 2y &= 4 \\ 4x + 2y &= -7 \end{aligned}$$

and print out the solutions. Since there are no more pairs of numbers in the DATA statement available for C and F, the computer prints OUT OF DATA IN 30 and stops.

If line number 55 (PRINT X, Y) had been omitted, the computer would have solved the three systems and then told us when it was out of data. If we had omitted line 20, and G were equal to zero, the computer would print DIVISION BY ZERO IN 37 and DIVISION BY ZERO IN 42. Had we omitted statement 60 (GO TO 30), the computer would have solved the first system, printed out the values of X and Y, and then gone to line 65, where it would be directed to print NO UNIQUE SOLUTION.

The particular choice of line numbers is arbitrary as long as the statements are numbered in the order the machine is to follow. We would normally number the statements 10, 20, 30, ..., 130, so that later we can insert additional statements. Thus, if we find that we have omitted two statements between those numbered 40 and 50, we can give them any two numbers between 40 and 50 -- say 44 and 46. Regarding DATA statements, we need only put the numbers in the order that we want them read (the first for A, the second for B, the third for D, the fourth for E, the fifth for C, the sixth for F, the seventh for the next C, etc.). In place of the three statements numbered 70, 80, and 85, we could have written the statement:

```
75      DATA 1,2,4,2,-7,5,1,3,4,-7
```

or, more naturally,

```
70      DATA 1,2,4,2
75      DATA -7,5
80      DATA 1,3
85      DATA 4,-7
```

to indicate that the coefficients appear in the first data statement and the various pairs of right-hand constants appear in the subsequent statements.

The program and the resulting run is shown below as it appears on the Teletype.

```
10      READ A,B,D,E
15      LET G=A*E-B*D
20       IF G=0 THEN 65
30      READ C,F
37      LET X=(C*E-B*F)/G
42      LET Y=(A*F-C*D)/G
55      PRINT X,Y
60      GO TO 30
65      PRINT "NO UNIQUE SOLUTION"
70      DATA 1,2,4
80      DATA 2,-7,5
85      DATA 1,3,4,-7
90      END
RUN
```

(continued on next page)

```
LINEAR          11:03           19-OCT-69

 4              -5.50000
 0.666667        0.166667
-3.66667         3.83333
OUT OF DATA IN 30
TIME:  0.10 SECS.
```

NOTE

Remember to terminate all statements by pressing the RETURN key.

After typing the program, we type the command RUN and press the RETURN key to direct the computer to execute the program. Note that the computer, before printing out the answers, printed the name LINEAR which we gave to the problem (refer to Paragraph 4.1) and the time and date of the computation. The message OUT OF DATA IN 30, may be ignored here. However, in some instances, it indicates an error in the program. The TIME message, printed out at the end of execution, indicates the compile and execute time used by the program; this time is slightly dependent upon other jobs being processed by the computer and consequently will not be exactly the same each time the same program is run.

1.3 FUNDAMENTAL CONCEPTS OF BASIC

BASIC can perform many operations such as adding, subtracting, multiplying, dividing, extracting square roots, raising a number to a power, and finding the sine of an angle measured in radians.

1.3.1 Arithmetic Operations

The computer performs its primary function (that of computation) by evaluating formulas similar to those used in standard mathematical calculation, with the exception that all BASIC formulas must be written on a single line. The following operators can be used to write a formula.

Operator	Example	Meaning
+	A + B	add B to A
+	+A	A itself
-	A - B	subtract B from A
-	-A	make A negative
*	A * B	multiply B by A
/	A / B	divide A by B
↑	X ↑ 2	find X^2 ⎧ the symbols ↑ and ** have
**	X**2	find X^2 ⎩ the same meaning

If we type A + B * C ↑ D, the computer first raises C to the power D, multiplies this result by B, and then adds the resulting product to A. We must use parentheses to indicate any other order. For example, if it is the product of B and C that we want raised to the power D, we must write A + (B * C) ↑ D; or if we want to multiply A + B by C to the power D, we write (A + B) * C ↑ D. We could add A to B, multiply their sum by C, and raise the product to the power D by writing ((A + B) * C) ↑ D. The order of precedence is summarized in the following rules.

a. The formula inside parentheses is evaluated before the parenthesized quantity is used in computations.

b. Normally two operators cannot be contiguous. However the operators + and - can follow the operators *, /, **, or ↑ (e.g., *-). In such a case, the + or - takes precedence over its leading *, /, **, or ↑. Otherwise:

c. In the absence of parentheses in a formula, ** and ↑ take precedence over * and /, which take precedence over + and -.

d. In the absence of parentheses in a formula whose only operators are * and /, BASIC performs the operations from left to right, in the order that they are read.

e. In the absence of parentheses in a formula whose only operators are + and -, BASIC performs the operations from left to right, in the order that they are read.

The rules tell us that the computer, faced with A - B - C, (as usual) subtracts B from A, and then C from their difference; faced with A/B/C, it divides A by B, and that quotient by C. Given A ↑ B ↑ C, the computer raises the number A to the power B and takes the resulting number and raises it to the power C. If there is any question about the precedence, put in more parentheses to eliminate possible ambiguities.

1.3.2 Mathematical Functions

In addition to these five arithmetic operations, BASIC can evaluate certain mathematical functions. These functions are given special three-letter English names.

Function		Interpretation	
SIN	(X)	Find the sine of X	X interpreted as an angle measured in radians
COS	(X)	Find the cosine of X	
TAN	(X)	Find the tangent of X	
COT	(X)	Find the cotangent of X	
ATN	(X)	Find the arctangent of X	
EXP	(X)	Find e raised to the X power (e^X)	
LOG	(X), or LN(X), or LOGE(X)	Find the natural logarithm of X (log to the base e)	X interpreted as a number
ABS	(X)	Find the absolute value of X (| X |)	
SQR	(X) or SQRT(X)	Find the square root of X ($\sqrt{X}$)	
CLOG	(X) or LOG10(X)	Find the common logarithm of X (log to the base 10)	

Other functions are also available in BASIC. They are described in Chapters 5 (INT, RND, SGN, TIM), 7 (NUM, DET), 8 (string functions), and 10 (LOC, LOF). In place of X, we may substitute any formula or number in parentheses following any of these functions. For example, we may ask the computer to find

$\sqrt{4 + X^3}$ by writing SQR (4 + X ↑ 3), or the arctangent of $3X - 2e^X + 8$ by writing ATN (3 * X - 2 * EXP (X) + 8). If the above value of $\left(\frac{5}{6}\right)^{17}$ is needed, the two-line program can be written:

```
10      PRINT(5/6)↑17
20      END
```

and the computer finds the decimal form of this number and prints it out.

1.3.3 Numbers

A number may be positive or negative and it may contain up to eight digits, but it must be expressed in decimal form (i.e., 2, -3.675, 12345678, -.98765432, and 483.4156). The following are not numbers in BASIC: 14/3 and SQR(7). The computer can find the decimal expansion of 14/3 or SQR(7), but we may not include either in a list of DATA. We gain further flexibility by using the letter E, which stands for: *times ten to the power*. Thus, we may write .0012345678 as .12345678E-2 or 12345678E-10 or 1234.5678E-6. We do not write E7 as a number, but write 1E7 to indicate that it is 1 that is multiplied by 10^7.

1.3.4 Variables

A simple (i.e., unsubscripted) numeric variable in BASIC is denoted by any letter or by any letter followed by a single digit. (Refer to Chapter 3 for a discussion of subscripted numeric variables and to Chapter 8 for a discussion of subscripted and unsubscripted string variables.) Thus, the computer interprets E7 as a variable, along with A, X, N5, I0, and O1. A numeric variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program is written. Variables are given or assigned values by LET and READ statements. The value so assigned does not change until the next time a LET or READ statement is encountered with a value for that variable. However, all numeric variables are set equal to 0 before a RUN. Consequently, it is only necessary to assign a value to a numeric variable when a value other than 0 is required.

Although the computer does little in the way of correcting during computation, it sometimes helps if an absolute value hasn't been indicated. For example, if you ask for the square root of -7 or the logarithm of -5, the computer gives the square root of 7 along with an error message stating that you have asked for the square root of a negative number, or it gives the logarithm of 5 along with the error message that you have asked for the logarithm of a negative number.

1.3.5 Relational Symbols

Six other mathematical symbols of relation are used in IF-THEN statements where it is necessary to compare values. An example of the use of these relation symbols was given in the sample program LINEAR.

Any of the following six standard relations may be used:

Symbol	Example	Meaning
=	A = B	A is equal to B
<	A <B	A is less than B
<=	A <=B	A is less than or equal to B
>	A >B	A is greater than B
>=	A >=B	A is greater than or equal to B
<>	A <>B	A is not equal to B

Note that while BASIC outputs its answers with only six places of accuracy, variables and formulas may have values accurate to more than six places. If it is desired that result X be checked to only N places, the function

```
INT       (X*1Ø↑N+.5)/1Ø↑N
```

should be used.

1.4 SUMMARY

Several elementary BASIC statements have been introduced in our discussions. In describing each of these statements, a line number is assumed, and brackets are used to denote a general type. For example, [variable] refers to any variable.

1.4.1 LET Statement

The LET statement is used when computations must be performed. This command is not of algebraic equality, but a command to the computer to perform the indicated computations and assign the answer to a certain variable. Each LET statement is of the form:

```
    LET [variable] = [formula]
or
    [variable] = [formula]
```

Generally, several variables may be assigned the same value by a single LET statement. Examples of assigning a value to a single variable are given in the following two statements:

```
1ØØ     LFT  X=X+1
259     W7=(W-X4↑3)*(Z-A/(A-B)-17)
```

Examples of assigning a value to more than one variable are given in the following statements:

5Ø	`X=Y3=A(3,1)=1`	The variables X, Y3, and A(3,1) are assigned the value 1.
9Ø	`LET W=Z=3*X-4*X↑2`	The variables W and Z are assigned the value $3X-4X^2$

1.4.2 READ and DATA Statements

READ and DATA statements are used to enter information into the computer. We use a READ statement to assign to the listed variables those values which are obtained from a DATA statement. Neither statement is used without the other. A READ statement causes the variables listed in it to be given in order, the next available numbers in the collection of DATA statements. Before the program is run, the computer takes all of the DATA statements in the order they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, the program is assumed to be finished and we get an OUT OF DATA message.

Since we have to read in data before we can work with it, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they occur in the correct order. A common practice is to collect all DATA statements and place them just before the END statement.

Each READ statement is of the form:

READ [sequence of variables]

Each DATA statement is of the form:

DATA [sequence of numbers]

```
150     READ X,Y,Z,X1,Y2,G9
330     DATA 4,2,1.7
340     DATA 6.734E-3,-174.321,3.1415927

234     READ B(K)
263     DATA 2,3,5,7,9,11,10,8,6,4

10      READ R(I,J)
440     DATA -3,5,-9,2.37,2.9876,-437.234E-5
450     DATA 2.765, 5.5576, 2.3789E2
```

Remember that numbers, not formulas, are put in a DATA statement, and that 15/7 and SQR(3) are formulas. Refer to Chapter 3 for a discussion of the subscripted variables.

1.4.3 PRINT Statement

The common uses of the PRINT statement are:

a. to print out the results of some computations

b. to print out verbatim a message included in the program

c. a combination of the two

d. to skip a line.

The following are examples of a type a.:

```
100     PRINT X,SQR(X)
135     PRINT X,Y,Z, B*B-4*A*C, EXP(A-B)
```

The first example prints X, and a few spaces to the right, the square root of X. The second prints five different numbers:

$$X, Y, Z, B^2, -4AC, \text{and } e^{A-B}$$

The computer computes the two formulas and prints up to five numbers per line in this format.

The following are examples of type b.:

```
100     PRINT "NO UNIQUE SOLUTION"
430     PRINT "X VALUE", "SINE", "RESOLUTION"
500     PRINT X,M,D
```

Line 100 prints the sample statement, and line 430 prints the three labels with spaces between them. The labels in 430 automatically line up with the three numbers called for in PRINT statement 500.

The following is an example of type c.:

```
150     PRINT "THE VALUE OF X IS" X
30      PRINT "THE SQUARE ROOT OF" X "IS" SQR(X)
```

If the first has computed the value of X to be 3, it prints out: THE VALUE OF X IS 3. If the second has computed the value of X to be 625, it prints out: THE SQUARE ROOT OF 625 IS 25.

The following is an example of type d.:

```
250     PRINT
```

The computer advances the paper one line when it encounters this command.

1.4.4 GO TO Statement

The GO TO statement is used when we want the computer to unconditionally transfer to some statement other than the next sequential statement. In the LINEAR problem, we direct the computer to go through the same process for different values of C and F with a GO TO statement. This statement is in the form of GO TO [line number].

```
150     GO TO 75
```

1.4.5 IF - THEN Statement

The IF - THEN statement is used to transfer conditionally from the sequential order of statements according to the truth of some relation. It is sometimes called a conditional GO TO statement. Each IF - THEN statement is of the form:

IF [formula] [relation] [formula], THEN [line number]

The comma preceding THEN is optional and can be omitted.

The following are two examples of this statement:

```
40      IF SIN(X)<=M THEN 80
20      IF G=0, THEN 65
```

The first asks if the sine of X is less than or equal to M, and skips to line 80 if so. The second asks if G is equal to 0, and skips to line 65 if so. In each case, if the answer to the question is no, the computer goes to the next line.

1.4.6 ON - GO TO Statement

The IF - THEN statement allows a two-way fork in a program; the ON - GO TO statement allows a many-way switch. The ON - GO TO statement has the form:

ON [formula], GO TO [line number], [line number], ... [line number]

The comma preceding the GO TO can be omitted. For example:

```
80      ON X GO TO 100, 200, 150
```

This condition causes the following to occur:

If X = 1, the program goes to line 100,
If X = 2, the program goes to line 200,
If X = 3, the program goes to line 150

In other words, any formula may occur in place of X, and the instruction may contain any number of line numbers, as long as it fits on a single line. The value of the formula is computed and its integer part is taken. If this is 1, the program transfers to the line whose number is first on the list; if its integer part is 2, the program transfers to the line whose number is the second one, etc. If the integer part of the formula is below 1, or larger than the number of line numbers listed, an error message is printed. To increase the similarity between the ON - GO TO and IF - THEN instructions, the instruction

```
75      IF X>5 THEN 200
```

may also be written as:

```
75      IF X>5 GO TO 200
```

Conversely, THEN may be used in an ON - GO TO statement.

1.4.7 END Statement

Every program must have an END statement, and it must be the statement with the highest line number in the program.

```
999      END
```

CHAPTER 2
LOOPS

We are frequently interested in writing a program in which one or more portions are executed a number of times, usually with slight variations each time. To write a program in such a way that the portions of the program to be repeated are written just once, we use loops. A loop is a block of instructions that the computer executes repeatedly until a specified terminal condition is met.

The use of loops is illustrated and explained by using two versions of a program that performs the simple task of printing out the positive integers 1 through 100 together with the square root of each. The first version, which does not use a loop, is 101 statements long and reads

```
10      PRINT 1,SQR(1)
20      PRINT 2,SQR(2)
30      PRINT 3,SQR(3)
        • • • • • • •
990     PRINT 99,SQR(99)
1000    PRINT 100,SQR(100)
1010    END
```

The second version, which uses one type of loop, obtains the same results with far fewer instructions (5 instead of 101):

```
10      LET X=1
20      PRINT X,SQR(X)
30      LET X=X+1
40      IF X<=100 THEN 20
50      END
```

Statement 10 gives the value of 1 to X and initializes the loop. In line 20, both 1 and its square root are printed. Then, in line 30, X is increased by 1, to a value of 2. Line 40 asks whether X is less than or equal to 100; an affirmative answer directs the computer back to line 20, where it prints 2 and $\sqrt{2}$ and goes to 30. Again, X is increased by 1, this time to 3, and at 40 it goes back to 20. This process is repeated -- line 20 (print 3 and $\sqrt{3}$), line 30 (X = 4), line 40 (since 4 < 100, go back to line 20), etc. -- until the loop has been traversed 100 times. Then, after it has printed 100 and its square root, X becomes 101. The computer now receives a negative answer to the question in line 40 (X is greater than 100, not less than or equal to it), does not return to 20 but moves on to line 50, and ends the program. All loops contain four characteristics:

a. initialization (line 10)

b. the body (line 20)

- modification (line 30)

d. an exit test (line 40)

2.1 FOR AND NEXT STATEMENTS

BASIC provides two statements to specify a loop: the FOR statement and the NEXT statement.

```
10      FOR X=1 TO 100
20      PRINT X,SQR(X)
30      NEXT X
50      END
```

In line 10, X is set equal to 1, and a test is executed, like that of line 40 above. Line 30 carries out two tasks: X is increased by 1, and control transfers back to the test in line 10. There the test is carried out to determine whether to execute the body of the loop again or to go on to the statement following line 30. Thus, lines 10 and 30 take the place of lines 10, 30, and 40 in the previous program.

Note that the value of X is increased by 1 each time BASIC goes through the loop. If we want a different increase, e.g., 5, we could specify it by writing the following:

```
10      FOR X=1 TO 100 STEP 5
```

and then the value of X on the first time through the loop would be 1, on the second time 6, on the third 11, and on the last time 96. The step of 5 which would take X beyond 100 to 101 causes control to transfer to line 50, which ends the program. The STEP may be positive, negative, or zero. We could have caused the original results to be printed in reverse order by writing line 10 as follows:

```
10      FOR X=100 TO 1 STEP-1
```

In the absence of a STEP instruction, a step-size of +1 is assumed.

The word BY may be substituted for the word STEP; FOR TO BY and FOR TO STEP statements are completely equivalent.

More complicated FOR statements are allowed. The initial value, the final value, and the step-size may all be formulas of any complexity. For example, we could write the following:

```
FOR X=N+7*Z TO (Z-N)/3 BY(N-4*Z)/10
```

For a positive or zero step-size, the loop continues as long as the control variable is less than or equal to the final value. For a negative step-size, the loop continues as long as the control variable is greater than or equal to the final value.

If the initial value is greater than the final value (less than the final value for negative step-size), the body of the loop is not performed at all, but the computer immediately passes to the statement following the NEXT. The following program for adding up the first n integers gives the correct result 0 when n is 0.

```
10      READ N
20      LET S=0
30      FOR K=1 TO N
40      LET S=S+K
50      NEXT K
60      PRINT S
70      GO TO 10
90      DATA 3,10,0
99      END
```

In the following description of the instructions used to specify a loop, a line number is assumed and brackets are used to denote a general type.

A FOR statement has one of two forms:

$$\text{FOR}\begin{bmatrix}\text{numeric}\\ \text{variable}\end{bmatrix} = \text{[formula] TO [formula] STEP [formula]}$$

or

$$\text{FOR}\begin{bmatrix}\text{numeric}\\ \text{variable}\end{bmatrix} = \text{[formula] TO [formula] BY [formula]}$$

Most commonly, the expressions are integers and the STEP or BY is omitted. In the latter case a step-size of +1 is assumed. The accompanying NEXT statement has one of two forms.

NEXT [variable]

NEXT [variable, variable,. . .variable]

The first form contains one variable that must be the same as that following FOR in the FOR statement. The second form of the NEXT statement contains two or more variables separated by commas. These variables must also match the variables in their accompanying FOR statements.

When the second form of NEXT is used, the variables must be written in the same order as they would be written in separate NEXT statements. That is, the variable that matches the last FOR statement is first, that which matches the next-to-last FOR is second, and the variable that matches the first FOR statement is last. This causes the loops to be nested properly (refer to section 2.2). For example:

```
FOR X                      FOR X
FOR Y                      FOR Y
FOR Z   is equivalent to   FOR Z
NEXT Z                     NEXT Z, Y, X
NEXT Y
NEXT X
```

Note that for each FOR statement there is one and only one variable in a NEXT statement, and vice versa. Some examples of FOR and NEXT statements are:

```
30      FOR X=Ø TO 3 STEP D
8Ø      NEXT X

12Ø     FOR X4=(17+COS(Z))/3 TO 3* SQR(1Ø) BY 1/4
235     NEXT X4

24Ø     FOR X=8 TO 3 STEP -1
456     FOR J=-3 TO 12 BY 2
5ØØ     NEXT J,X
```

Line 120 specifies that the successive values of X4 are .25 apart, in increasing order. Line 240 specifies that the successive values of X will be 8, 7, 6, 5, 4, 3. Line 456 specifies that J will take on values -3, -1, 1, 3, 5, 7, 9, and 11. If the initial, final, or step-size index values are given as formulas, these formulas are evaluated only upon entering the FOR statement; therefore, if after this evaluation we change the value of a variable in one of these formulas, we do not affect the index value.

The control variable can be changed in the body of the loop; it should be noted that the exit test always uses the latest value of this variable.

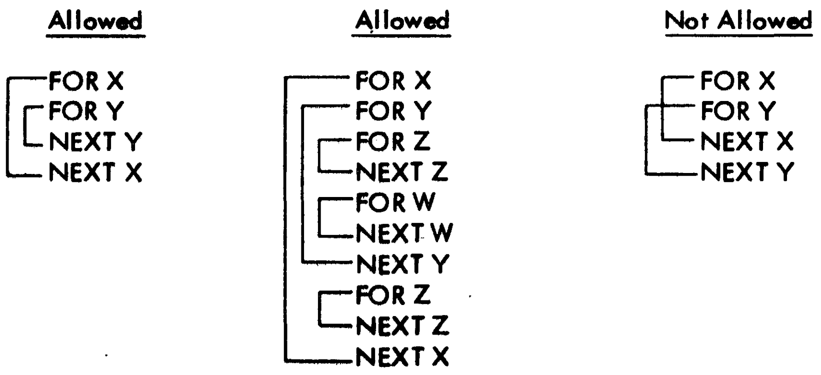
The following difficulty can occur with loops, both FOR-NEXT loops and loops explicitly written with LET and IF statements (as in the example on page 2-1). The calculation of the index values (initial, final, and step-size) is subject to precision limitations inherent in the computer. These values are represented in the computer as binary numbers. When the values are integer, they can be represented exactly in binary; however, it is not always possible to represent decimal values exactly in binary when they contain a fractional part. For example, a loop of the form:

```
4Ø FOR X=Ø TO 1Ø STEP Ø.1
95 NEXT X
```

executes 100 times instead of 101 times because the internal value for 0.1 is not exactly 0.1 After the hundredth execution of the loop, X is not exactly equal to 10, it is slightly larger than 10, so the loop stops. Whenever possible, it is advisable to use indices that have integer values because then the loop will <u>always</u> execute the correct number of times.

2.2 NESTED LOOPS

Nested loops (loops within loops) can be expressed with FOR and NEXT statements. They must be nested and not crossed as the following skeleton examples illustrate:



Allowed

```
FOR X
FOR Y
NEXT Y
NEXT X
```

Allowed

```
FOR X
FOR Y
FOR Z
NEXT Z
FOR W
NEXT W
NEXT Y
FOR Z
NEXT Z
NEXT X
```

Not Allowed

```
FOR X
FOR Y
NEXT X
NEXT Y
```

CHAPTER 3
LISTS AND TABLES

In addition to the ordinary variables used by BASIC, variables can be used to designate the elements of a list or a table. Many occasions arise where a list or a table of numbers is used over and over, and, since it is inconvenient to use a separate variable for each number, BASIC allows the programmer to designate the name of a list or table by a single letter or a single letter followed by a single digit.

Lists are used when we might ordinarily use a single subscript, as in writing the coefficients of a polynomial ($a_0, a_1, a_2, \ldots, a_n$). Tables are used when a double subscript is to be used, as in writing the elements of a matrix ($b_{i,j}$). The variables used in BASIC consist of a single letter or a letter and a digit which is the name of the list or table, followed by the subscript in parentheses. Thus,

A(0), A(1), A(2), ..., A(N)

represents the coefficients of a polynomial, and

B7(1,1), B7(1, 2), ..., B7(N,N)

represents the elements of a matrix. (Refer to Chapter 8 for a discussion of string variables.)

The single letter or the letter and digit denoting a list or a table name may also be used without confusion to denote a simple variable. However, the same name may not be used to denote both a list and a table in the same program because BASIC recognizes a list as a special kind of table having only one column. The form of the subscript is flexible: A list item B(I + K) may be used, or a table item Q(A(3,7), B-C) may be used. The value of the subscript must not be less than zero.

We can enter the list A(0), A(1), ..., A(10) into a program by the following lines:

```
10      FOR I=0 TO 10
20      READ A(I)
30      NEXT I
40      DATA 0,2,3,-5,2.2,4,-9,123,4,-4,3
```

3.1 THE DIMENSION STATEMENT (DIM)

BASIC automatically reserves room for any list or table with subscripts of 10 or fewer. However, if we want larger subscripts, we must use a DIM statement. This statement indicates to the computer that the specified space is to be allowed for the list or table. DIM can also be written as DIMENSION. For example, the instruction

```
10      DIM A(15)
```

reserves 16 spaces for list A (A(0), A(1), A(2), ..., A(15)). The instruction

```
20      DIMENSION Y5(10,15)
```

reserves 176 spaces for matrix Y5 (10 + 1 rows * 15 + 1 columns). Space may be reserved for more than one list and/or table with a single DIM statement by separating the entries with commas, as shown in the following example:

```
30      DIM A(100),B(20,30),C4(25)
```

A DIM (or DIMENSION) statement is not executed; therefore, it may appear on any line before the END statement. However, the best place to put it is at the beginning so that it is not forgotten. If we enter a table with a subscript greater than 10, without a DIM statement, BASIC gives an error message, telling us that we have a subscript error. This condition can be rectified by entering a DIM statement with a line number less than the line number of the END statement.

A DIM (or DIMENSION) statement is normally used to reserve additional space, but in a long program that requires many small tables, it may be used to reserve less space for tables in order to have more space for the program. When in doubt, declare a larger dimension than you expect to use, but not one so large that there is no room for the program. For example, if we want a list of 15 numbers entered, we may write the following:

```
10      DIM A(25)
20      READ N
30      FOR I=1 TO N
40      READ A(I)
50      NEXT I
60      DATA 15
70      DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

Statements 20 and 60 could have been eliminated by writing 30 as FOR I = 1 TO 15 but the program as typed allows for the lengthening of the list by changing only statement 60, as long as the list does not exceed 25 and there is sufficient data.

We could enter a 3-by-5 table into a program by writing the following:

```
10      FOR I=1 TO 3
20      FOR J=1 TO 5
30      READ B(I,J)
40      NEXT J
50      NEXT I
60      DATA 2,3,-5,-9,2
70      DATA 4,-7,3,4,-2
80      DATA 3,-3,5,7,8
```

Again, we may enter a table with no DIM (or DIMENSION) statement: BASIC then handles all the entries from B(0,0) to B(10,10).

3.2 EXAMPLE

Below are the statements and run of a problem which uses both a list and a table. The program computes the total sales of five salesmen, all of whom sell the same three products. The list, P, gives the price per item of the three products and the table, S, tells how many items of each product each man sold. Product 1 sells for $1.25 per item, product 2, for $4.30 per item, and product 3, for $2.50 per item; also, salesman 1 sold 40 items of the first product, 10 of the second, 35 of the third, and so on. The program reads in the price list in lines 40 through 80, using data in lines 910 through 930. The same program could be used again, modifying only line 900 if the prices change, and only lines 910 through 930 to enter the sales in another month. This sample program does not need a DIM statement, because the computer automatically reserves enough space to allow all subscripts to run from 0 to 10.

NOTE

Since spaces are ignored, statements may be indented for visual identity of the various loops within the program.

```
10      FOR I=1 TO 3
20        READ P(I)
30      NEXT I
40      FOR I=1 TO 3
50        FOR J=1 TO 5
60      READ S(I,J)
70          NEXT J
80       NEXT I
90      FOR J=1 TO 5
100       LET S=0
110       FOR I=1 TO 3
120        LET S=S+P(I)*S(I,J)
130       NEXT I
140       PRINT "TOTAL SALES FOR SALESMAN"J,"$"S
150      NEXT J
900      DATA 1.25,4.30,2.50
910      DATA 40,20,37,29,42
920      DATA 10,16,3,21,8
930      DATA 35,47,29,16,33
999      END
```

(continued on next page)

```
RUN
SALES1          11:06          20-OCT-69
TOTAL SALES FOR SALESMAN 1 $ 180.500
TOTAL SALES FOR SALESMAN 2 $ 211.300
TOTAL SALES FOR SALESMAN 3 $ 131.650
TOTAL SALES FOR SALESMAN 4 $ 166.500
TOTAL SALES FOR SALESMAN 5 $ 169.400
TIME: 0.16 SECS.
```

3.3 SUMMARY

Because the number of simple variable names is limited, BASIC allows a programmer to use lists and tables to increase the number of problems that can be programmed easily and concisely. A single letter or a single letter followed by a single digit is used for the name of the list or table, and the subscript that follows is enclosed in parentheses. A subscript may be a number or any legal expression.

Lists and tables are called subscripted variables, and simple variables are called unsubscripted variables. Usually, you can use a subscripted variable anywhere that you use an unsubscripted variable. However, the variable mentioned immediately after FOR in the FOR statement and after NEXT in the NEXT statement must be an unsubscripted variable. The initial, terminal, and step values may be any legal expression.

3.3.1 The DIM (or DIMENSION) Statement

To enter a list or a table with a subscript greater than 10, a DIM statement, which has DIMENSION as an alternate form, is used to retain sufficent space, as in the following examples:

```
20      DIMENSION H(35)
35      DIM Q8(5,25)
```

The first example enables us to enter list H with 36 items (H(0), H(1), ..., H(35)). The second reserves space for a table of 156 items (5 + 1 rows * 25 + 1 columns).

CHAPTER 4
HOW TO RUN BASIC

After learning how to write a BASIC program, we must learn how to gain access to BASIC via the Teletype so that we can type in a program and have the computer solve it. Steps required to communicate with the monitor must first be performed. These steps are fully explained in DECsystem-10 Users Handbook and the DECsystem-10 Operating System Commands manual.

4.1 GAINING ACCESS TO BASIC

After arriving at a terminal, follow three steps in order to enter BASIC

1. Contact the DECsystem-10 computer,
2. Complete the LOGIN procedure, and
3. Access BASIC.

4.1.1 Contacting the DECsystem-10 Computer

Turn the terminal knob to LINE. Next, the terminal must be connected to the computer, either directly, by means of a cable that leads from the terminal to the computer, or indirectly, using the telephone system to link the terminal to the computer.

For direct connections, the user is not required to do anything more to contact the system than to turn the knob to LINE.

Since the procedure for obtaining a telephone system connection differs from one installation to another, the user should obtain the instructions from the operations staff at his particular installation.

4.1.2 Completing the LOGIN Procedure

Three steps comprise the LOGIN procedure:

1. Depress ↑C and type LOGIN. This signals the computer that you want to use it. After you type LOGIN, the monitor types your job number, the version of the monitor, and your terminal number. It then types a #.

2. Type your project-programmer number. This number is assigned to you by the computer administration staff. Next, the monitor asks for your password by typing PASSWORD:.

3. Type your password (also assigned by the computer administrations staff). It is not printed.

If your password and project-programmer number are both valid, the monitor types the time, date and day of the week. Refer to Paragraph 4.7 for an example of the LOGIN procedure.

4.1.3 Accessing BASIC

When the DECsystem-10 is ready to accept commands, the monitor responds with a period. Type R BASIC to clear the user's core memory area and establish contact with the BASIC program. When BASIC is ready to accept commands it types READY, FOR HELP TYPE HELP.

NOTE

> In some cases the system automatically executes the R BASIC command for the user. If READY, FOR HELP TYPE HELP appears immediately after the LOGIN procedure, this option has been enacted.

You can either type HELP to get a list of commands that you can give to BASIC or type any command or statement that you wish.

If you are going to create a new program, type in:

```
NEW
```

BASIC responds with the following:

```
NEW FILE NAME--
```

Type in the name of your new program. If you want to work with a previously created program that you saved on a storage device, type in the following:

```
OLD
```

BASIC then asks for the name of the old program, as follows:

```
OLD FILE NAME--
```

Respond by typing in the name of your old file. If your old file is stored on a device other than the disk, you must type in the device name as in the following example:

```
            OLD FILE NAME--DTA6:SAMPLE
```

BASIC retrieves the file named SAMPLE from DECtape 6 and replaces the current contents of user core with the file SAMPLE. The disk may be specified as the device on which the old program is stored, but this is not necessary because the disk is the device used when no device is specified. For example, the following statements are equivalent:

```
            OLD FILE NAME--DSK:TEST1
            OLD FILE NAME--TEST 1
```

Device names are as follows:

DSK	the disk
DTA0 through DTA7	DECtapes number 0 through 7 on the first control unit
DTB0 through DTB7	DECtapes number 0 through 7 on the second control unit
TTY	your Teletype
TTY0 through TTY177	Teletypes number 0 through 177
LPT	the line printer
MTA0 through MTA7	magnetic tapes number 0 through 7
PTP	the high-speed paper-tape punch
PTR	the high-speed paper-tape reader
CDP	the card punch
CDR	the card reader
SYS	the system device where system programs are stored
BAS or ***1	the library where your installation stores BASIC programs for all BASIC users

Not all installations have all of these devices; if you specify a device that does not exist or that is not available for your use, BASIC returns an error message. Also, while it is possible to store a file on the card punch, for example, the file cannot be retrieved from this device but must be retrieved from the card reader. If you specify for OLD a device that can only do output, an error message will be returned..

Program names can be any combination of letters and digits up to and including six characters in length. In addition to specifying a program name, you may also specify an extension. The extension follows the name and is separated from it by a period. An extension is any combination of letters and digits up to and including three characters in length. In previous chapters we have used program names such as LINEAR and SALES1. If you recall an old program from storage, you must use exactly the same name and extension you assigned to it when it was saved.

1 When the asterisk is used, it follows the filename and extension rather than preceding them as with the other devices.

You can also type the name of your file (and the device on which it is located) on the same line as the NEW or OLD command. In this case, BASIC will not ask for the name of the file. For example:

```
NEW TEST

OLD DTA6:SAMPLE
```

The NEW OR OLD -- request can be answered not only by NEW or OLD, but also by any other command (refer to Chapter 9 for a description of the commands) or statement. If NEW OR OLD -- is answered by a NEW, OLD, or RENAME command, the current device, filename, and extension are established by the arguments specified with the command; if a device is not specified explicitly, the disk is assumed; if a filename is specified without an extension, the extension BAS is assumed; it is illegal to specify an extension without specifying a filename.

If NEW OR OLD -- is answered by anything other than a NEW, OLD, or RENAME command, the current device, filename, and extension are established as DSK, NONAME, and BAS, respectively. For example, the following sequence creates a disk file called NONAME.BAS.

```
.R BASIC
NEW OR OLD -- 5 PRINT "TESTING"
10 END
SAVE
```

A new current device, filename, and extension are established whenever a NEW, OLD, or RENAME command is given.

To indicate that you wish to use the library BAS, you can type either of the following:

```
filename.ext***

BAS:filename.ext
```

When BASIC reads either of these forms, it looks for "device" BAS. If BASIC cannot find BAS, it assumes that you mean the disk area with the project-programmer number [5,1] where BAS normally resides.

4.2 ENTERING THE PROGRAM

After you type in your filename (whether it is old or new), BASIC responds with the following:

```
READY
```

You can begin to type in your program. Make sure that each line begins with a line number containing no more than five digits and containing no spaces or nondigit characters. Also, be sure to start at the beginning of the Teletype line for each new line. Press the RETURN key upon completion of each line.

If, in the process of typing a statement, you make a typing error and notice it before you terminate the line, you can correct it by pressing the RUBOUT key once for each character to be erased, going backward until the character in error is reached. Then continue typing, beginning with the character in error. The following is an example of this correcting process:

```
10      PRNIT\TIN\INT 2,3
```

NOTE

The RUBOUT key echoes as a backslash (\), followed by the deleted characters and a second backslash.

4.3 EXECUTING THE PROGRAM

After typing the complete program (do not forget to end with an END statement), type RUN or RUNNH, followed by the RETURN key. BASIC types the name of the program, the time of day, the current date (unless RUNNH is specified), and then it analyzes the program. If the program can be run, BASIC executes it and, via PRINT statements, types out any results that were requested. The typeout of results does not guarantee that the program is correct (the results could be wrong), but it does indicate that no grammatical errors exist (e.g., missing line numbers, misspelled words, or illegal syntax). If errors of this type do exist, BASIC types a message (or several messages) to you. A list of these diagnostic messages, with their meanings, is given in Appendix B.

4.4 CORRECTING THE PROGRAM

If you receive an error message typeout informing you, for example, that line 60 is in error, the line can be corrected by typing in a new line 60 to replace the erroneous one. If the statement on line 110 is to be eliminated from your program, it is accomplished by typing the following:

```
110
```

If you wish to insert a statement between lines 60 and 70, type a line number between 60 and 70 (e.g., 65), followed by the statement.

4.5 INTERRUPTING THE EXECUTION OF THE PROGRAM

If the results being typed out seem to be incorrect and you want to stop the execution of your program or suppress its typeout, type ↑O (hold down CTRL key and at the same time type O) to suppress the typeout, or type ↑C twice to stop execution, as indicated in the following example:

```
↑C
↑C
```

Stops execution of your program, closes any files that are open in the program (refer to Chapter 10), and

Returns to BASIC command level.

If you typed ↑C, BASIC responds with the following:

```
READY
```

whereupon you can modify or add statements and/or type RUN or any other command.

4.5.1 Returning to Monitor Level

If you wish to leave BASIC and return to monitor level, type:

```
MONITOR
```

the monitor responds with a period and waits for you to type a monitor command. If you wish to return to BASIC, you must not type a command that will change what you have in core (i.e., the ASSIGN command does not change what is in core, but the DIRECT command does change core). To return to BASIC, type the following: .START or .REENTER or .CONT

```
.START  or  .REENTER  or  .CONT
```

BASIC responds with

```
READY
```

and you can continue working in BASIC.

4.6 LEAVING THE COMPUTER

When you wish to leave the computer, type the BYE or GOODBYE command.

No files created on the disk by BASIC commands or program statements are deleted by this procedure.

The system monitor responds to the BYE or GOODBYE command by logging you off the system completely, unless your files stored on the disk take up so much room that you are over the logged-out quota set by the system administrator. In that case, the following message is typed out (n and m are the appropriate integers):

```
DSK LOGGED OUT QUOTA n EXCEEDED BY m BLOCKS
CONFIRM:
```

If you then type

```
H↵
```

instructions for deleting files at logout time are typed on your Teletype.

4.7 EXAMPLE OF BASIC RUN

The following is a simple example of the use of BASIC under a timesharing monitor:

```
.↑C                           GO TO MONITOR LEVEL
.LOGIN                        REQUEST LOGIN
JOB 7 5S0318A TTY34           MONITOR TYPES OUT YOUR ASSIGNED
                              JOB NUMBER, THE CURRENT VERSION
                              NUMBER OF THE MONITOR, AND YOUR
                              TELETYPE NUMBER

.                             MONITOR REQUESTS YOUR PROJECT-
#27,20                        PROGRAMMER NUMBER; TYPE IT IN

PASSWORD:                     MONITOR REQUESTS YOUR PASSWORD;
                              TYPE IT IN; IT WILL NOT ECHO BACK

0927 29-OCT-69 WED            MONITOR TYPES OUT THE TIME OF
                              DAY, THE CURRENT DATE, THE DAY OF
                              THE WEEK, AND A PERIOD

.R BASIC                      INSTRUCT MONITOR TO BRING BASIC
                              INTO CORE AND START ITS EXECUTION

READY, FOR HELP TYPE HELP     BASIC INDICATES THAT IT IS READY
                              TO RECEIVE A COMMAND OR STATE-
                              MENT

NEW                           TYPE THE COMMAND NEW


NEW FILE NAME--SAMPLE         BASIC ASKS FOR NEW FILENAME

READY                         BASIC IS NOW READY TO RECEIVE
                              STATEMENTS

10      FOR N=1 TO 7          TYPE IN STATEMENTS

20      PRINT N, SQR(N)

30      NEXT N

40      PRINT "DONE"

50      END

RUN                           RUN PROGRAM

SAMPLE        11:14             29-OCT-69

 1             1

 2             1.41421

 3             1.73205

 4             2

 5             2.23607

 6             2.44949
```

(continued on next page)

```
 7                2.64575

DONE

TIME: 0.20 SECS.

READY

BYE

JOB 7, USER [27, 20] LOGGED OFF TTY34 0930 29-OCT-69

SAVED ALL 1 FILE (5 DISK BLOCKS)

RUNTIME 0 MIN, 01 SEC

.
```

4.8 ERRORS AND DEBUGGING

Occasionally, the first run of a new problem is free of errors and gives the correct answers, but, more commonly, errors are present and have to be corrected. Errors are of two types: errors of form (grammatical errors) which prevent the running of the program, and logical errors in the program which cause the computer to produce wrong answers or no answers at all.

Errors of form cause error messages to be printed, and the various types of error messages are listed and explained in Appendix B. Logical errors are more difficult to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines, or by deleting lines from the program. As indicated previously, a line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those of two existing lines; and a line is deleted by typing its line number and pressing the RETURN key. Note that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers begin by using arbitrary line numbers that are multiples of five or ten.

These corrections can be made either before or after a run. Since BASIC sorts out lines and arranges them in order, a line may be retyped out of sequence. Simply retype the offending line with its original line number.

4.8.1 Example of Finding and Correcting Errors

We can best illustrate the process of finding the errors (bugs) in a program and correcting (debugging) them by an example. Consider the problem of finding that value of X between 0 and 3 for which the sine of X is a maximum, and ask the machine to print out this value of X and the value of its sine.

Although we know that π/2 is the correct value, we use the computer to test successive values of X from 0 to 3, first using intervals of .1, then of .01, and finally of .001. Thus, we ask the computer to find the sine of 0, of .1, of .2, of .3..., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. It does so by testing SIN(0) and SIN(.1) to see which is larger, and calling the larger of these two numbers M. It then picks the larger of M and SIN (.2) and calls it M. This number is checked against SIN (.3). Each time a larger value of M is found, the value of X is "remembered" in X0. When it finishes, M will have been assigned to the largest value. It then repeats the search, this time checking the 301 numbers 0, .01, .02, .03, ..., 2.98, 2.99, and 3, finding the sine of each, and checking to see which has the largest sine. At the end of each of these three searches, we want the computer to print three numbers: the value X0 which has the largest sine, the sine of that number, and the interval of search.

Before going to the Teletype, we write a program such as the following:

```
10      READ D
20      LET X0=0
30      FOR X=0 TO 3 STEP D
40      IF SIN(X)<=M THEN 100
50      LET X0=X
60      LET M=SIN(X0)
70      PRINT X0,X,D
80      NEXT X0
90      GO TO 20
100     DATA .1,.01,.001
110     END
```

The following is a list of the entire sequence on the Teletype with explanatory comments on the right side:

```
READY, FOR HELP TYPE HELP
NEW
NEW FILE NAME--MAXSIN
READY
10      READ D
20      LWR X0=0
30      FOR X=0 TO 3 STEP D
40      IF SINE\E\(X)<=M THEN 100
50      LET X0=X
60      LET M=SIN(X)
70      PRINT X0,X,D
80      NEXT T\T\X0
90      GO TO 20
20      LET X0=0
100     DATA .1,.01,.001
110     END
RUN
```

Note the use of the RUBOUT key (echoes as a \) to erase a character in line 40 (which should have started IF SIN (X), etc.) and in line 80.

We discover that LET was mistyped in line 20, and we correct it after 90.

```
MAXSIN          11:35           20-OCT-69

ILLEGAL VARIABLE IN 70
NEXT WITHOUT FOR IN 80
FOR WITHOUT NEXT IN 30
TIME: 0.05 SECS.
READY

70      PRINT X0,X,D
40      IF SIN (X) <=M THEN 80
80      NEXT X
RUN

MAXSIN          11:36           20-OCT-69

 0.1             0.1             0.1
 0.2             0.2             0.1
 0.3             ↑C↑C
READY


20
RUN

MAXSIN          11:37           20-OCT-69
UNDEFINED LINE NUMBER 20 IN 90
TIME: 0.03 SECS.

90      GO TO 10
RUN

MAXSIN          11:43           20-OCT-69

 0.1             0.1             0.1
 0.2             0.2             0.1
 0.3             ↑C↑C
READY
```

After receiving the first error message, we inspect line 70 and find that we used XO for a variable instead of X0. The next two error messages relate to lines 30 and 80 having mixed variables. These are corrected by changing line 80.

Both of these changes are made by retyping lines 70 and 80. In looking over the program, we also discover that the IF - THEN statement in 40 directed the computer to a DATA statement and not to line 80 where it should go. This is obviously incorrect. We are having every value of X printed, so we direct the machine to cease operations by typing ↑C twice even while it is running. We notice that SIN(0) is compared with M on the first time through the loop, but we had assigned a value to X0 but not to M. However, we recall that all variables are set equal to zero before a RUN; therefore, line 20 is unnecessary.

Line 90, of course, sent us back to line 20 to repeat the operation and not back to line 10 to pick up a new value for D. We retype line 90 and then type RUN again.

We are about to print out the same table as before. Each time that it goes through the loop, it is printing out X0, the current value of X, and the interval size.

```
70
85     PRINT X0,M,D
5      PRINT "X VALUE","SIN",RESOLUTION"
RUN

MAXSIN          11:44           20-OCT-69

ILLEGAL VARIABLE IN 5
TIME: 0.08 SECS.
READY
5      PRINT "X VALUE","SIN","RESOLUTION"
RUN
```

We rectify this condition by moving the PRINT statement outside the loop. Typing 70 deletes that line, and line 85 is outside of the loop. We also realize that we want M printed, not X. We also decide to put in headings for the columns by a PRINT statement.

There is an error in our PRINT statement: no left quotation mark for the third item.

(continued on next page)

```
MAXSIN          11:47           20-OCT-69
X VALUE         SINE            RESOLUTION
 1.60            0.999574        0.1
 1.57            1.              0.01
 1.57099         1.              0.001
OUT OF DATA IN 10
TIME: 0.96 SECS.
READY
```

Retype line 5, with all of the required quotation marks.

These are the desired results. Of the 31 numbers (0, .1, .2, .3,..., 2.8, 2.9, 3), it is 1.6 which has the largest sine, namely .999574; this is true for finer subdivisions.

```
LIST

MAXSIN 11:48 20-OCT-69

5      PRINT "X VALUE","SINE","RESOLUTION"
10     READ D
30     FOR X=0 TO 3 STEP D
40     IF SIN(X)<=M THEN 80
50     LET X0=X
60     LET M=SIN(X)
80     NEXT X
85     PRINT X0,M,D
90     GO TO 10
100    DATA .1, .01,.001
110    END
```

Having changed so many parts of the program, we ask for a list of the corrected program.

```
READY
SAVE
READY
```

The program is saved for later use.

A PRINT statement could have been inserted to check on the machine computations. For example, if M were checked, we could have inserted 65 PRINT M, and seen the values.

CHAPTER 5
FUNCTIONS AND SUBROUTINES

5.1 FUNCTIONS

Occasionally, you may want to calculate a function, for example, the square of a number. Instead of writing a small program to calculate this function, BASIC provides functions as part of the language, some of which are described in Chapter 1. The remaining functions are described here, in Chapter 7, and in Chapters 8 and 10.

The desired function is called by a three-letter name. The value to be used is expressed explicitly or implicitly in parentheses and follows the function name. The expression enclosed in parentheses is the argument of the function, and it is evaluated and used as indicated by the function name. For example:

```
15      LET B=SQR(4+X↑3)
```

indicates that the expression (4 + X ↑3) is to be evaluated and then the square root taken.

5.1.1 The Integer Function (INT)

The INT function appears in algebraic notation as [X] and returns the greatest integer of X that is less than or equal to X. For example:

```
INT (2.35) = 2
INT (-2.35) = -3
INT (12) = 12
```

One use of this function is to round numbers to the nearest integer by asking for INT (X + .5). For example:

```
INT (2.9 + .5) = INT (3.4) = 3
```

rounds 2.9 to 3. Another use is to round to any specific number of decimal places. For example:

```
INT (X * 10 ↑ 2 + .5) / 10 ↑ 2
```

rounds X correct to two decimal places and

INT (X * 10 ↑ D + .5) /10 ↑ D

rounds X correct to D decimal places.

5.1.2 The Random Number Generating Function (RND)

The RND function produces random numbers between 0 and 1. This function is used to simulate events that happen in a somewhat random way. RND does not need an argument.

If we want the first 20 random numbers, we can write the program shown below and get 20 six-digit decimals.

```
10      FOR L=1 TO 20
20      PRINT RND,
30      NEXT L
40      END
RUN

RANDOM          13:24           20-OCT-69

 0.406533        0.88445         0.681969        0.939462        0.253358
 0.863799        0.880238        0.638311        0.602898        0.990032
 0.863799        0.897931        0.628126        0.613262        0.303217
 5.00548E-2      0.393226        0.680219        0.632246        0.668218
```

NOTE

This is a sample run of random numbers. The format of the PRINT statement is discussed in Chapter 6.

```
RUN

RANDOM          13:25           20-OCT-69

 0.406533        0.88445         0.681969        0.939462        0.253358
 0.863799
```

A second RUN gives exactly the same random numbers as the first RUN; this is done to facilitate the debugging of programs. If we want 20 random one-digit integers, we could change line 20 to read as follows:

```
20      PRINT INT (10*RND),
RUN
```

We would obtain the following:

```
RANDOM          13:26          20-OCT-69

 4               8              6               9               2
 8               8              6               6               9
 5               8              6               6               3
 0               3              6               6               6
```

To vary the type of random numbers (20 random numbers ranging from 1 to 9, inclusive), change line 20 as follows:

```
20      PRINT  INT(9*RND  +1);
RUN

RANDOM           13:28           20-OCT-69

 4   8   7   9   3   8   8   6   6   9   6   6   3   1   4   7   6   7
```

To obtain random numbers which are integers from 5 to 24, inclusive, change line 20 to the following:

```
20      PRINT  INT(20*RND  +5);
RUN

RANDOM           13:30           20-OCT-69

 13   22   18   23   10   22   22   17   17   24   16   22   17   17   11   6
 12   18   17   18
```

If random numbers are to be chosen from the A integers of which B is the smallest, call for INT (A*RND+B).

5.1.3 The RANDOMIZE Statement

As noted when we ran the first program of this chapter twice, we got the same numbers in the same order each time. However, we get a different set with the RANDOMIZE statement, as in the following program:

```
5       RANDOMIZE
10      FOR L=1 TO 20
20      PRINT  INT(10*RND);
30      NEXT L
40      END
RUN

RNDNOS           13:32           20-OCT-69

 1   9   4   2   1   1   6   6   3   8   4   9   8   6   5   8   6   2   6   0


RUN

RNDNOS           13:33           20-OCT-69

 1   1   4   6   6   6   0   5   3   8   4   0   8   1   0   5   1   8   0   1
```

RANDOMIZE (RANDOM) resets the numbers in a random way. For example, if this is the first instruction in a program using random numbers, then repeated RUNs of the program produce different results. If the instruction is absent, then the official list of random numbers is obtained in the usual order. It is suggested that a simulated model should be debugged without this instruction so that one always obtains the same random numbers in test runs. After the program is debugged, and before starting production runs, you insert the following:

```
1       RANDOM
```

5.1.4 The Sign Function (SGN)

The SGN function is one which assigns the value 1 to any positive number, 0 to zero, and -1 to any negative number. Thus, SGN (7.23) = 1, SGN (0) = 0, and SGN (-.2387) = -1. For example, the following statement:

```
50      ON SGN(X)+2 GO TO 100,200,300
```

transfers to 100 if X < 0, to 200 if X = 0, and to 300 if X > 0.

5.1.5 The Time Function (TIM)

The TIM function returns the elapsed execution time, in seconds, of a program from the beginning of execution. This time does not include compile and load time when a single program is run. However, when programs are chained together (refer to 6.6 for a description of chaining), TIM returns the total of the elapsed execution time since the start of execution of the first program plus the compile, load and execution times of each subsequent program. The TIM function does not accept an argument. For example:

```
10      READ A, B, C
              .
              .
              .
115     IF TIM = 16 THEN 150
              .
              .
              .
150     END
```

5.1.6 The Define User Function (DEF) and Function End Statement (FNEND)

In addition to the functions BASIC provides, you may define up to 26 functions of your own with the DEF statement. The name of the defined function must be three letters, the first two of which are FN, e.g., FNA, FNB, ..., FNZ. Each DEF statement introduces a single function. For example, if you repeatedly use the function $e^{-X^2} + 5$, introduce the function by the following:

```
                30      DEF FNE(X)=EXP(-X↑2)+5
```

and call for various values of the function by FNE (.1), FNE (3.45), FNE (A+2), etc. This statement saves a great deal of time when you need values of the function for a number of different values of the variable.

The DEF statement may occur anywhere in the program, and the expression to the right of the equal sign may be any formula that fits on one line. It may include any combination of other functions, such as those defined by different DEF statements; it also can involve other variables besides those denoting the argument of the function.

As in the following example each defined function may have zero, one, two, or more numeric variables; string variables (refer to Chapter 8) are not allowed:

```
                10      DEF FNB(X,Y)=3*X*Y-Y↑3
                105     DEF FNC(X,Y,Z,W)=FNB(X,Y)/FNB(Z,W)
                530     DEF FNA=3.1416*R↑2
```

In the definition of FNA, the current value of R is used when FNA occurs. Similarly, if FNR is defined by the following:

```
70      DEF FNR(X)=SQR(2+LOG(X)-EXP(Y*Z):(X+SIN(2*Z)))
```

you can ask for FNR(2.7), and give new values to Y and Z before the next use of FNR.

The method of having multiple line DEFs is illustrated by the "max" function shown below. Using this method, the possibility of using IF ...THEN as part of the definition is a great help as shown in the following example:

```
10      DEF FNM(X,Y)
20      LET FNM=X
30      IF Y<=X THEN 50
40      LET FNM=Y
50      FNEND
```

The absence of the equals sign (=) in line 10 indicates that this is a multiple line DEF. In line 50, FNEND terminates the definition. The expression FNM, without an argument, serves as a temporary variable for the computation of the function value. The following example defines N-factorial:

```
10      DEF FNF(N)
20      LET FNF=1
30      FOR K=1 TO N
40      LET FNF=K*FNF
50      NEXT K
60      FNEND
```

Any variable which is not an argument of FN_ in a DEF loop has its current value in the program. Multiple line DEFs may not be nested and there must not be a transfer from inside the DEF to outside its range, or vice versa. GOSUB and RETURN statements (refer to Section 5.2) are not allowed in multiple line DEFs.

5.2 SUBROUTINES

When you have a procedure that is to be followed in several places in your program, the procedure may be written as a subroutine. A subroutine is a self-contained program which is incorporated into the main program at specified points. A subroutine differs from other control techniques in that the computer remembers where it was before it entered the subroutine, and it returns to the appropriate place in the main program after executing the subroutine.

5.2.1 GOSUB and RETURN Statements

Two new statements, GOSUB and RETURN, are required with subroutines. The subroutine is entered with a GOSUB statement which can appear at any place in the main program except within a multiple line DEF. The GOSUB statement is similar to a GO TO statement; however, with a GOSUB statement, the computer remembers where it was prior to the transfer. Following is an example of the GOSUB statement:

```
90      GOSUB 210
```

where 210 is the line number of the first statement in the subroutine. The last line in the subroutine is a RETURN statement which directs the computer to the statement following the GOSUB from which it transferred. For example:

```
350      RETURN
```

returns to the next highest line number greater than the GOSUB call.

Subroutines may appear anywhere in the main program except within the range of a multiple line DEF. Care should be taken to make certain that the computer enters a subroutine only through a GOSUB statement and exits via a RETURN statement.

5.2.2 Example

A program for determining the greatest common divisor (GCD) of three integers, using the Euclidean Algorithm, illustrates the use of a subroutine. The first two numbers are selected in lines 30 and 40, and their GCD is determined in the subroutine, lines 200 through 310. The GCD just found is called X in line 60; the third number is called Y, in line 70; and the subroutine is entered from line 80 to find the GCD of these two numbers. This number is, of course, the greatest common divisor of the three given numbers and is printed out with them in line 90.

A GOSUB inside a subroutine to perform another subroutine is called a nested GOSUB. It is necessary to exit from a subroutine only with a RETURN statement. You may have several RETURNs in the subroutine, as long as exactly one of them will be used.

```
10      PRINT "A", "B", "C", "GCD"
20      READ A, B, C
30      LET X=A
40      LET Y=B
50      GOSUB 200
60      LET X=G
70      LET Y=C
80      GOSUB 200
90      PRINT A,B,C,G
100     GO TO 20
110     DATA 60,90,120
120     DATA 38456,64872,98765
130     DATA 32,384,72
200          LET Q=INT(X/Y)
```

(continued on next page)

```
2&A-B        is illegal
```

A storage word may be relocatable in the left half as well as in the right half. For example:

```
XWD A,B
```

CHAPTER 6
MORE SOPHISTICATED TECHNIQUES

The preceding chapters have covered the essential elements of BASIC. At this point, you are in a position to write BASIC programs and to input these programs to the computer via your Teletype. The commands and techniques discussed so far are sufficient for most programs. This chapter and remaining ones are for a programmer who wishes to perform more intricate manipulations and to express programs in a more sophisticated manner.

6.1 MORE ABOUT THE PRINT STATEMENT

The PRINT statement permits a greater flexibility for the more advanced programmer who wishes to have a different format for his output. BASIC normally outputs items from PRINT statements in the forms described in this chapter*. Numeric items are printed in the format:

```
Snn...nb
|\___/|
|  |  └──one space
|  |
|  └──numeric value
|
└─sign:  space if positive; - if negative
```

String items (refer to Chapter 8) are printed exactly as they appear but without the enclosing quotes.

The Teletype line is divided into zones of 14 spaces each. A comma in a PRINT statement is a signal to the Teletype to move to the next print zone on the current line or, if necessary, to the beginning of the first print zone of the next line. A semicolon in a PRINT statement causes no motion of the Teletype. <PA> (page) in a PRINT statement moves the Teletype to the beginning of the first print zone of the first line on the next page of output. Commas, semicolons, and <PA> delimiters can appear in PRINT statements without intervening data items. Each delimiter causes Teletype movement as previously described. For example, PRINT A,,B causes the value of A to be printed in the first zone,

*This chapter describes the noquote mode of output. The user can explicitly change the mode to quote mode by using a QUOTE statement. Refer to Chapter 10 for the description of quote and noquote modes and their associated statements.

the Teletype to be moved to the third zone, and the value of B to be printed in the third zone. If two items in a PRINT statement are clearly distinct, the separating commas, semicolons, or <PA> delimiters can be omitted and the items are treated as though they were separated by one semicolon.

When you type in the following program:

```
10      FOR I=1 TO 15
20      PRINT I
30      NEXT I
40      END
```

the Teletype prints 1 at the beginning of a line, 2 at the beginning of the next line, and, finally, 15 on the fifteenth line. But, by changing line 20 to read as follows:

```
20      PRINT I,
```

the numbers are printed in the zones, reading as follows:

```
1              2              3              4              5
6              7              8              9              10
11             12             13             14             15
```

If you want the numbers printed in this fashion, but compressed, change line 20 by replacing the comma with a semicolon as in the following example:

```
20      PRINT I;
```

The following results are printed:

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

The end of a PRINT statement signals a new line, unless a comma or semicolon is the last symbol. Thus, the following instruction:

```
50      PRINT X, Y
```

prints two numbers and then returns to the next line, while the instruction:

```
50      PRINT X, Y,
```

prints these two values and does not return. The next number to be printed appears in the third zone, after the values of X and Y in the first two zones.

Since the end of a PRINT statement signals a new line,

```
250     PRINT
```

causes the Teletype to advance the paper one line, to put a blank line for vertical spacing of your results, or to complete a partially filled line.

```
50      FOR M=1 TO N
110     FOR J=0 TO M
120     PRINT B(M,J);
130     NEXT J
140     PRINT
150     NEXT M
```

This program prints B(1,0) and next to it B(1,1). Without line 140, the Teletype would go on printing B(2,0), B(2,1), and B(2,2) on the same line, and then B(3,0), B(3,1), etc. After the Teletype prints the B(1,1) value corresponding to M = 1, line 140 directs it to start a new line; after printing the value of B(2,2) corresponding to M = 2, line 140 directs it to start another new line, etc.

The following instructions:

```
50     PRINT "TIME-"; "SHAR"; "ING";
51     PRINT " ON"; " THE "; "PDP-10"
```

cause the printing of the following:

```
TIME-SHARING ON THE PDP-10
```

(The items enclosed in quotes in statements 50 and 51 are strings.)

The following instructions:

```
10     N=5
20     PRINT "END OF PAGE" N <PA>
30     PRINT "ITEM",,"NO. ORDERED",,"TOTAL PRICE"
```

cause the printing of

```
END OF PAGE 5
```

followed by a form-feed to position the Teletype paper at the top of a new page, where the following is printed:

```
ITEM                            NO. ORDERED                     TOTAL PRICE
```

Formatting of output can be controlled even further by means of the TAB function, in the form TAB(n), where n is the desired print position. TAB can contain any numeric formula as its argument. The value of the numeric formula is computed and then truncated to an integer. This integer is treated modulo the current output right margin. Setting the output right margin is described in Section 6.7. For example, if the output right margin is 72, which is the default margin, a value in the range 0 through 71 is obtained. The first print position on the line is column 0. Thus, TAB(17) causes the Teletype to move to column 17 (unless it has already passed this position, in which case the TAB is ignored). For example, inserting the following line in a loop

```
55      PRINT X; TAB(12); Y; TAB(27);Z
```

causes the X values to start in column 0, the Y values in column 12, and the Z values in column 27.

The following rules are used to interpret the printed results:

a. If a number is an integer, the decimal point is not printed. If the integer contains more than eight digits, it is printed in the format as follows.

```
sn.nnnnnEsp
|| \___/\_/
||   |   |___ E (Exponent) followed by the sign of the exponent,
||   |        followed by p (power of 10)
||   |
||   |_____next five digits
||
||_________first digit
|
|__sign:  space if positive; - if negative
```

For example, 32,437,580,259 is written as 3.24376E+10.

b. For any decimal number, no more than six significant digits are printed.

c. For a number less than 0.1, the E notation is used, unless the entire significant part of the number can be printed as a 6-digit decimal number. Thus, 0.03456 indicates that the number is exactly .0345600000, while 3.45600E-2 indicates that the number has been rounded to .0345600.

d. Trailing zeros after the decimal point are not printed.

The following program, in which powers of 2 are printed out, demonstrates how numbers are printed.

```
10      FOR N=-5 TO 30
20      PRINT 2↑N;
30      NEXT N
40      END

POWERS          11:54           20-OCT-69

 0.03125  0.0625  0.125  0.25  0.5  1  2  4  8  16  32  64  128  256
 512  1024  2048  4096  8192  16384  32768  65536  131072  262144
 524288  1048576  2097152  4194304  8388608  16777216  33554432
 67108864  1.32418E+8  2.68435E+8  5.36871E+8  1.07374E+9
```

6.2 INPUT STATEMENT

At times, during the running of a program, it is desirable to have data entered. This is particularly true when one person writes the program and saves it on the storage device as a library program (refer to SAVE command, Chapter 9), and other persons use the program and supply their own data. Data may be entered by an INPUT statement, which acts as a READ but accepts numbers of alphanumeric data from the Teletype keyboard. For example, to supply values for X and Y into a program, type the following:

```
40      INPUT X,Y
```

prior to the first statement which uses either of these numbers. When BASIC encounters this statement, it types a question mark. The user types two numbers, separated by a comma, and presses the RETURN key, and BASIC continues the program. No number can be longer than 8 digits.

Frequently, an INPUT statement is combined with a PRINT statement to make sure that the user knows what the question mark is asking for. You might type in the following statement:

```
20      PRINT "YOUR VALUES OF X,Y, AND Z ARE";
30      INPUT X,Y,Z
```

and BASIC types out the following:

```
YOUR VALUE OF X,Y, AND Z ARE?
```

Without the semicolon at the end of line 20, the question mark would have been printed on the next line. Data entered via an INPUT statement is not saved with the program. Therefore, INPUT should be used only when small amounts of data are to be entered, or when necessary during the running of the program.

6.3 STOP STATEMENT

STOP is equivalent to GO TO xxxxx, where xxxxx is the line number of the END statement in the program. For example, the following two program portions are exactly equivalent:

```
250     GO TO 999               250     STOP
        .........                       ....
340     GO TO 999               340     STOP
        .........                       ....
999     END                     999     END
```

6.4 REMARKS STATEMENT (REM)

REM provides a means for inserting explanatory remarks in the program. BASIC completely ignores the remainder of that line, allowing you to follow the REM with directions for using the program, with identifications of the parts of a long program, or with any other information. Although what follows REM is ignored, its line number may be used in a GO TO or IF-THEN statement as in the following:

```
100     REM INSERT IN LINES 900-998.  THE FIRST
110     REM NUMBER IS N, THE NUMBER OF POINTS.  THEN
120     REM THE DATA POINTS THEMSELVES ARE ENTERED, BY
200     REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS

        ......
300     RETURN
        .......
520     GOSUB 200
```

A second method for adding comments to a program consists of placing an apostrophe (') at the end of the line, and following it by a remark. Everything following the apostrophe is ignored. This method cannot be used in an image statement. Image statements are described in Chapter 11. Apostrophes within string constants are not treated as remark characters.

6.5 RESTORE STATEMENT

The RESTORE statement permits READing the data in the DATA statements of a program more than once. Whenever RESTORE is encountered in a program, BASIC restores the data block pointer to the first number. A subsequent READ statement then starts reading the data all over again. However, if the desired data is preceded by code numbers or parameters, superfluous READ statements should be used to pass over these numbers. As an example, the following program portion reads the data, restores the data block to its original state, and reads the data again. Note the use of line 570 (READ X) to pass over the value of N, which is already known.

```
100     READ N
110     FOR I=1 TO N
120     READ X
        ......

200     NEXT I
        ......
560     RESTORE
570     READ X
580     FOR I=1 TO N
590     READ X
        ......
700     DATA .....
710     DATA .....
```

6.6 CHAIN STATEMENT

The CHAIN statement provides a means for one program to call another program so that programs can be written separately and executed together in a chain. The CHAIN statement has one of the forms:

```
       CHAIN [alphabetic string]
or     CHAIN [alphabetic string], [numeric formula]
```

The alphabetic string is either: a) the name of the program being chained to, in the form device:filename.ext (optionally enclosed in quotes), or b) a string variable* that has as its value the name of the program being chained to, in the form device:filename.ext. The device and the extension

*A string variable is a variable that is used to store an alphabetic string. A string variable is composed of a letter and a dollar sign ($) or a letter, a number, and a dollar sign ($), e.g., A$ or B2$. String variables are described in Chapter 8.

can be omitted, but the filename must be present. If the device is omitted, DSK: is assumed; if the extension is omitted, .BAS is assumed.

The numeric formula specifies a line number in the program being chained to; its value is truncated to an integer.

A few examples of the CHAIN statement are:

```
CHAIN A$
CHAIN B2$, N*EXP(W)
CHAIN PTR:MAIN, 50
```

When BASIC encounters a CHAIN statement in a program, it stops execution of that program, retrieves the program named in the CHAIN statement from the specified device, compiles the chained program, and begins execution either at the line number specified in the CHAIN statement or at the beginning of the program if no line number was specified. Only the heading of the first program in the chain is printed, and the TIME: message is printed only after the last program in the chain has been executed. Error messages for the programs in the chain, excluding the first program, have the name of the program appended. For example:

```
OVERFLOW IN 1100 IN TEST4.BAK
```

indicates that an overflow error occurred in line 1100 in the chained program TEST4.BAK. Programs that run individually, or the first program in a chain will not have the program name appended.

The following is an example of program chaining.

```
LIST

PROG3        12:05       25-JAN-71

10      PRINT 10
11      STOP
20      PRINT 20
21      END

READY
SAVE
NEW
NEW FILENAME -- PROG2
READY
10      INPUT N
20      CHAIN PROG3, N
30      END
RUNNH
?10
 10
TIME: 0.02 SECS
```

6.7 MARGIN STATEMENT

Normally, the right margin for output to the Teletype is 72 characters. The MARGIN statement allows the user to specify a right margin of 1 to 132 characters. This margin becomes effective on the first new line of output after the MARGIN statement, and remains in effect until the next time the margin is set by a MARGIN statement or until the end of the program's execution, whichever is sooner. At the end of program execution, the output margin is reset to 72 characters.

The form of the margin statement is:

```
MARGIN [numeric formula]
```

The numeric formula is a numeric constant, variable, or expression that specifies the right margin; it is truncated to an integer before the margin is set. Some examples of the MARGIN statement are:

```
MARGIN    75
MARGIN    132*N
```

The right margin for input from the Teletype is not affected by MARGIN statements; it is always 142 characters. Lines of input that are longer than 142 characters will result in error messages.

The monitor, as well as BASIC, considers the normal Teletype output margin to be 72 characters. Therefore, when a margin greater than 72 characters is needed, the monitor command SET TTY WIDTH must be used in addition to the BASIC MARGIN statement. Otherwise, the monitor will output a leading carriage return-line feed if an attempt is made to output a seventy-third character on a line. Before the program is run, the user must issue the command:

```
MONITOR
```

to BASIC and then type:

```
SET TTY WIDTH 132
REENTER
```

to reenter BASIC. The monitor will not output its carriage return-line feed until after the 132nd character on a line; consequently, BASIC can control the margin as the MARGIN statements specify without interference from the monitor. The SET TTY WIDTH monitor command is implemented in 5.02 and later monitors.

6.8 PAGE STATEMENT

Normally, output to the Teletype is not divided into pages. The PAGE statement allows the user to set a page size of any positive number of lines. This page size remains in effect until the page size is set again by a PAGE statement, or until the Teletype is set back into nopage mode by a NOPAGE statement (described in Section 6.9), or until the end of the program's execution. At the end of program execution, the Teletype is reset to nopage mode.

The form of the PAGE statement is:

PAGE [numeric formula]

The numeric formula specifies the page size; it is truncated to an integer before the page size is set.

When a PAGE statement is executed, BASIC ends the current output line (if necessary), outputs a form-feed to position the Teletype paper at the top of the next page, and starts counting lines beginning with the next line of output. As soon as a new page is necessary, a form-feed is output. Whenever a PRINT statement containing <PA> is executed, the line count for the Teletype page is set back to zero.

6.9 NOPAGE STATEMENT

The NOPAGE statement sets the Teletype back to nopage mode (i.e., the output to the Teletype is no longer automatically divided into pages). The NOPAGE statement need only be used to change the mode back from page mode (set by a PAGE statement) because the default is nopage mode for all Teletype output. The form of the statement is:

NOPAGE

The NOPAGE statement has no effect on the execution of <PA> delimiters in PRINT statements; they are executed as usual.

CHAPTER 7
VECTORS AND MATRICES

Operations on lists and tables occur frequently; therefore, a special set of 13 instructions for matrix computations, all of which are identified by the starting word MAT, is used. These instructions are not necessary and can be replaced by combinations of other BASIC instructions, but use of the MAT instructions results in shorter programs that run much faster.

The MAT instructions are as follows:

MAT READ a, b, c	Read the three matrices, their dimensions having been previously specified.
MAT c = ZER	Fill out c with zeros.
MAT c = CON	Fill out c with ones.
MAT c = IDN	Set up c as an identity matrix.
MAT PRINT a, b, c	Print the three matrices. (Semicolons can be used immediately following any matrix which you wish to have printed in a closely packed format.)
MAT INPUT v	Call for the input of a vector.
MAT b = a	Set the matrix b equal to the matrix a.
MAT c = a + b	Add the two matrices a and b.
MAT c = a - b	Subtract the matrix b from the matrix a.
MAT c = a * b	Multiply the matrix a by the matrix b.
MAT c = TRN(a)	Transpose the matrix a.
MAT c = (k) * a	Multiply the matrix a by the number k. The number, which must be in a parentheses, may also be given by a formula.
MAT c = INV (a)	Invert the matrix a.

7.1 MAT INSTRUCTION CONVENTIONS

The following convention has been adopted for MAT instructions: while every vector has a component 0, and every matrix has a row 0 and a column 0, the MAT instructions ignore these. Thus, if we have a matrix of dimension M-by-N in a MAT instruction, the rows are numbered 1, 2, ..., M, and the columns 1, 2, ..., N.

If a numeric array is referenced in a MAT statement other than MAT INPUT, BASIC sets up the array as a matrix with two dimensions unless the user has specifically declared in a DIM (or DIMENSION) statement that the array is a vector.

The DIM statement may simply indicate what the maximum dimension is to be. Thus, if we write the following:

```
DIM M(20,35)
```

M may have up to 20 rows and up to 35 columns. This statement is written to reserve enough space for the matrix; consequently, the only concern at this point is that the dimensions declared are large enough to accomodate the matrix. However, in the absence of DIM (or DIMENSION) statements, all vectors may have up to 10 components and matrices up to 10 rows and 10 columns. This is to say that in the absence of DIM (or DIMENSION) statements, this much space is automatically reserved for vectors and matrices on their appearance in the program. The actual dimension of a matrix may be determined either when it is first set up (by a DIM statement) or when it is computed. Thus the following

```
10      DIM M(20,7)
        -----
50      MAT READ M
```

reads a 20-by-7 matrix for M, while the following:

```
50      MAT READ M(17,30)
```

reads a 17-by-30 matrix for M, provided sufficient space has been saved for it by writing

```
10      DIMENSION M(20,35)
```

7.2 MAT C = ZER, MAT C = CON, MAT C = IDN

The following three instructions:

MAT M = ZER	(sets up matrix M with all components equal to zero)
MAT M = CON	(sets up matrix M with all components equal to one)
MAT M = IDN	(sets up matrix M as an identity matrix)

act like MAT READ as far as the dimension of the resulting matrix is concerned. For example,

MAT M = CON (7,3)

sets up a 7-by-3 matrix with 1 in every component, while in the following:

MAT M = CON

sets up a matrix, with ones in every component, and a 10-by-10 dimension (unless previously given other dimensions). It should be noted, however, that these instructions have no effect on row and column zero. Thus, the following instructions:

```
10      DIM M(20,7)
20      MAT READ M(7,3)
........
35      MAT M=CON
70      MAT M=ZER(15,7)
........
90      MAT M=ZER(16,10)
```

first read in a 7-by-3 matrix for M. Then they set up a 7-by-3 matrix of all 1s for M (the actual dimension having been set up as 7-by-3 in line 20). Next they set up M as a 15-by-7 all-zero matrix. (Note that although this is larger than the previous M, it is within the limits set in 10.) An error message results because of line 90. The limit set in line 10 is (20 + 1) x (7 + 1) = 168 components, and in 90 we are calling for (16 + 1) x (10 + 1) = 187 components. Thus, although the zero rows and columns are ignored in MAT instructions, they play a role in determining dimension limits. For example,

```
90      MAT M=ZER(25,5)
```

would not yield an error message.

Perhaps it should be noted that an instruction such as MAT READ M(2,2) which sets up a matrix and which, as previously mentioned, ignores the zero row and column, does, however, affect the zero row and column. The redimensioning which may be implicit in an instruction causes the relocation of some numbers; therefore, they may not appear subsequently in the same place. Thus, even if we have first LET M(1,0) = M(2,0) = 1, and then MAT READ M(2,2), the values of M(1,0) and M(2,0) now are 0. Thus when using MAT instructions, it is best not to use row and column zero.

7.3 MAT PRINT A, B, C

The following instruction:

MAT PRINT A, B; C

causes the three matrices to be printed with A and C in the normal format (i.e., with five components to a line and each new row starting on a new line) and B closely packed.

Vectors may be used in place of matrices, as long as the above rules are observed. Since a vector like V(I) is treated as a column vector by BASIC, a row vector has to be introduced as a matrix that has only one row, namely row 1. Thus,

```
DIM X(7), Y(0,5)
```

introduces a 7-component column vector and a 5-component row vector.

If V is a vector, then

```
MAT PRINT V
```

prints the vector V as a column vector.

```
MAT PRINT V,
```

prints V as a row vector, five numbers to the line, while

```
MAT PRINT V;
```

prints V as a row vector, closely packed.

7.4 MAT INPUT V AND THE NUM FUNCTION

The following instruction:

```
MAT INPUT V
```

calls for the input of a vector. The number of components in the vector need not be specified. Normally, the input is limited by its having to be typed on one line. However, by ending the line of input with an ampersand (&) before the carriage return, the machine asks for more input on the next line. There must be at least one data item preceding the ampersand on the line or an error message will be issued. Note that, although the number of components need not be specified, if we wish to input more than 10 numbers, we must save sufficient space with a DIM statement. After the input, the function NUM equals the number of components, and V(1), V(2), ..., V(NUM) become the numbers that are input, allowing variable length input. For example,

```
5       LET S=0
10      MAT INPUT V
20      LET N=NUM
30      IF N=0 THEN 99
40      FOR I=1 TO N
45      LET S=S+V(I)
50      NEXT I
60      PRINT S/N
70      GO TO 5
99      END
```

allows the user to type in sets of numbers, which are averaged. The program takes advantage of the fact that zero numbers may be input, and it uses this as a signal to stop. Thus, the user can stop by simply pushing RETURN on an input request. If an ampersand is used, it need only be preceded by a comma when the item immediately preceding it is an unquoted string.

7.5 MAT B = A

This instruction sets up B to be the same as A and, in doing so, dimensions B to be the same as A, provided that sufficient space has been saved for B.

7.6 MAT C = A + B AND MAT C = A - B

For these instructions to be legal, A and B must have the same dimensions, and enough space must be saved for C. These statements cause C to assume the same dimensions as A and B. Instructions such as MAT A = A ± B are legal; the indicated operation is performed and the answer stored in A. Only a single arithmetic operation is allowed; therefore, MAT D = A + B - C is illegal but may be achieved with two MAT instructions.

7.7 MAT C = A * B

For this instruction to be legal, it is necessary that the number of columns in A be equal to the number of rows in B. For example, if matrix A has dimension L-by-M and matrix B has dimension M-by-N, then C = A * B has dimension L-by-N. It should be noted that while MAT A = A + B may be legal, MAT A = A * B is self-destructive because, in multiplying two matrices, we destroy components which would be needed to complete the computation. MAT B = A * A is, of course, legal provided that A is a "square" matrix.

7.8 MAT C = TRN(A)

This instruction lets C be the transpose of the matrix A. Thus, if matrix A is an M-by-N matrix, C is an N-by-M matrix. The instruction MAT C = TRN (C) is legal.

7.9 MAT C = (K) * A

This instruction allows C to be the matrix A multiplied by the number K (i.e., each component of A is multiplied by K to form the components of C). The number K, which must be in parentheses, may be replaced by a formula. MAT A = (K) * A is legal.

7.10 MAT C = INV(A) AND THE DET FUNCTION

This instruction allows C to be the inverse of A. (A must be a "square" matrix.) The function DET is available after the execution of the inversion, and it will equal the determinant of A. Consequently, the user can obtain the determinant of a matrix by inverting the matrix and then noting what value DET has. If the determinant of a matrix is zero, the matrix is singular and its inverse is meaningless. When an attempt is made to invert a matrix whose determinant is zero, the warning message is printed,

```
            % SINGULAR MATRIX INVERTED IN nn
```

DET is set equal to zero, and the program execution continues.

7.11 EXAMPLES OF MATRIX PROGRAMS

The first example reads in A and B in line 30 and, in so doing, sets up the correct dimensions. Then, in line 40, A + A is computed and the answer is called C. This automatically dimensions C to be the same as A. Note that the data in line 90 results in A being 2-by-3 and in B being 3-by-3. Both MAT PRINT formats are illustrated, and one method of labeling a matrix print is shown.

```
10      DIM A(20,20), B(20,20), C(20,20)
20      READ M,N
30      MAT READ A(M,N),B(N,N)
40      MAT C=A+A
50      MAT PRINT C;
60      MAT C=A*B
70      PRINT
75      PRINT "A*B=",
80      MAT PRINT C
90      DATA 2,3
91      DATA 1,2,3
92      DATA 4,5,6
93      DATA 1,0,-1
94      DATA 0,-1,-1
95      DATA -1,0,0
99      END
RUN

MATRIX          08:31          09-MAR-71



 2   4   6

 8   10  12


A*B=

-2              -2              -3

-2              -5              -9


TIME: 0.13 SECS.
```

The second example inverts an n-by-n Hilbert matrix:

```
1        1/2        1/3 . . .     1/n
1/2      1/3        1/4 . . .     1/n + 1
1/3      1/4        1/5 . . .     1/n + 2
 .        .         . . . .
 .        .         . . . .
 .        .         . . . .
1/n      1/n + 1     1/n + 2      1/2n-1
```

Ordinary BASIC instructions are used to set up the matrix in lines 50 to 90. Note that this occurs after correct dimensions have been declared. A single instruction then results in the computation of the inverse, and one more instruction prints it. Because the function DET is available after an inversion, it is taken advantage of in line 130, and is used to print the value of the determinant of A. In this example, we have supplied 4 for N in the DATA statement and have made a run for this case:

```
5       REM THIS PROGRAM INVERTS AN N-BY-N HILBERT MATRIX
10      DIM A(20,20), B(20,20)
20      READ N
30      MAT A=CON(N,N)
50      FOR I=1 TO N
60      FOR J=1 TO N
70      LET A(I,J)=1/(I+J-1)
80      NEXT J
90      NEXT I
100     MAT B=INV(A)
115     PRINT "INV(A)="
120     MAT PRINT B
125     PRINT
130     PRINT "DETERMINANT OF A=" DET
190     DATA 4
199     END
RUN

HILMAT          13:52           20-OCT-69

INV(A)=

 16.0001        -120.001         240.003        -140.002
-120.001         1200.01        -2700.03         1680.02
 240.003        -2700.03         6480.08        -4200.05
-140.002         1680.02        -4200.05         2800.03

DETERMINANT OF A=1.65342E-7
```

A 20-by-20 matrix is inverted in about 0.5 seconds. However, the reader is warned that beyond n = 7, the Hilbert matrix cannot be inverted because of severe round-off errors.

7.12 SIMULATION OF N-DIMENSIONAL ARRAYS

Although it is not possible to create n-dimensional arrays in BASIC, the method outlined below does simulate them. The example is of a three-dimensional array, but it has been written in such a way that it could be easily changed to four dimensions or higher. We use the fact that functions can have any number of variables, and we set up a 1-to-1 correspondence between the components of the array and the components of a vector which equals the product of the dimensions of the array. For example, if the array has dimensions 2, 3, 5, then the vector has 30 components. A multiple line DEF could be used in place of the simple DEF in line 30 if the user wished to include error messages. The printout is in the form of two 3-by-5 matrices.

```
10      DIM V(1000)
20      MAT READ D(3)
30      DEF FNA(I,J,K)=((I-1)*D(2)+(J-1))*D(3)+K
50      FOR I=1 TO D(1)
55      FOR J=1 TO D(2)
60      FOR K=1 TO D(3)
80      LET V(FNA(I,J,K))=I+2*J+K↑2
90      PRINT V(FNA(I,J,K)),
100     NEXT K
110     NEXT J
112     PRINT
115     PRINT
120     NEXT I
900     DATA 2,3,5
999     END
RUN

3ARRAY          08:07           27-OCT-69
 4               7               12              19              28
 6               9               14              21              30
 8               11              16              23              32

 5               8               13              20              29
 7               10              15              22              31
 9               12              17              24              33
```

CHAPTER 8
ALPHANUMERIC INFORMATION (STRINGS)

In previous chapters, we have dealt only with numerical information. However, BASIC also processes alphanumeric information in the form of strings. A string is a sequence of characters, each of which is a letter, a digit, a space, or some other character. A string, however, cannot contain a character that is a line terminator (i.e., a line feed, form feed, or vertical tab), or a carriage return.

String constants are normally enclosed in quotes (e.g., "TOTAL VALUE"). In some cases in some statements, the quotes can be omitted. Where this is allowed, it is explicitly stated in the description of the particular type of statement found elsewhere in this manual.

Variables may be introduced for simple strings and string vectors, but not for string matrices. Any simple variable, followed by a dollar sign ($), stands for a string; e.g., A$ and C7$. A vector variable, followed by $, denotes a list of strings; e.g., V$(n) or A2$(n), where n is the nth string in the list. For example, V$(7) is the seventh string in the list V.

8.1 READING AND PRINTING STRINGS

Strings may be read and printed. For example:

```
10      READ A$, B$, C$
20      PRINT C$; B$; A$
30      DATA ING,SHAR,TIME-
40      END
```

causes TIME-SHARING to be printed. The effect of the semicolon in the PRINT statement is consistent with that discussed in Chapter 6; i.e., it causes output of the alphanumeric items in a close-packed form. Commas, <PA> delimiters, and TABs may be used as in any other PRINT statement. The loop:

```
70      FOR I=1 TO 12
80      READ M$(I)
90      NEXT I
```

reads a list of 12 strings.

In place of the READ and PRINT, corresponding MAT instructions may be used for lists. For example, MAT PRINT M$; causes the members of the list to be printed without spaces between them. We may also use INPUT or MAT INPUT. After a MAT INPUT, the function NUM equals the number of strings inputted. When using the MAT INPUT statement, you can continue inputting strings on the next line by typing an ampersand (&) on the current line immediately before pressing the RETURN key. A comma must precede the ampersand if the string immediately before the ampersand is unquoted. If the string is unquoted and a comma does not separate the string from the ampersand, the ampersand will be treated as part of the string. Thus, either MARY,& or "MARY"& is legal input.

As usual, lists are assumed to have no more than 10 elements; otherwise, a DIM (or DIMENSION) statement is required. The following statement:

```
10      DIM M$(20)
```

saves space for 20 strings in the M$ list.

In the DATA statements, numbers and strings may be intermixed. Numbers are assigned only to numerical variables, and strings only to string variables. Strings in DATA statements are recognized by the fact that they start with a letter. If they do not, they must be enclosed in quotes. The same requirement holds for a string containing a comma. For example:

```
90      DATA 10,ABC,5,"4FG","SEPT. 22, 1968",2
```

The only convention on INPUT and MAT INPUT is that a string containing a comma must be enclosed in quotes. The following example shows the correct format for a response to a MAT INPUT:

MR. JONES, "146 MAIN ST., MAYNARD, MASS."

8.2 STRING CONVENTIONS

In every method of inputting string information into a program (DATA, INPUT, MAT INPUT, etc.), leading blanks are ignored unless the string, including the blanks, is enclosed in quotes. String constants (which must be enclosed in quotes) or string variables may occur in LET and IF-THEN statements. The following two examples are self-explanatory:

```
10      LET Y$="YES"
20      IF A7$="YES" THEN 200
```

The relation "<" is interpreted as "earlier in alphabetic order." The other relational symbols work in a similar manner. In any comparison, trailing blanks in a string are ignored, as in the following:

"YES" = "YES "

We illustrate these possibilities by the following program, which reads a list of strings and alphabetizes them:

```
10      DIM L$(50)
20      READ N
30      MAT READ L$(N)
40      FOR I=1 TO N
50      FOR J=1 TO N-I
60      IF L$(J) < L$(J+1) THEN 100
70      LET A$=L$(J)
80      LET L$(J)=L$(J+1)
90      LET L$(J+1)=A$
100     NEXT J
110     NEXT I
120     MAT PRINT L$
900     DATA 5,ONE,TWO,THREE,FOUR,FIVE
999     END
```

Omitting the $ signs in this program serves to read a list of numbers and to print them in increasing order.

A rather common use is illustrated by the following:

```
330     PRINT "DO YOU WISH TO CONTINUE";
340     INPUT A$
350     IF A$="YES" THEN 10
360     STOP
```

8.3 NUMERIC AND STRING DATA BLOCKS

Numeric and string data are kept in two separate blocks, and these act independently of each other. The RESTORE statement resets both the data pointers for the numerical data and string data back to the beginning of their blocks. RESTORE* resets the pointer only for the numerical data and RESTORE $ only for the string data.

8.4 THE CHANGE STATEMENT

In BASIC, it is very easy to obtain the individual digits in a number by using the function INT. One way to obtain the individual characters in a string is with the instruction CHANGE. The use of CHANGE is best illustrated with the following examples.

```
5      DIM A(65)
10     READ A$
15     CHANGE A$ TO A
20     FOR I=0 TO A(0)
25     PRINT A(I);
35     NEXT I
40     DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
45     END
RUN

CHANGE         13:55          20-OCT-69

 26   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79
 80   81   82   83   84   85   86   87   88   89   90
```

In line 15, the instruction CHANGE A$ TO A has caused the vector A to have as its zero component the number of characters in the string A$ and, also, to have certain numbers in the other components. These numbers are the American Standard Code for Information Interchange (ASCII) numbers for the characters appearing in the string (e.g., A(1) is 65 - the ASCII number for A).

Table 8-1 lists the ASCII numbers for printing and nonprinting characters. Note that the nonprinting characters are shown in the table as codes containing two or three letters. These codes are not output; the actual meaning of the ASCII number is output (e.g., 7 causes the bell to ring, it does not print BEL).

Table 8-1
ASCII Numbers and Equivalent Characters

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
0	NUL	Null	14	SO	Shift out
1	SOH	Start of heading	15	SI	Shift in
2	STX	Start of text	16	DLE	Data link escape
3	ETX	End of text	17	DC1	Device control 1
4	EOT	End of transmission	18	DC2	Device control 2
5	ENQ	Enquiry	19	DC3	Device control 3
6	ACK	Acknowledge	20	DC4	Device control 4
7	BEL	Bell	21	NAK	Negative acknowledgement
8	BS	Backspace	22	SYN	Synchronous idle
9	HT	Horizontal tab	23	ETB	End of transmission block
10	LF	Line feed	24	CAN	Cancel
11	VT	Vertical tab	25	EM	End of medium
12	FF	Form feed	26	SUB	Substitute
13	CR	Carriage return	27	ESC	Escape
Note: Recall that line feed (LF), form feed (FF), vertical tab (VT), and carriage return (CR) are illegal in strings.					

(continued on next page)

Table 8-1 (Cont)
ASCII Numbers and Equivalent Characters

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
28	FS	File separator	72	H	Upper case H
29	GS	Group separator	73	I	Upper case I
30	RS	Record separator	74	J	Upper case J
31	US	Unit separator	75	K	Upper case K
32	SP	Space or blank	76	L	Upper case L
33	!	Exclamation mark	77	M	Upper case M
34	"	Quotation mark	78	N	Upper case N
35	#	Number sign	79	O	Upper case O
36	$	Dollar sign	80	P	Upper case P
37	%	Percent sign	81	Q	Upper case Q
38	&	Ampersand	82	R	Upper case R
39	'	Apostrophe	83	S	Upper case S
40	(	Left parenthesis	84	T	Upper case T
41	)	Right parenthesis	85	U	Upper case U
42	*	Asterisk	86	V	Upper case V
43	+	Plus sign	87	W	Upper case W
44	,	Comma	88	X	Upper case X
45	-	Minus sign or hyphen	89	Y	Upper case Y
46	.	Period or decimal point	90	Z	Upper case Z
47	/	Slash	91	[	Left square bracket
48	0	Zero	92	\	Back slash
49	1	One	93	]	Right square bracket
50	2	Two	94	^ or ↑	Circumflex or up arrow
51	3	Three	95	← or _	Back arrow or underscore
52	4	Four	96	`	Grave accent
53	5	Five	97	a	Lower case a
54	6	Six	98	b	Lower case b
55	7	Seven	99	c	Lower case c
56	8	Eight	100	d	Lower case d
57	9	Nine	101	e	Lower case e
58	:	Colon	102	f	Lower case f
59	;	Semicolon	103	g	Lower case g
60	<	Left angle bracket	104	h	Lower case h
61	=	Equal sign	105	i	Lower case i
62	>	Right angle bracket	106	j	Lower case j
63	?	Question mark	107	k	Lower case k
64	@	At sign	108	l	Lower case l
65	A	Upper case A	109	m	Lower case m
66	B	Upper case B	110	n	Lower case n
67	C	Upper case C	111	o	Lower case o
68	D	Upper case D	112	p	Lower case p
69	E	Upper case E	113	q	Lower case q
70	F	Upper case F	114	r	Lower case r
71	G	Upper case G	115	s	Lower case s

(continued on next page)

Table 8-1 (Cont)
ASCII Numbers and Equivalent Characters

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
116	t	Lower case t	122	z	Lower case z
117	u	Lower case u	123	{	Left brace
118	v	Lower case v	124	|	Vertical line
119	w	Lower case w	125	}	Right brace
120	x	Lower case x	126	~	Tilde
121	y	Lower case y	127	DEL	Delete

The other use of CHANGE is illustrated by the following:

```
10      FOR I=0 TO 5
15      READ A(I)
20      NEXT I
25      DATA 5,65,66,67,68,69
30      CHANGE A TO A$
35      PRINT A$
40      END
```

This program prints ABCDE because the numbers 65 through 69 are the code numbers for A through E. Before CHANGE is used in the vector-to-string direction, we must give the number of characters which are to be in the string as the zero component of the vector. In line 15, A(0) is read as 5. The following is a final example:

```
5       DIM V(128)
10      PRINT "WHAT DO YOU WANT THE VECTOR V TO BE";
20      MAT INPUT V
30      LET V(0)=NUM
35      IF NUM=0 THEN 70
40      CHANGE V TO A$
50      PRINT A$
60      GO TO 10
70      END
RUN

EXAMPLE         13:59           20-OCT-69

WHAT DC YOU WANT THE V TO BE? 40,45,60,45,89,90
(-<-YZ
WHAT DO YOU WANT THE VECTOR V TO BE? 33,34,35,36,37,38,39,40,41,42 &
? 43,44,45,46,47,48,49,50
!"#$%&'()*+,-./012
WHAT DO YOU WANT THE VECTOR V TO BE?
TIME: 0.10 SECS.
```

Note that in this example we have used the availability of the function NUM after a MAT INPUT to find the number of characters in the string which is to result from line 40.

8.5 STRING CONCATENATION

Strings can be concatenated by means of the plus sign operator (+). The plus sign can be used to concatenate string formulas wherever a string formula is legal, with the exception that information cannot be stored by means of LET or CHANGE statements in concatenated string variables. That is, concatenated string variables cannot appear to the left of the equal sign in a LET statement or as the right-hand argument in a CHANGE statement. For example, LET A$=B$+C$ is legal, but LET A$+B$=C$ is not; and similarly, CHANGE A$+B$ TO X is legal, but CHANGE X TO A$+B$ is not. An example of string concatenation is:

```
10      INPUT A$
20      CHAIN A$+"MAIN.PRG"
30      END
RUNNH

?DTA4:
```

The program causes chaining to DTA4:MAIN.PRG, which is the program MAIN.PRG on DECtape drive 4.

8.6 STRING MANIPULATION FUNCTIONS

A number of functions have been implemented that perform manipulations on strings. These functions are LEN, ASC, CHR$, VAL, STR$, LEFT$, RIGHT$, MID$, SPACE$, and INSTR. Functions that return strings have names that end in a dollar sign ($); those functions that return numbers have names that do not end in a dollar sign.

8.6.1 The LEN Function

The LEN function returns the number of characters in a string. It has the form:

LEN (string formula)

Examples:

```
10      READ A$, B$
20      PRINT LEN(A$+B$+"AROUND")
30      DATA "UP, ", "DOWN, AND "
40      END
RUNNH
20

10      IF LEN (A$)<>0 THEN 30
20      PRINT "A$ IS A NULL STRING"
30      END
```

8.6.2 The ASC and CHR$ Functions

The ASC and CHR$ functions perform conversion of ASCII numbers in the same manner as the CHANGE statement. The ASC function converts one character to its ASCII decimal equivalent, and the CHR$ function converts an ASCII decimal number to its equivalent character.

The ASC function has the form:

ASC (argument)

The argument can be either one character or the two- or three-letter code that represents a nonprinting character (refer to Table 8-1 for these codes). ASC returns the equivalent ASCII decimal number for the character.

The CHR$ function has the form:

CHR$ (numeric formula)

The value of the numeric formula is truncated to an integer that must be in the range 0 through 127 and cannot be the numbers 10 through 13. If the integer is less than 0 or greater than 127 or one of the numbers 10 through 13, an error message is issued. This integer is then interpreted as an ASCII decimal number that is converted to its equivalent character (refer to Table 8-1 for the ASCII numbers and the equivalent characters).

An example of the ASC and CHR$ functions follows.

```
5       FOR T=ASC(A) TO ASC(A)+3
10      PRINT "THIS IS TEST " + CHR$(T)
```

This is the beginning of a FOR loop that successively prints:

```
THIS IS TEST A
        .
        .
        .
THIS IS TEST B
        .
        .
        .
THIS IS TEST C
        .
        .
        .
THIS IS TEST D
```

8.6.3 The VAL and STR$ Functions

The VAL and STR$ functions perform conversions from numbers to strings and strings to numbers. The form of the VAL function is:

VAL (string formula)

The string formula must look like a number; if it does not, an error message is issued. VAL returns the actual number that the string represents. The VAL function does not return the ASCII value of the number that the string represents, it returns the number. For example, VAL ("25") returns the number 25. The 25 that is the argument to VAL is a string, the 25 that VAL returns is a number.

If the string argument represents a number that is greater than about 1.7E38 in magnitude or non-zero, but less than about 1.4E-39 in magnitude, the appropriate overflow or underflow message is issued and the value returned is about 1.7E38, about -1.7E38, or zero, whichever is appropriate.

Example:

```
10      INPUT A$
20      PRINT VAL (A$)*2
         .
         .
         .
100     END
RUNNH

?2.4611121
 4.92222
```

The STR$ function returns the string representation (as a number) of its argument. The form of STR$ is:

STR$ (numeric formula)

The string that is returned is in the form in which numbers are output in BASIC (see Section 6.1). For example, PRINT STR$ (1.76111124) prints the string 1.76111.

Examples:

```
10      A=2561
20      B$=STR$(A)
30      PRINT B$
40      END
RUNNH

2561


10      A=25
20      B$=STR$(A)
30      CHANGE B$ TO X
40      PRINT X(0); X(1); X(2)
50      END
RUNNH

 2  50  53
```

8.6.4 The LEFT$, RIGHT$, and MID$ Functions

The LEFT$, RIGHT$, and MID$ functions return substrings of their string arguments.

The LEFT$ function returns a substring of a specified number of characters starting with the leftmost character of its string argument. The LEFT$ function has the form:

LEFT$ (string formula, numeric formula)

The value of the numeric formula is truncated to an integer that specifies the number of characters in the substring. If the specified number of characters is greater than the length of the string argument, the entire string is returned. If the specified number of characters is less than or equal to zero, an error message is issued. For example,

```
10      PRINT LEFT$("THIS IS A TEST",7)
```

prints the substring

```
THIS IS
```

The RIGHT$ function returns a substring of specified length ending at the rightmost character of its string argument. The form of the RIGHT$ function is:

RIGHT$ (string formula, numeric formula)

The value of the numeric formula is truncated to an integer that specifies the number of characters in the substring to be returned. If the number of characters is greater than the length of the string argument, the entire string is returned. If the specified number of characters is less than or equal to zero, an error message is issued. For example,

```
5       A$="HERE AND THERE"
10      PRINT RIGHT$(A$,5)
```

prints the substring

```
THERE
```

The MID$ function returns a substring of its string argument starting a specified number of characters from the leftmost character of the string argument. The number of characters in the substring can also be specified. The form of the MID$ function is:

MID$ (string formula, numeric formula-1, numeric formula-2)

The second numeric formula, which is truncated to an integer that specifies the number of characters in the substring, is optional and can be omitted. If this argument is omitted, the substring includes all the remaining characters in the string argument. The first numeric formula is truncated to an integer that specifies the leftmost character at which the substring is to start. MID$ returns a null string if the first numeric formula when truncated to an integer is greater than the number of characters in the string argument; if it is less than or equal to zero, an error message is issued. If the number of characters in the substring is specified (by the second numeric formula) and is greater than the number of characters in the string argument beginning at the specified character, MID$ returns the string argument starting at the specified character. If the number of characters is less than or equal to zero, an error message is issued.

Examples:

```
10      PRINT MID$ ("TOTAL OUTPUT IN MARCH",17)
                .
                .
                .

RUNNH
MARCH

10      PRINT MID$ ("ABCDEF",3,1)
                .
                .
                .

RUNNH
C
```

8.6.5 The SPACE$ Function

The SPACE$ function returns a string of spaces. The form of the SPACE$ function is:

SPACE$ (numeric formula)

The value of the numeric formula is truncated to an integer that specifies the number of spaces in the string to be returned. If the integer is less than or equal to zero or greater than 132, an error message is issued.

Example:

```
10      A$=B$="HERE"
20      FOR T=1 TO 3
30      PRINT A$; SPACE$(T); B$
                  .
                  .
                  .

RUNNH


HERE HERE
HERE  HERE
HERE   HERE
```

8.6.6 The INSTR Function

The INSTR function searches for a specified substring within a string and returns the position of the first character of that substring within the string. The positions are numbered from the leftmost character in the string. The user can optionally specify that the search for the substring begin at a character position other than the first. The form of the INSTR function is:

INSTR (numeric formula, string formula-1, string formula-2)

The numeric formula, which is truncated to an integer that specifies the starting character position, is optional and can be omitted. If the numeric argument is omitted, the search begins at the first character position. The first string argument is the string searched; the second string argument is the substring searched for. If the value of the numeric formula (if specified) is greater than the number of characters in the string or if the substring cannot be found in the string, INSTR returns a value of zero. If the value of the numeric formula is less than or equal to zero, an error message is issued.
If the second string argument is a null string, INSTR returns the character position at which the search started, unless that position is past the last character in the string. In that case, INSTR returns a value of zero.

Examples:

```
10      PRINT INSTR ("ABCDCEF", "C")
                    .
                    .
                    .

RUNNH

3

10      PRINT INSTR (4,"ABCDCEF", "C")
                    .
                    .
                    .

RUNNH

5
```

Note that if the second string argument occurs more than once within that part of the first string argument that is searched, the first occurrence found is used.

CHAPTER 9
EDIT AND CONTROL

There are BASIC commands which:

1. Create, edit and manipulate files,
2. Run BASIC programs,
3. Cause the user to enter monitor mode,
4. Obtain information, and
5. Set the input mode.

These commands operate on an entire BASIC program, and therefore are functionally different from the BASIC statements which comprise the program. For example; typing the LENGTH command causes BASIC to output the length (in characters) of the current program in the user's core memory. If, however, the user includes the LENGTH command as a statement in his BASIC program, an error is generated and the program cannot run.

9.1 CREATING THE FILE IN CORE MEMORY

A file is a collection of data. This data may be BASIC statements, thereby comprising a BASIC program; it may be data for a BASIC program, or it may be a combination of a program and data. Seven media are used to store files. They are:

1. Core memory,
2. Disk pack and drum,
3. DECtape and magnetic tape, and
4. Paper tape and punch cards.

At the time the user types R BASIC, a core memory area is allocated for his use and cleared of any previous user's files. Core memory may be thought of as a working storage area. Any work done on a file is performed in core memory, however, the user may not keep files in core memory for a prolonged period of time. Permanent storage of that nature is reserved for storage devices such as disk, magnetic tape, DECtape, paper tape and punch cards.

In order to create a new file, edit an existing file or run a file containing a BASIC program, the file must first be established in the user's core memory. The NEW command, the OLD command and the default to NONAME provide the means by which a file is established in core memory. Refer to the WEAVE command in Paragraph 9.3 for an additional method of moving files into core memory.

NEW filename.ext

The user gives the NEW command to establish a new file in core memory. This file is given the name filename.ext. Before establishing the new file, BASIC clears the user's core memory. Thus, the file previously in user core (if any) is destroyed. (To retain the file, a SAVE command should be issued before the NEW command is given. Refer to Paragraph 9.3.)

When issuing the NEW command, filename.ext may be omitted. In this case BASIC asks for the filename.ext by typing:

```
NEW FILE NAME--
```

The user types the filename.ext.

If the extension is not included (i.e., .ext is left out) BASIC assumes it is .BAS. If a carriage return is substituted for filename.ext, BASIC types ?WHAT? and disregards the NEW command.

Once the NEW command is given, BASIC establishes the file by clearing the user's core and assigning the filename. When it is ready to accept the contents of the file, BASIC types READY. The user then inputs the file simply by typing it on the terminal.

When the user is finished inputting the new file, he types ↑C to return to the BASIC user mode (he was in input mode).

The user should be aware that at this point the new file is only in core memory and not in permanent storage. This means that a command which clears the user's core memory (for example; a BYE, NEW or OLD command) destroys the file the user just input. The SAVE and REPLACE commands (described in Paragraph 9.4) store a file on the disk.

```
READY
NEW RESIS.BAS                           Establish a new file called RESIS.BAS.

READY
10  INPUT R1, R2, R3                                  Type the file.
20  R= (R1*R2*R3) / ((R2*R3) + R1*(R2+R3))
30  PRINT "THE PARALLEL RESISTANCE = "; R
40  END
SAVE                                    Put RESIS.BAS on disk storage.

READY
```

OLD dev:filename.ext

By using the OLD command, the user replaces the file in core memory with one from a storage device. As with the NEW command, the contents of the user's core memory are cleared before the designated file is brought in. The storage device on which the file is located is given by dev:. Omitting dev: causes the default device (i.e., disk) to be selected. The file is identified by filename.ext. If omitted, BASIC requests the filename.ext by typing:

```
OLD FILE NAME--
```

The user should then type the filename.ext. If he presses carriage return, the command aborts and returns to BASIC user level. If the extension is omitted .BAS is assumed. The file obtained from the device must have line numbers. The indicated filename is now the current filename.

Retrieving a file from a device in this manner does not delete the file on the source storage device. However, if the user modifies the file in core memory, thereby creating a new version of that file, the new version is not retained on a permanent storage device until a SAVE, REPLACE or COPY command is executed. (Refer to Paragraph 9.4.)

```
READY
OLD DTA1: SOS.OUT        Bring a copy of the file SOS.OUT, which is
                         presently on DTA1:, into user core.
READY
```

Default to NONAME

There is a third way to establish a file in core memory. After BASIC responds with READY, the user may start typing the contents of his file. By this action the user is adding to the material in core memory without assigning a new name to core memory and without initially clearing the contents of core. If no filename is assigned to core memory (as is the case after issuing R BASIC), it takes the default name NONAME. Any action the user takes with the file should use NONAME unless a RENAME command is given.

```
.R BASIC

READY, FOR HELP TYPE HELP
05  INPUT E1                                               Type in contents of
10  INPUT R                                                file.
25  I=E1/R
35  PRINT "THE EQUIVALENT CURRENT IS",I, " AMPERES"
40  END
LIST                                                       Request output of
                                                           file in user core.

NONAME         14:06          12-SEPT-73                   Filename is called
                                                           NONAME since no
                                                           filename was specified.
05  INPUT E1
10  INPUT R
25  I=E1/R
35  PRINT "THE EQUIVALENT CURRENT IS",I, " AMPERES"
40  END

READY
```

The RENAME command alters the name of the file in core memory. This function is useful especially after a default to NONAME.

RENAME dev:filename.ext

The name of the user's file in core memory is changed to dev:filename.ext when the user issues the RENAME command.

If dev: and/or .ext are left out when the command is given, the original dev: and/or .ext are kept.

```
READY
RENAME EQUIV.BAS

READY
```

9.2 LISTING FILES

Often the contents of a file in the user's core memory or of a file in permanent storage must be examined. The LIST, QUEUE and ↑O commands produce a printed copy of the desired file. In addition, refer to the COPY command in Paragraph 9.4.

```
LIST  range, range, ...
LISTNH  range, range, ...
```

The LIST and LISTNH commands, given without any range arguments, print the entire contents of a file in the user's core memory area. LIST prints a one-line heading which includes the name of the file, the time and the date. LISTNH prints the specified lines without the heading.

If only part of the core storage file is desired, range arguments are used to identify the desired lines. If more than one part is needed, additional range arguments can be added provided that each succeeding range specification is separated by a comma.

Range arguments are permitted in one of two forms:

n	A single line is printed when its line-number, n, is used as a range argument.
x-y	A group of lines is output when the range argument is put in the form x-y, where x is the line-number of the first line in the group, and y is the line-number of the last line in the group.

The lines are printed on the user's terminal in order of ascending line-numbers.

```
LISTREVERSE
LISTNHREVERSE
```

LISTREVERSE and LISTNHREVERSE print the contents of the user's core memory area in order of descending line numbers. LISTREVERSE precedes the output with a heading, LISTNHREVERSE eliminates the heading.

```
LISTREVERSE

EQUIV.BAS       14:15           12-SEPT-73


40   END
35   PRINT "THE EQUIVALENT CURRENT IS",I, " AMPERES"
25   I=E/R
10   INPUT R
05   INPUT E1

READY
```

↑O

↑O suppresses the output of a file. BASIC responds with READY after termination.

QUEUE filename.ext/UNSAVE/nCOPIES/LIMITm

The QUEUE command causes the specified file to be printed on the line printer. This file must have been previously stored on disk by a SAVE, REPLACE or COPY command. The file in core storage is not affected by this command.

If the extension of the filename is omitted, .BAS is assumed.

The three optional switches /UNSAVE, /nCOPIES, and /LIMITm can be included in any order. UNSAVE and LIMIT can be abbreviated to as little as U and L respectively while the word COPIES can be omitted entirely. For example, QUEUE RETURN/U/L12/2 tells BASIC to list two copies of the file RETURN.BAS, but not exceed 12 pages in doing so. The file is deleted (unsaved).

When the /UNSAVE switch is given, the file is immediately removed from the user's permanent (disk) storage area, then it is listed. Without this switch the file is retained. The n/COPIES switch causes n copies of the file to be printed to a maximum of 63 copies. Without this switch, one copy is printed. The /LIMITm switch indicates the maximum number of line printer pages that can be printed. Without this switch 200 pages is the limit. The arguments n and m must be integers.

More than one file listing can be requested by placing a comma between each succeeding filename and its associated switches.

```
QUEUE EQUIV.BAS/2COPIES

FILES QUEUED:
EQUIV.BAS

READY
```

Request 2 copies of the file EQUIV.BAS be output on the line printer.

9.3 EDITING A FILE IN CORE MEMORY

After a file has been entered into the user's core memory by a NEW command, OLD command or a default to NONAME, the user may want to edit the file to eliminate any errors.

9.3.1 Replacing Complete Lines

The user inserts new lines and replaces existing lines by first typing the appropriate line-number and following it with a line of text.

<u>Type Line Number</u>

When BASIC is in user mode, typing a line number followed by text can have one of two consequences.

1. If there is no existing line associated with that line-number, BASIC enters both the new line-number and the line into the file in the user's core memory.
2. If there is already a line in the core file with the specified line-number, that line is deleted and the new line is inserted in its place.

Simply typing a line number without any text establishes a blank line.

9.3.2 Deleting Lines

$\boxed{\text{DELETE range, range, ...}}$

The DELETE command eliminates lines from the user's file in core memory. The range arguments specify which line(s) are to be eliminated.

Range arguments are permitted in one of two forms:

n A single line is deleted when its line-number, n, is used as a range argument.

x-y A group of lines is deleted when the range argument is put in the form x-y, where x is the line-number of the first line in the group, and y is the line-number of the last line in the group.

```
DELETE 125, 250-425, 900        Delete line 125, lines 250 thru 425
                                inclusive and line 900.
READY
```

9.3.3 Renumbering Lines in the Core File

RESEQUENCE n,f,k

RESEQUENCE modifies the line-numbers of the file in user core. Line-number f is changed to line-number n. Succeeding lines are then incremented by k. f must be less than n.

When f is omitted entirely the first line of the file takes line-number n, and succeeding lines take line-numbers n+k, n+2k, and so forth. Even though f is omitted, both commas are retained.

In cases where a single argument, n, is given, the first line-number is changed to n and succeeding line-numbers are produced, incrementing by 10.

```
RESEQUENCE 100,25,100        Renumber line 25 to 100 and number
                             succeeding lines incrementing by 100.
READY
```

9.3.4 Clearing the Entire File

SCRATCH

The SCRATCH command deletes all line-numbers and their associated lines from user core. The name associated with the file in core is kept the same.

```
SCRATCH
READY
```

9.3.5 Merging Another File Into the File

```
WEAVE dev:filename.ext
```

The WEAVE command locates a file with the name filename.ext on dev:. This file is then merged into the file in user core. If two lines have the same line-number, the line in the file named in the WEAVE command replaces the line in the file in user core. Otherwise, the lines from the file are merged in sequential line-number order into the file in user core.

```
OLD RESIS.BAS          Take a copy of file RESIS.BAS from disk storage and
                       put it in user core.
READY
LISTNH                 List RESIS.BAS

10  INPUT R1, R2, R3
20  R= (R1*R2*R3) / ((R2*R3) + R1*(R2+R3))
30  PRINT "THE PARALLEL RESISTANCE = "; R
40  END

READY
OLD EQUIV.BAS          Take EQUIV.BAS and put it in user core. (RESIS.BAS
                       is cleared from user core.)
READY
LISTNH                 List EQUIV.BAS

05  INPUT E1
10  INPUT R
25  I=E1/R
35  PRINT "THE EQUIVALENT CURRENT IS",I, " AMPERES"
40  END

READY
WEAVE RESIS.BAS        MERGE RESIS.BAS into the file in user core
                       (EQUIV.BAS).
READY
LIST


EQUIV.BAS      15:25          12-SEPT-73

05  INPUT E1                                   Line 05 is inserted.
10  INPUT R1, R2, R3                           Line 10 is replaced.
20  R= (R1*R2*R3) / ((R2*R3) + R1*(R2+R3))     Line 20 is inserted.
25  I=E1/R                                     Line 25 is retained.
30  PRINT "THE PARALLEL RESISTANCE = "; R      Line 30 is inserted.
35  PRINT "THE EQUIVALENT CURRENT IS",I, " AMPERES"
                                               Line 35 is retained.
40  END                                        Line 40 is replaced.

READY
```

9.4 TRANSFERRING FILES

Once a user has prepared a file in his portion of core memory, he will want to move the modified file to permanent storage such as, disk, DECtape, or papertape. In addition, the user may want to move files from one storage device to another, especially from disk to DECtape.

9.4.1 Transferring Files From the User's Core Storage

Upon creating, weaving and editing a file in core storage, a user wants to retain a copy of the file for future use. The SAVE and REPLACE commands are then used.

```
SAVE dev:filename.ext
```

The SAVE command puts the file currently in user core on the storage device dev:, under the name of filename.ext.

If dev: is omitted DSK: is assumed. If .ext is omitted .BAS is assumed. The filename.ext may be omitted, in which case the current filename.ext is used. The extension cannot be specified if the filename is omitted.

The SAVE command does not overwrite an existing file of the same name. REPLACE should be used if that result is desired.

```
SAVE                    Instruct BASIC to SAVE the present file on disk storage.
READY
```

```
REPLACE  dev:filename.ext
```

The REPLACE command deletes an existing file called filename.ext which is on the device dev: and inserts a new file from user core in its place, keeping the same name.

If the device is DSK: or DECtape, the old file must be present on the device or an error message will be issued.

The arguments dev:, filename, and .ext can be omitted with the same conditions described for the SAVE command.

Also, refer to the OLD command (Paragraph 9.1) and the WEAVE command (Paragraph 9.3.5), both of which transfer lines into core memory.

```
REPLACE
READY
```

9.4.2 Transferring Files From One Storage Device to Another

```
COPY dev1:filename1.ext >dev2:filename2.ext
```

The COPY command reads filename1.ext on dev1: and transfers a copy of it to dev2: where it is given the name filename2.ext.

If the device is omitted, DSK: is assumed. If the device is not a disk or DECtape, the filename and extension can be omitted. Note, when the filename is omitted, the extension must also be omitted. Should the device be a disk or DECtape, the filename must be specified, but the extension can be omitted, and then the extension .BAS is used.

The filename1.ext need not have line-numbers to be acceptable to COPY. The program currently in core is not disturbed by a COPY command.

```
COPY RESIS.BAS > DTA3:24RES.BAS      Make a copy of RESIS.BAS
                                     (which is on disk storage) on
READY                                DECtape 3, and call it 24RES.BAS.
```

9.4.3 Destroying Files

```
UNSAVE dev:filename.ext, dev:filename.ext, ...
```

The UNSAVE command deletes the named files from the indicated devices.

The arguments dev:, filename, and .ext can be omitted. If dev: is omitted, DSK: is assumed. If .ext is omitted, .BAS is assumed. If filename.ext is omitted, the current filename.ext is used.

In specifying more than one file to be UNSAVEd, the user must separate the filenames with commas.

```
UNSAVE RESIS.BAS

FILES UNSAVED:
RESIS

READY
```

9.5 COMPILING AND EXECUTING A BASIC PROGRAM IN CORE MEMORY

If the user's core file is a BASIC program and it has been created, edited, and saved, it is ready to be compiled and executed. Note: A BASIC file does not have to be a BASIC program. The Edit and Control Commands discussed in this chapter are also used in creating files containing data and files containing text. The user does not want to compile and execute a data file or a text file. The RUN, CHAIN and ↑C commands aid the user in processing his BASIC program.

```
RUN n
RUNNH n
```

The RUN commands compile the entire program residing in user core. The RUN command generates a heading upon running the program; while RUNNH deletes the heading. After compilation, the program is executed starting at statement number n. If n is omitted, execution starts at the very beginning of the program.

CHAIN

The CHAIN statement can be included in a BASIC program to cause one program to run another program. For further information refer to Paragraph 6.6.

```
↑C↑C
```

Two ↑C's stop a running program and return the user to the BASIC command mode. All files that were opened by the program are closed.[1]

[1] In monitors prior to 5.05 (and in 1040 monitors), two CTRL-C's (↑C↑C) may return the user to monitor mode, which is indicated when a period is typed. If this occurs, type REENTER to return to BASIC command mode.

9.6 ENTERING MONITOR MODE FROM BASIC

While in BASIC, the user may desire to use additional I/O devices, obtain system information or request a special service. In order to accomplish these and other similar tasks, the user must put his terminal in monitor mode.

9.6.1 What Is Monitor Mode?

Once BASIC has printed "READY, FOR HELP TYPE HELP", the user knows that he has successfully entered BASIC and he can now type BASIC commands. In the DECsystem-10 environment, there can be many people using a large variety of system programs (of which BASIC is one), running their own programs, or performing other functions. The monitor is the supervisory program which schedules and controls those operations requested by each user, so that the system can better serve all users.

To issue a request to the monitor (a monitor command) after entering BASIC, the user must leave BASIC and enter monitor mode. He may then use the appropriate monitor command(s).

When the user is finished and desires to reenter BASIC, he must type certain monitor command(s) to leave monitor mode and enter BASIC.

Refer to the DECsystem-10 Operating System Commands in the DECsystem-10 User's Handbook (DEC-10-NGZB-D) or to the DECsystem-10 Software Notebooks.

"Operating system commands" is another name for "monitor commands".

Table 9-1

Commands That Enter Monitor Mode From BASIC

Command	Result
MONITOR	Causes the user to leave BASIC and enter monitor mode. The process is complete when the monitor types a period, indicating that the user may type any monitor command. Caution -- All Monitor Commands which run a program in performing their function, destroy the contents of the user's core. This means that all work done in BASIC that has not been permanently stored somewhere else (e.g., papertape, DECtape, or disk) by using a BASIC command will be lost. Refer to Paragraph 9.4.
SYSTEM	Is almost identical to the MONITOR command in function. Unlike MONITOR, however, it does not allow the user to reenter BASIC by typing CONTINUE; only REENTER or START will succeed.

Table 9-2

Useful Monitor Commands

Command	Function
.ASSIGN dev:	Allocates an I/O device to the user's job for the duration of the job or until a DEASSIGN command is given. No operator intervention is required (preserves user core).
.DAYTIME	Types the date followed by the time of day (preserves user core).
.DEASSIGN dev:	Releases a device the user has previously assigned to his job (preserves user core).
.DIRECTORY	Lists the names of all user files currently on disk storage (destroys user core).
.DISMOUNT dev:	Releases a device the user has previously requested through a MOUNT command. This command requires operator intervention (destroys user core).
.R GRIPE	Records user comments for later review by the operations staff (destroys user core).
.MOUNT dev:	Requests an I/O device be allocated to the user's job. This command requires operator intervention (destroys user core).
.PLEASE	Allows the user uninterrupted communications with the operator (destroys user core).
.REWIND dev:	Rewinds a magnetic tape or a DECtape (destroys user core).
.SEND	Transmits a one-way message to a designated station (preserves user core).
.SYSTAT	Runs a program which prints status information about the system (destroys user core).
.TIME	Types out the total running time of the whole job (preserves user core).
.UNLOAD dev:	Rewinds and DEASSIGNs or DISMOUNTs a DECtape or magnetic tape (destroys user core).
dev: can be DTAn:, DSKa:, LPTn:, etc. n=numeric, a=alphanumeric	

9.6.2 Returning to BASIC From Monitor Mode

There are two different methods of returning to BASIC from monitor mode. The method the user employs depends upon whether he preserved or destroyed his core memory area.

9.6.2.1 User's Core Preserved - If the user has entered monitor mode and has not issued a monitor command which destroys the contents of his core memory, he may use one of the following commands to reenter BASIC.

Table 9-3

Commands That Reenter BASIC When Core is Preserved

Command	Function
.REENTER or .START	The user can exit from the monitor and reenter BASIC by typing either REENTER or START. If, while in the monitor, the user has issued a monitor command which destroys user core, neither the REENTER nor the START command causes the user to reenter BASIC. In this case he must type R BASIC.
.CONTINUE	The CONTINUE command serves exactly the same purpose as REENTER or START as far as BASIC is concerned. However, it will only be successful in reentering BASIC after a BASIC MONITOR command; it will not work after a SYSTEM command has been used.

```
READY
SYSTEM                  Enter Monitor Mode.

.ASSIGN DTA3:           Request DECtape 3 be allocated to the job.

DTA3 ASSIGNED           Monitor assigns DTA3:
.CONTINUE               Try to reenter to BASIC.

?CAN'T CONTINUE         Cannot type CONTINUE after a SYSTEM command.
.START                  Alternative request to return to BASIC.

READY                   O.K. BASIC responds.
```

9.6.2.2 User's Core Destroyed - When the user desires to return to BASIC after he has destroyed his core storage area he does so by typing R BASIC. BASIC responds with "READY, FOR HELP TYPE HELP" when it is ready to accept commands.

```
READY
MONITOR                    Enter Monitor Mode.
.DIRECTORY                 Request listing of user files.

HEADGS  BAS     2  <055>   23-APR-73     DSKC:    [27,23]
BASE2   BAS     3  <155>   15-JUL-73
BTPLSR  RNO     7  <055>   15-AUG-73

  TOTAL OF  12 BLOCKS IN   3 FILES ON DSKC:   [27,23]

.REENTER                   Try to get back to BASIC.
?NO START ADR              No starting address exists since .DIRECTORY destroyed
                           user core.
.R BASIC                   BASIC must be recalled.
READY, FOR HELP TYPE HELP
```

9.7 OBTAINING INFORMATION

Three commands, CATALOG, HELP and LENGTH, retrieve important information from the system concerning available I/O devices, commands and program size.

CATALOG device:

After the CATALOG device: command is entered, the system lists on the user's terminal the names and extensions of the user's files residing on the named device. When device: is omitted, DSK: is assumed.

device: can be

BAS:	Typing BAS: as the device outputs the library programs residing on the storage area BAS.
SYS:	Typing SYS: as the device lists the system programs stored on the system device SYS.
DSKn:	n (a number or letter) specifies a particular disk when more than one is available. If n is omitted, files on all disks are listed.
DTAn:	n (usually a number 0,1,2,...) identifies a particular DECtape when more than one is available. If the operating system command, ASSIGN, has been used to assign a DECtape or DECtapes, n may be omitted. The files on the DECtape with the lowest logical number will be listed. Should n be omitted and no DECtapes are assigned the user, logical unit 0 is assumed. Errors may result from omitting n in this situation. Refer to ASSIGN in Paragraph 9.2 for more guidance in using DECtapes.

```
READY
CATALOG DSKC:                  Type the names of all files on disk DSKC:

HEADGS.BAS
BASE2.BAS                      Files on DSKC: are listed.
POPUL.BAS

READY


READY
CATALOG DTA:                   Type the names of files on the DECtape with
                               the lowest logical unit number which is
                               assigned to the user.
OPER1 ACTION REQUESTED         In this case no DECtape was assigned to the
                               user so DTA0: was assumed, but this DECtape
                               was assigned to another user.


READY
MONITOR
.ASSIGN DTA1:                  Assign DTA1: to your job.

DTA1 ASSIGNED
REENTER

READY
CATALOG DTA:                   Type the names of files on DTA1:

FILA.REL
UPPER.CAS
SOS.OUT

READY
```

HELP

After the user types the HELP command BASIC types a list of BASIC commands and a brief explanation of each on the user's terminal.

```
READY
HELP                           Request BASIC to output a brief description
                               of its commands.


THIS IS THE HELP FILE FOR DECSYSTEM-10 BASIC VERSION 17B

THE FOLLOWING IS A SHORT (TWO PAGE) DESCRIPTION OF
SOME OF THE MOST COMMONLY USED COMMANDS.  FOR MORE
INFORMATION SEE THE BASIC MANUAL IN THE DECSYSTEM-10
SOFTWARE NOTEBOOKS.

BYE

        LOGS THE USER'S JOB OFF THE SYSTEM.
.
.
.

READY
```

LENGTH

The LENGTH command instructs the system to output the approximate length of the source program (stored in the user's core memory) expressed as the number of characters.

```
READY
LENGTH
295 CHARACTERS
READY
```

ADDITIONAL INFORMATION

Information about the system is available via the use of operating system commands. Refer to Paragraph 9.6.

9.8 SETTING THE INPUT MODE

The two input mode commands (TAPE and KEY) are useful only to users whose terminals are equipped with paper tape apparatus. One commonly used machine of this type is the LT33B (Figure C-1).

Table 9-4

Input Mode Commands

By specifying:	Input Mode
KEY	The system is set to accept input from the terminal keyboard.
TAPE	The user is aided in inputting information stored on paper tape.

When neither command is input, KEY is assumed. Refer to Appendix C for more information. TAPE and KEY apply only to input mode, and do not affect output to the terminal.

```
READY
TAPE

READY
```

9.9 LEAVING BASIC

When the user has finished all his work, he wants to use BYE or GOODBYE.

```
BYE
GOODBYE
```

BYE and GOODBYE exit the user from BASIC and log him off the DECsystem-10. Refer to Section 4.6.

CHAPTER 10
DATA FILE CAPABILITY

The data file capability allows a program to write information into and read information from data files that are on the disk.

Nine input/output channels are reserved for handling data files from a program. A data file must be assigned to a channel before it can be referenced in the program. At any given time, a program can have one and only one file on each channel and one and only one channel assigned to each file. Consequently, a maximum of nine files can be open simultaneously. However, because it is possible for a program to change or establish file/channel assignments while it is running, there is no limit to the number of data files that can be referenced in one program.

10.1 TYPES OF DATA FILES

There are two types of data files acceptable to BASIC: sequential access files and random access files.

10.1.1 Sequential Access Files

Sequential access files are those files that contain information that must be read or written sequentially, one item after another, from the beginning of the file. A sequential access file is either in write mode or read mode, but cannot be in both modes at the same time. When read mode is established, reading starts at the beginning of the file. When write mode is established, the file is erased and writing starts at the beginning of the file.

An important distinction to note about sequential access files is that they can be listed in readable form on the user's Teletype or the line printer. Sequential access files consist of lines that contain data items. A sequential access file is either a line-numbered file or a nonline-numbered file, depending upon whether or not its lines begin with line numbers. Line-numbered files are like BASIC programs in that they can be manipulated by any of the commands described in Chapter 9 (e.g., OLD, LIST, DELETE) except the RUN(NH) and CHAIN commands. Nonline-numbered files cannot be handled by any of the commands that expect a file to have line numbers; they can only be manipulated by the

COPY, QUEUE, and UNSAVE commands. They can be listed on the user's Teletype by means of the COPY command; for example:

```
            COPY TEST4 > TTY:
```

Sequential access files do not necessarily have to be created by a program; they can be created at the editing level in BASIC. Line-numbered files can be created or modified just as a BASIC program is created or modified. Nonline-numbered files can be created at the Teletype and then transferred to a storage device such as the disk by means of the COPY command. The following conventions must be observed when dealing with a sequential access file at the editing level:

a. In line-numbered files, each line number must be followed immediately by at least one space, a tab, or the letter D.

b. A line can contain any number of data items separated from one another by at least one space, a comma, or a tab. However, the line must not be longer than 142 characters (counting the line number and its following delimiter, but not the carriage return and line feed that terminate the line). It is not necessary to have a space, comma, or tab after the last data item on the line. Note that blanks and tabs are not ignored in a data file as they are in a program.

c. A data item is any numeric constant (refer to Section 1.3.3) or string constant (refer to Chapter 8). Numeric constants must not contain blanks or tabs. If a string is to contain a blank, comma, or tab, the user must enclose the string in quotes; otherwise it will be read as more than one data item by the statements that read data.

Section 10.4 contains an example of the use of a line-numbered data file created at the editing level. Section 10.5.1 contains an example of a program that creates both a line-numbered data file and a nonline-numbered data file and shows what these files look like when they are copied to the Teletype.

Because it requires execution time for a program to read and write line numbers in a data file, a nonline-numbered data file should be used in preference to a line-numbered data file unless the user specifically wishes to edit the data file with commands such as DELETE.

Another distinction between sequential access files is whether the file is a pure data file or a text file. A pure data file is used primarily for the storage of data. A text file contains data that is probably destined for output to the line printer, because it is a report, a financial statement, or the like. The user must follow slightly different procedures in his program depending on the type of file he wishes to handle. For example, a string that contains a blank must be enclosed in quotes when it is written into a pure data file, otherwise it will be seen as more than one string when data is read from the file. However, such a string should not be enclosed in quotes when it is written into a text file because text files are not normally read back into a program, and the superfluous quotes would spoil the appearance of the file when it is printed. The procedures to follow when handling each type of file are explained in Sections 10.5.1 and 10.7.

10.1.2 Random Access Files

Random access files are data files that are not necessarily read or written sequentially. The user can read items from or write items into a random access file without having the items follow one after the other. The items in a random access file are not recorded in a form suitable for listing, and therefore cannot be output to the user's Teletype or the line printer. Random access files cannot be handled by any of the BASIC commands other than COPY and UNSAVE. A random access file can be copied to the disk, DECtape, or magnetic tape, but not to any other device (Teletype, paper-tape punch, card punch, or line printer). Copying a random access file to a device other than disk, DECtape, or magnetic tape will cause errors to be introduced into the file. If the system program PIP is used to transfer a random access file to disk, DECtape, or magnetic tape, the file must be transferred in binary mode. Refer to the PIP manual for information on how to use PIP.

Random access files, unlike sequential access files, do not distinguish between read mode and write mode. The user can read or write any item in a random access file at any time by first setting a pointer to that item. A random access file contains either string data or numeric data, but not both. Each data item in a random access file takes up the same amount of storage space, called a record, on the disk. BASIC must know the record size for the random access file in order to correctly move the pointer for that file from one data item to another. The record size for a random access numeric file is set by BASIC because the storage space required for a number in such a file is always the same. The storage space required for a string, however, is dependent upon the number of characters in the string. Thus, for a random access string file the user must specify the number of characters in the longest string in the file so that BASIC can set the record size accordingly. This specification takes place when the file is assigned to a channel. Refer to the description of the FILES and FILE statements in Section 10.2. When creating a new random access string file, if the user specifies too few characters an error message is issued when a string too long to fit into a record is written. If too many characters are specified for a record, the strings will always fit, but space will be wasted on the disk. When he is dealing with an existing file, the user does not have to specify a record size. If he does specify a record size for an existing file, the record size must match that with which the file was written.

BASIC processes random access files more quickly than it processes sequential access files. Consequently, if the user wishes to read or write large amounts of data in sequential order, but does not require that the data be in listable form, he should consider using a random access file to take advantage of its speed. A random access file can easily be read or written in sequential order.

10.2 THE FILE AND FILES STATEMENTS

The FILE and FILES statements perform identical functions. They both assign a file to a channel and establish the type of the file (sequential, random access string, or random access numeric). The difference between FILE and FILES is that FILE is an executable statement while FILES is not. Before execution of the program begins, BASIC collects all of the FILES statements in the program, makes the channel assignments, and sets the file types as they were declared in the FILES statements. The FILES statements are not used again during that execution of the program. GO TO and GOSUB statements to FILES statements work just as they do to REM statements; i.e., execution will transfer to the first executable statement following the FILES statement. The FILE statement, on the other hand, assigns channels and establishes file types during program execution, thereby allowing the user to change file/channel assignments during the running of his program.

The FILE and FILES statements accept filename arguments of the form:

filenm.ext type

where filenm and .ext are the filename and extension of the file in the form described in Chapter 4. The filename must be specified, but the extension can be omitted. If the extension is omitted, .BAS is assumed. Type can be a percent sign (%); a dollar sign ($) optionally followed by one, two, or three digits; or omitted. If type is omitted, the file is assumed to be a sequential access file. If a percent sign is specified, the file is assumed to be a random access numeric file. A dollar sign optionally followed by a one- to three-digit number indicates a random access string file. The number following the dollar sign specifies the number of characters in the longest string that the file will contain. A maximum of 132 characters and a minimum of one character can be specified. If the number is omitted from the dollar sign type and the file does not presently exist, a default length of 34 characters is established. If the number is omitted from the dollar sign type and the file does exist, the length with which the file was previously written is established.

The FILES statement has the form:

FILES filenm.ext type, filenm.ext type,...filenm.ext type

where the arguments can be separated by a comma or a semicolon. Channels are assigned consecutively to the arguments of all the FILES statements in the program. If an argument is omitted, the channel for the missing argument is skipped. For example, if a program contains only these FILES statements:

```
10      FILES ,, A;,B
20      FILES C,D,
30      FILES E
```

file A will be assigned to channel 3, file B to channel 5, file C to channel 6, file D to channel 7, and file E to channel 9.

The FILE statement has the form:

FILE arg1, arg2, ... argn

where the arguments can be separated by a comma or a semicolon. At least one argument must be present in a FILE statement. Each argument that assigns a sequential access file to a channel is of the form:

```
   #N, string formula
or #N: string formula
```

Each argument that assigns a random access file to a channel is of the form:

```
   :N, string formula
or :N: string formula
```

N is a numeric formula having a value from 1 to 9 that specifies the channel; the value is truncated to an integer if necessary. The string formula is of the form:

filenm. ext type

Note that the channel specifier for a random access file is preceded by a colon (:) while the channel specifier for a sequential access file is preceded by a number sign (#). This is true of all data file statements and functions that include channel specifiers. Some data file statements and functions do not require the number sign or colon to be specified explicitly, but default to one or the other. See the description of the various statements and functions in the following sections for details. An attempt to reference a file with a channel specifier of the wrong type causes an error message.

The FILE statement does not permit the enclosing quotes to be omitted when its string formula argument is a constant. This is because a statement of the form FILE :1, B$ would cause an ambiguity. The B$ could be taken as a variable (B$) or as a random access string file named B.

Before the FILE statement assigns a file to a channel, it checks to see if a file already exists on that channel; if so, the old file is closed and removed from the channel before the new file is assigned. The type of the old file is immaterial; it is permissible, for example, to close an old sequential access file on a channel and then open a random access file on that channel. Any file open on a channel at the end of program execution or whenever BASIC is reentered is automatically closed and removed from that channel.

Examples of FILES and FILE statements are:

```
10      FILE #1, "ONEDAT": #4,"OUTDAT"
20      FILE #9: "CHKDAT.4", :4, B$+"%"
30      FILES FOUR.OUT$, MAIN.8;;; PROG$16
40      FILF #R*2, "RASFIL"
```

10.3 THE SCRATCH AND RESTORE STATEMENTS

The SCRATCH statement has the form:

SCRATCH arg1, arg2, ... argn

The RESTORE statement has the form:

RESTORE arg1, arg2, ... argn

where the arguments can be separated by a comma or a semicolon. An argument is of the form:

For sequential access files:
#N

For random access files:
:N

where N is a numeric formula having a value from 1 through 9 that specifies the channel. If necessary, the value is truncated to an integer. If neither a number sign nor a colon is present in front of the N, the number sign is assumed. At least one argument must be present in a SCRATCH or RESTORE statement.

Scratching a sequential access file erases it and sets it in write mode. Writing will start at the beginning of the file. Referencing a sequential access file with a statement that does input (READ, INPUT, or IF END, described in Sections 10.4 and 10.10) while it is in write mode results in a fatal error.

Scratching a random access file simply erases it and sets the pointer for the file to the first record in the file.

Restoring a sequential access file sets the file in read mode. Reading will start at the beginning of the file. Referencing a sequential access file with a statement that does output (WRITE or PRINT, described in Section 10.5) while it is in read mode results in a fatal error. When a sequential access file is opened by a FILES or FILE statement and the file exists at that time, it is automatically set in read mode; it is not necessary to restore it. It is only necessary to restore a sequential access file if it has been set in write mode and the user wishes to set it to read mode in the same program.

Restoring a random access file simply sets the pointer for the file to the first record in the file. When a random access file is opened on a channel by a FILE or FILES statement, its pointer is automatically set to point to the first record of the file.

Examples of the SCRATCH and RESTORE statements are:

```
10      SCRATCH #4, :2, #T-1, 1
20      SCRATCH #1, 2, 3, 4
80      RESTORE :2 $9, 1
90      RESTORE SQR (X), 2, 3, 7
```

10.4 THE READ AND INPUT STATEMENTS

The READ and INPUT statements read data items from files. The READ statement has the following forms:

> For sequential access files:
> READ #N, variable, variable, ... variable
>
> For random access files:
> READ :N, variable, variable, ...variable

The INPUT statement has the following forms:

> For sequential access files:
> INPUT #N, variable, variable, ... variable
>
> For random access files:
> INPUT :N, variable, variable, ...variable

N is a numeric formula having a value from 1 through 9 that specifies the channel. The value is truncated to an integer if necessary. At least one variable must be present in each READ or INPUT statement. The delimiter following N can be a comma or a colon. The variables are separated from one another by a comma or semicolon.

The variables in a READ or INPUT statement for a sequential access file can be string or numeric or a mixture of both. The variables in a READ or INPUT statement for a random access file can be string or numeric, but not both, because a given random access file cannot contain both string and numeric data items.

READ and INPUT statements for sequential access files differ from one another in the following way. The READ statement expects each line of data in the file to begin with a line number, which it then skips. That is, the line number is not treated as data. If a line number is not present, an error message is issued. The INPUT statement, on the other hand, does not expect a line number on each line of data. If one is present, it is read as data. It is illegal to use both INPUT and READ statements to read from the same sequential access file unless the file has been restored between the two types of statements. An attempt to mix READ and INPUT statements for sequential access files results in a fatal error message.

Examples of the READ and INPUT statements for sequential access files are:

```
10      READ #2, A(I), L, B$
30      READ #6, Z$
105     INPUT #4, B(K)
120     INPUT #7, W$, M
```

READ and INPUT statements for random access files are completely equivalent. They both begin reading at the item that the pointer for the file specifies, and continue reading sequentially until all of the variables have been filled. It is legal to use both READ and INPUT statements to input from the same random access file.

If the user attempts to read beyond the last item in either a sequential access or a random access file, a fatal error message is issued. In a random access file, it is possible to have items that have not been written but that are within the file (because some subsequent item has been written). If such an item is in a numeric file and is read, a value of zero is input. If such an item is in a string file, a string containing no characters is returned.

Examples of READ and INPUT statements for random access files are:

```
20      READ :2, A, B(I), C; F2
50      READ :4, F$, G$(8)
210     INPUT :1, Q(2)
240     INPUT :5: N1; N2; N3
```

The following example shows a sequential access file being created at the editing level and then read by a program.

```
NEW
NEW FILE NAME--TEST2

READY
10  "LANTHANIDE SERIES"
20  LA,CE,PR,ND,PM,SM,EU,GD,TB,DY,HO,ER
25  TM,YB,LU,57,71
SAVE
```

The user types in and then SAVEs the data file "TEST2".

```
READY
OLD
OLD FILE NAME--TABLE

READY
LISNH
1   DIM A$(15)
5   FILES TEST2
12  READ #1,B$
15  FOR X=1 TO 15
20  READ #1,A$(X)
25  NEXT X
30  READ #1,N1,N2
35  PRINT "THIS IS THE ";B$
40  PRINT
42  PRINT "ELEMENT", "ATOMIC NUMBER"
44  PRINT
45  FOR Y=1 TO 15
50  PRINT A$(Y),N1-1+ Y
55  NEXT Y
100 END
```

The old file "TABLE" is retrieved and listed.

(continued on next page)

```
READY
RUN

TABLE           13:31           15-JUL-70


THIS IS THE LANTHANIDE SERIES

ELEMENT         ATOMIC NUMBER

LA              57
CE              58
PR              59
ND              60
PM              61
SM              62
EU              63
GD              64
TB              65
DY              66
HO              67
ER              68
TM              69
YB              70
LU              71

TIME:  0.18 SECS.
READY
```

An example of reading from a random access file is given in Section 10.6.

10.5 THE WRITE AND PRINT STATEMENTS

The WRITE and PRINT statements write data items into files.

10.5.1 WRITE and PRINT Statements for Sequential Access Files

The WRITE and PRINT statements for sequential access files have the following forms:

WRITE #N, list of formulas and delimiters
PRINT #N, list of formulas and delimiters

where N is the channel specifier. The delimiter following N can be a comma or a colon; it can be omitted if the list is omitted. The formulas in the list can be string or numeric or both. The TAB function can be used. The delimiters can be commas, semicolons, or <PA> delimiters; they have the same meanings that they have in the PRINT statement for the Teletype (refer to Chapter 6).

WRITE and PRINT statements for sequential access files differ from one another in the following way. The WRITE statement begins each line of output with a line number followed by a tab. The first line in the file is numbered 1000 and subsequent line numbers are incremented by 10. The PRINT statement, on the other hand, does not begin lines with line numbers. It is illegal to use both WRITE and PRINT statements to write to the same sequential access file unless the file has been erased (by means of the SCRATCH command) between the two types of statement. An attempt to mix WRITE and PRINT statements results in a fatal error message.

Files created by WRITE statements are normally read by READ statements. Files created by PRINT statements are normally read by INPUT statements.

Examples of the WRITE and PRINT statements for sequential access files are:

```
50      WRITE #2, SQR(A)+EXP(G); Q(I)
75      PRINT #7, <PA> B(I),,C(I),,D(I)
110     WRITE #3
```

The normal mode of output for WRITE and PRINT statements for sequential access data files is noquote mode. In noquote mode, strings are not enclosed in quotes even if they contain characters that the READ and INPUT statements see as delimiters. Also, strings are concatenated if they are output with a semicolon separating them. Noquote mode is the mode used when writing a text file (refer to Section 10.1.1 for a description of text files and pure data files). Noquote is the default mode; a sequential access file is automatically set in noquote mode when it is assigned to a channel by a FILE or FILES statement. However, noquote mode is not suitable when writing pure data files because the integrity of the data is not maintained. In order to write a pure data file, the file must be set in quote mode. This can be done by the QUOTE or QUOTE ALL statement, both of which are described in Section 10.7. When a file is in quote mode, BASIC accepts WRITE and PRINT statements that are in the usual form, but it makes whatever small changes that are necessary to the formatting in order to preserve the integrity of the data items. Refer to Section 10.7 for details about the changes that are made.

An example of the actions performed by the WRITE and PRINT statements follows.

```
10      FILES A, B
20      SCRATCH #1,2
30      WRITE #1, 1; 2, TAB(70), 3
40      PRINT #2, "A"; 4
50      END

RUNNH

TIME:  0.02 SECS.

READY

COPY A > TTY:

01000 1 2
01010 3
```

(continued on next page)

```
READY
COPY B > TTY:

A 4

READY
```

10.5.2 WRITE and PRINT Statements for Random Access Files

The WRITE and PRINT statements for random access files have the forms:

```
WRITE :N, formula, formula, ... formula
PRINT :N, formula, formula, ... formula
```

where N is the channel specifier. The delimiter following the channel specifier can be a comma or a colon. At least one formula must be present in each statement. The formulas are separated from one another by a comma or semicolon. In a given statement, all of the formulas must be string or all of them must be numeric because a random access file is either string or numeric but not both.

WRITE and PRINT statements for random access files are exactly equivalent; they both begin writing into the record that the pointer for the file specifies, and continue writing sequentially until all of their arguments have been written. It is legal to use both WRITE and PRINT statements to write to the same random access file.

Examples of WRITE and PRINT statements for random access files are:

```
25      WRITE :2, N, L; M
35      PRINT :4: A$, B$+Q$(I)
```

An example of writing to a random access file is shown below in Section 10.6.

10.6 THE SET STATEMENT AND THE LOC AND LOF FUNCTIONS

The SET statement has the form:

```
SET arg1, arg2, ... argn
```

where the arguments can be separated by commas or semicolons. Each argument has the form:

```
     :N, numeric formula
or   :N: numeric formula
```

where N is the channel specifier. The colon preceding the channel specifier can be omitted because SET is only used for random access files; the colon is therefore redundant. Each SET statement must have at least one argument. When a SET statement is executed, the pointer for the file on the specified channel is moved so that it points to the item in the file that is specified by the numeric formula, which has been truncated to an integer. If the numeric formula after truncation is less than or equal to zero, an error message is issued. The items in the file are numbered sequentially; the first item in the file is 1, the second 2, and so forth. The next statement in the program that reads from or writes to the random access file will read or write the item to which the pointer was set, provided that the pointer has not been moved again by a subsequent SET statement or another statement.

Examples of SET statements are:

```
55      SET  :3, 100, :4, 150
85      SET  :1,1; :4,215
```

An example of a program using the SET statement follows.

```
10      FILES TEST4%
20      FOR T=1 TO 10
30      WRITE :1, T
40      NEXT T
50      FOR T=1 TO 10 BY 2
60      SET :1, T
70      READ :1, X
80      PRINT X
90      NEXT T
100     END

RUNNH

 1
 3
 5
 7
 9

TIME:  0.01 SECS.
```

Two functions, LOC and LOF, return information about random access files. LOC returns the number of the record to which the pointer for the file currently points, and LOF returns the number of the last record in the file.

The forms of LOC are:

LOC(N)
LOC(:N)

The forms of LOF are:

LOF(N)
LOF(:N)

where N is the channel specifier. An error message is issued if a random access file is not assigned to the specified channel when the function is executed.

An example of these functions is:

```
        10      IF LOC(2)<=LOF(2) THEN 30
        20      PRINT "FINISHED FILE ON CHANNEL 2"
```

10.7 THE QUOTE, QUOTE ALL, NOQUOTE, AND NOQUOTE ALL STATEMENTS

As was discussed in Section 10.5.1, the default mode for output to sequential access data files or to the TELETYPE is noquote mode. The QUOTE and QUOTE ALL statements allow the user to change the mode of the Teletype and sequential access files to quote mode. Quote mode changes the way that the data items are written into the files or onto the Teletype. In quote mode, strings are enclosed in double quotes by BASIC if they contain blanks, tabs, or commas; a leading blank is output immediately before strings and negative numbers; and a double quote character cannot be output by the user. If such an attempt is made to output a double quote character, an error message is issued. Also a data item cannot be longer than the maximum amount of space available on a new line. If an attempt is made to output a data item longer than this, a fatal error message results. In noquote mode, the data item would be split across two or more lines. These modifications to the normal formatting are sufficient to insure that the integrity of the data is maintained, as was discussed in Section 10.5.1.

The opposite of quote mode is noquote mode, which can be set by the NOQUOTE and NOQUOTE ALL statements. Noquote mode is the default mode for the Teletype and sequential access files. Whenever a sequential access file is assigned to a channel by a FILES or a FILE statement, it is automatically set in noquote mode. NOQUOTE and NOQUOTE ALL statements are only necessary if the user wishes to change a file from quote to noquote mode.

When creating a pure data file, in addition to setting the file in quote mode, it is good practice to separate the formulas in the WRITE or PRINT statements with semicolons to pack the data items close together. Although separating the formulas with commas is permissible, it will waste space on the disk.

The form of the QUOTE statement is:

    QUOTE arg1, arg2, ... argn

where each argument has the form:

```
            #N
        or  N
```

where N is the channel specifier. If an argument is omitted, the Teletype is specified; for example,

```
        30      QUOTE , 1, 4
```

refers to the Teletype and the files on channels 1 and 4.

Since QUOTE is assumed to have at least one argument, the statement

```
        50      QUOTE
```

specifies the Teletype.

The form of the QUOTE ALL statement is:

QUOTE ALL

QUOTE ALL refers to channels 1 through 9, but not to the Teletype.

When a channel is referenced in a QUOTE or QUOTE ALL statement and that channel has a sequential access file currently assigned to it, output to the file is done in quote mode. If a sequential access file is not presently assigned to the channel, nothing is done and no error message is returned.

The form of the NOQUOTE statement is the same as that of the QUOTE statement, except that the word NOQUOTE is substituted for the word QUOTE. Examples of NOQUOTE statements are:

```
        10      NOQUOTE #7,,2
        20      NOQUOTE
```

The first example specifies the files on channels 7 and 2 and the Teletype. The second example specifies the Teletype.

The form of the NOQUOTE ALL statement is:

NOQUOTE ALL

When a channel is referenced by a NOQUOTE or NOQUOTE ALL statement and that channel has a sequential access file currently assigned to it, output to the file will be written in noquote mode. If a sequential access file is not presently assigned to the channel, nothing is done and no error message is returned.

The use of the QUOTE ALL or NOQUOTE ALL statement is a convenient way to set all sequential access files currently assigned to channels into the appropriate mode, since the statements will not return error messages about or affect unassigned channels or the Teletype, and will not damage any of the random access files currently assigned to channels.

Quote or noquote mode can be set even if the file is in read mode because these modes have no effect on input. They will affect the output if the file is subsequently put into write mode.

If the mode is changed from quote to noquote or vice versa, the change takes effect immediately.

10.8 THE MARGIN AND MARGIN ALL STATEMENTS

Normally, the right output margin for the Teletype and sequential access files is 72 characters. Whenever a sequential access file is assigned to a channel by a FILES or a FILE statement, the file's output margin is automatically set to 72 characters. At the beginning of and also at the end of program execution, the Teletype output margin is set to 72 characters. There is no margin in a random access file.

The MARGIN and MARGIN ALL statements allow the user to set the right output margin for the Teletype or any sequential access file from 1 to 132 characters.[1] The form of the MARGIN statement is:

MARGIN arg1, arg2, ... argn

where each argument has the form:

#N, numeric formula

The arguments can be separated by commas or semicolons. N is the channel specifier. The numeric formula specifies the margin size; it is truncated to an integer. Either a comma or a colon can be used to separate the channel number from the margin size.

If only the margin size is present in the argument, that argument refers to the Teletype. For example:

```
35      MARGIN 75, #8:132
```

sets a margin of 75 characters for the Teletype and a margin of 132 characters for the file on channel 8.

The form of the MARGIN ALL statement is:

MARGIN ALL numeric formula

This statement sets the sequential access files on channels 1 through 9 to the margin specified by the numeric formula, the value of which is truncated to an integer before the margin is set. The Teletype is not affected by the MARGIN ALL statement. Examples of the MARGIN ALL statement are:

```
60      MARGIN ALL 132
65      MARGIN ALL N*ABS(K(I))
```

Neither the MARGIN nor MARGIN ALL statement has any effect on random access files or on channels that have no files assigned to them. Consequently, the MARGIN ALL statement is a convenient way to set a margin for all sequential access files currently assigned to channels.

[1] The monitor command SET TTY WIDTH must be used in addition to the BASIC MARGIN statement if the user wishes to set the output margin for the Teletype to any size greater than 72 characters. Refer to Section 6.7 for details.

The margins set by the MARGIN and MARGIN ALL statements apply only to output. The margin for input lines for both the Teletype and sequential access files is not affected by these statements; it is always 142 characters. An attempt to input a line longer than 142 characters results in an error message.

A margin set by a MARGIN or MARGIN ALL statement takes effect as soon as a new line of output is begun for the Teletype or the sequential access file.

Although the right margin can be set to any number between 1 and 132 characters, the margin for lines output by WRITE statements must be at least 7 characters to allow for the line number and its following tab. If the margin is less than 7 characters for a line-numbered file, an error message is issued by the first WRITE statement referencing the file.

10.9 THE PAGE, PAGE ALL, NOPAGE, AND NOPAGE ALL STATEMENTS

Normally, output to the Teletype or to sequential access files is not divided into pages; that is, it is in nopage mode. Whenever a sequential access file is assigned to a channel by a FILES or a FILE statement, it is automatically set in nopage mode. At the beginning and also at the end of program execution, the Teletype is set to nopage mode. The PAGE and PAGE ALL statements allow the user to set a page size of any positive number of lines for the Teletype and sequential access files. The NOPAGE and NOPAGE ALL statements allow the user to set the Teletype and sequential access files to nopage mode. Nopage and page modes are meaningless for random access files.

The form of the PAGE statement is:

PAGE arg1, arg2, ... argn

where each argument has the form:

#N, numeric formula

The arguments can be separated by commas or semicolons. N is the channel specifier. The numeric formula is truncated to an integer and used to specify the page size. Either a comma or a colon can be used to separate the channel number from the page size.

If only a page size is present in an argument, that argument refers to the Teletype; for example:

```
40      PAGE #1, 66; 50, #7:62
```

sets the files on channels 1 and 7 to page sizes of 66 and 62 lines respectively, and the Teletype to a page size of 50 lines.

The form of the PAGE ALL statement is:

PAGE ALL numeric formula

This statement sets the sequential access files on channels 1 through 9 to a page size specified by the numeric formula; however, the Teletype is not affected. The value of the numeric formula is truncated to an integer before the page size is set. An example of the PAGE ALL statement is:

```
90      PAGE ALL Q(2)*B
```

Neither the PAGE nor PAGE ALL statement has any effect on random access files or on channels that have no files assigned to them. Consequently, the PAGE ALL statement is a convenient way to set a page size for all of the sequential access files currently assigned to channels. If a PAGE or PAGE ALL statement specifies a page size of zero or less than zero, an error message is issued.

When a PAGE or PAGE ALL statement is executed for a sequential access file that is in write mode or for the Teletype, BASIC ends the current line of output (if necessary), outputs a leading form feed, and starts counting lines beginning with the next line output. Subsequently, whenever a new page becomes necessary, a form feed is output and the line count is set back to zero. Execution of a <PA> delimiter sets the line count to zero. PAGE and PAGE ALL statements can be executed for sequential access files in read mode; in this case, the leading form feed is not output. A page size remains in effect until another PAGE or PAGE ALL statement changes it, until a NOPAGE or NOPAGE ALL statement is executed for that file or the Teletype, or until the end of program execution. Setting the page size for the Teletype is further described in Chapter 6.

The form of the NOPAGE statement is:

NOPAGE arg1, arg2, ... argn

where each argument has the form:

```
   #N
or  N
```

where N is the channel specifier. If an argument is omitted, the Teletype is specified; for example:

```
10      NOPAGE #3,, 2
```

refers to the Teletype and the files on channels 2 and 3.

Since the NOPAGE statement is assumed to have at least one argument, the statement

```
70      NOPAGE
```

refers to the Teletype.

The form of the NOPAGE ALL statement is:

NOPAGE ALL

The NOPAGE ALL statement sets all of the sequential access files on channels 1 through 9 in nopage mode, but does not affect the Teletype.

Like the PAGE and PAGE ALL statements, NOPAGE and NOPAGE ALL statements have no effect on channels that have random access files or no files assigned to them. Consequently, the NOPAGE ALL statement is a convenient way to set all of the sequential access files currently assigned to channels into nopage mode.

10.10 THE IF END STATEMENT

The IF END statement allows the user to determine whether or not there is any data left in a file between the current position in the file and the end of the file.

The statement forms are:

For sequential access files:

IF END #N, $\begin{Bmatrix}\text{GO TO}\\ \text{THEN}\end{Bmatrix}$ line number

For random access files:

IF END :N, $\begin{Bmatrix}\text{GO TO}\\ \text{THEN}\end{Bmatrix}$ line number

where N is the channel specifier. The line number must refer to a line in the program and must follow the rules for line numbers discussed in Chapter 1. Either THEN or GO TO must be used in the statement. The comma preceding THEN or GO TO is optional.

The IF END statement will execute for a sequential access file only if the file is in read mode; an error message will be issued if the file is in write mode or if it does not exist. The IF END statement will always execute for a random access file that exists because such a file does not distinguish between read and write modes. For the purposes of the IF END statement, the end of a random access file is considered to be just beyond the final record in the file. The LOC and LOF functions described in Section 10.6 can also be used to determine whether or not there is any data between the current pointer position in a random access file and the end of the file.

If an IF END statement is executed for a sequential access file that is in read mode but that has not yet been referenced by a READ or INPUT statement, the IF END statement will assume that the file does not have line numbers. Thus, if an IF END statement is executed for a line-numbered file that has not been referenced by a READ statement, the IF END statement will treat line numbers as data items and will erroneously report that there is data in the file if only line numbers remain in the file. As soon as a READ or INPUT statement is executed for a file, the IF END statement correctly interprets the kind of file (line-numbered or nonline-numbered) and can distinguish between line numbers and data.

The following example shows how the IF END statement works for sequential access files.

```
10      FILES TEST
20      SCRATCH #1
30      FOR X=1 TO 5
40      READ A
50      WRITE #1, A
60      NEXT X
70      RESTORE #1
80      FOR I=1 TO 10
90      PRINT "I = "; I,
100     IF END #1 THEN 170
110     READ #1, B(I)
120     PRINT B(I)
130     NEXT I
140     PRINT "FAILED"
150     STOP
160     DATA -1,-2,-3,-4,-5,-6,-7,-8,-9,-10
170     END

RUNNH

I =  1           -1
I =  2           -2
I =  3           -3
I =  4           -4
I =  5           -5
I =  6

TIME:  0.10 SECS.
```

If the final record written into a random access file is record number 1804, for example, the IF END statement will cause a transfer when it is executed only if the pointer for that file has a value of 1805 or greater at that time.

CHAPTER 11
FORMATTED OUTPUT

The user who wishes to control the format of his output more than is permitted by the PRINT, PRINT#, and WRITE# statements described in Chapters 6 and 10 can use the statements described in this chapter. These statements are PRINT USING, PRINT USING#, and WRITE USING#. They all use a special formatting string, called an image, to format their output.

11.1 THE USING STATEMENTS

The PRINT USING statement allows formatting of string and numeric output to the Teletype. The forms of the PRINT USING statement are:

```
PRINT USING line number, list
PRINT USING string formula, list
```

The PRINT USING# and WRITE USING# statements allow formatting of output to data files. PRINT USING# formats output to data files without line numbers; WRITE USING# formats output to line-numbered data files. The forms of the PRINT USING# statement are:

```
PRINT USING #N, line number, list
PRINT USING #N, string formula, list
PRINT #N, USING line number, list
PRINT #N, USING string formula, list
```

The forms of the WRITE USING# statement are:

```
WRITE USING #N, line number, list
WRITE USING #N, string formula, list
WRITE #N, USING line number, list
WRITE #N, USING string formula, list
```

N is a numeric formula having a value from 1 through 9 that specifies the channel that the file is on. The comma following N can be omitted in the forms in which N precedes the word USING. The list has the form:

```
formula delimiter formula delimiter...formula
```

The formulas are either string or numeric and the delimiters are commas or semicolons. At least one formula must be present in the list.

The USING statements output each formula in their lists under the control of an image that specifies the format. The image is a string of characters that describe the form of the output (integer, decimal, string, etc.) and the placement of the output on the output line. If the USING statement contains a line number as its argument, the image is on the line specified by that line number. Such a line is called an image statement and has the form:

line number : string of characters

The string of characters in an image statement is not enclosed in quotes. For example:

```
10      PRINT USING 20, A
20      : THE ANSWER IS ####
```

Image statements cannot be terminated by the apostrophe remarks indicator because an apostrophe can be used as a format control character in an image.

If the USING statement contains a string formula as its argument, the image is the value of the string formula. If the string formula is a string constant, it must be enclosed in quotes. An example of the image in the USING statement is:

```
10      PRINT USING "THE ANSWER IS ####", A
```

When a USING statement is executed, BASIC begins a new line of output, and the first argument in the USING statement is output into the first specification in the image. If there are more arguments in the USING statement than specifications in the image, a new output line is begun and the specifications in the image are used again. USING statements always write complete lines. The current margin set for the Teletype or the data file referenced does not affect USING statements; however, an attempt to create a line longer than 132 characters results in an error message. Quote and noquote modes do not affect USING statements; USING statements ignore both modes.

The WRITE USING# statement performs the same functions as the PRINT USING# statement except that WRITE USING# places a line number and a tab at the beginning of each line. Neither the line number nor the tab are specified in the image. WRITE USING# statements must be used for files that have line numbers, and PRINT USING# statements must be used for files that do not have line numbers. If an attempt is made to use a WRITE# or WRITE USING# statement for a file that was previously written by PRINT# or PRINT USING# statements, an error message will be issued unless an intervening SCRATCH# statement erased the file. Similarly, an attempt to use PRINT# or PRINT USING# statements for a file that was previously referenced by WRITE# or WRITE USING# statements results in an error message unless an intervening SCRATCH# statement erased the file.

An example of PRINT USING# and WRITE USING# is shown below.

```
10      FILES TEST1, TEST2
20      SCRATCH #1, #2
30      A$ = "THE INDEX IS ##"
40      FOR T = 1 TO 3
50      PRINT USING #1, A$, T
60      WRITE USING #2, A$, T
70      NEXT T
80      END
RUNNH

TIME:  0.01 SECS.

READY
COPY TEST1 > TTY:
THE INDEX IS 1
THE INDEX IS 2
THE INDEX IS 3

READY
COPY TEST2 > TTY:
1000 THE INDEX IS 1
1010 THE INDEX IS 2
1020 THE INDEX IS 3

READY
```

11.2 IMAGE SPECIFICATIONS

An image is a string that contains format characters and printing characters. The format characters form specifications that describe how the values of the arguments of the USING statement will be arranged on an output line. More than one specification can be present in an image, but to avoid ambiguities when outputting numbers, the user should separate numeric specifications by string specifications, printing characters, or spaces. Note that spaces are printing characters and, therefore, as many spaces as are inserted between specifications will be inserted between the output items. That is, if two spaces separate a numeric specification from the preceding specification, two spaces will separate the numbers that are output according to these specifications. If numeric specifications are not separated from one another, ambiguities will generally exist and BASIC will make arbitrary decisions about the specifications. In general, it will accept as much of the specifications as it can, stopping when a character is seen that clearly delimits a specification because it considers that it has reached the end of the specification. String specifications need not be separated from one another because they are not ambiguous. Printing characters are output exactly as they appear in the image.

Image specifications can be divided into three major kinds:

a. Numeric image specifications

b. Edited numeric image specifications

c. String image specifications

11.2.1 Numeric Image Specifications

Numeric image specifications are used to describe the formats of integer and decimal numbers. Format characters within the image specification indicate the digits, sign, decimal point, and exponent of the number. Numbers in BASIC normally contain eight significant digits, and never contain more than nine significant digits. If a numeric image specification would cause a number to be output with more than nine significant digits, zeroes are substituted for all digits after the ninth. The format characters in all numeric image specifications must be contiguous.

The format characters used in numeric image specifications are:

> # (number sign)
> . (decimal point)
> ↑↑↑↑[1] (four up-arrows)

Number signs in the specification indicate the digits in the number and a minus sign if the number is negative. At least two number signs must be present at the beginning of the image specification; an isolated number sign is treated as a printing character. A number sign is written in the image specification for each digit in the number to be output plus one additional number sign to indicate a minus sign if the number to be output is negative. For example, to output a negative four-digit integer, at least five number signs should be written in the image specification; a non-negative number containing four digits requires only four number signs.

11.2.1.1 Integer Image Specifications - Numbers can be output as integers by means of an image specification containing only number signs. As stated above, an additional number sign must be included in the image specification for a minus sign if the number is negative. If the number is positive or zero, no sign is output; if the number is negative, a minus sign is output. If insufficient characters are present in the image specification, an ampersand (&) is placed in the first position of the output field and the field is widened to the right to accommodate the number. If the image specification width is larger than necessary to accommodate the number, the number is right-justified in the output field. The number to be output is truncated to an integer if it is not an integer. An example showing integer image specifications follows.

```
10      READ A,B,C,D,E
20      DATA 25.6, -14.7, 4, -9.1, -41876.3
30      PRINT USING "#### ###", A, B, C
40      A$ ="#####"
50      PRINT USING A$, D, E
60      END

RUNNH


  25  -14
   4
  -9
&-41876
```

[1]On some Teletypes, the circumflex (^) is used instead of the up-arrow (↑).

11.2.1.2 Decimal Image Specifications - Decimal image specifications must contain number signs, as in integer image specifications, and a single decimal point. Optionally, the user can include four up-arrows (↑↑↑↑) at the end of a decimal specification to indicate that the number is to be output with an explicit exponent. A number output under control of a decimal image specification always contains an explicit decimal point.

When four up-arrows are present in the image specification, an explicit exponent is output in the form E±nn. The sign of the exponent is always output, a plus sign for positive or zero exponents, a minus sign for negative exponents (e.g., E+01).

The decimal point in the image specification causes the decimal point to be fixed in the output field. Thus number signs that precede the decimal point in the image specification reserve space in the output field for the digits before the decimal point and a minus sign if the number is negative. At least one digit is always output before the decimal point, even if the digit is zero. The number signs that follow the decimal point in the image specification reserve space for the digits after the decimal point in the output field.

If the number is to be output with an explicit exponent, a position must be reserved for the sign of the number even if the number is positive (a space is output for the sign of a positive number). When the number with the exponent is output all of the positions before the decimal point in the output field are used and the exponent is adjusted accordingly. If the number is not to be output with an explicit exponent, and more spaces are reserved before the decimal point than are necessary, the number is right-justified in the output field and leading spaces are appended. If insufficient spaces are reserved before the decimal point, an ampersand (&) is placed in the first position of the output field and the field is widened to the right to accommodate the number.

Whether or not the number is output with an explicit exponent, as many digits are output following the decimal point as there are number signs following the decimal point in the image specification. The number is rounded or trailing zeros are added if necessary.

An example of the use of decimal image specifications follows.

```
10      READ A,B,C,D,E,F
20      :####.## ##. ##.####
30      PRINT USING 20, A,B,C,D,E,F
40      DATA 100.256, 3.6, 218.24318
50      DATA -4.6, 3, Ø.Ø1256
60      PRINT
70      PRINT USING 80, 100.2, 14
80      :###.#↑↑↑↑ ###.↑↑↑↑
90      END

RUNNH

 100.26  4. &218.2432
  -4.60  3.  Ø.Ø126

 10.ØE+Ø1  14.E+ØØ
```

11.2.2 Edited Numeric Image Specification

For those users who wish to output numbers in a form suitable for accounting reports, payrolls, and the like, additional format characters can be included in numeric image specifications to cause the numbers to be edited. The format characters used for edited numeric specifications are:

,	(comma)
-	(minus sign)
*	(asterisk)
$	(dollar sign)

Comma

One or more commas in the integer part of a numeric image specification causes the digits in the output number to be grouped into hundreds, thousands, etc., and separated by commas (e.g., 1,000,000). The commas, however, cannot be in the first two places in the specification. Only one comma need be present in the image specification for the number to be output with commas in the required places, but a pound sign or a comma must be present in the image specification to reserve space for each comma to be output. For example, to print the number 1,365,072, the image specification must contain one comma and at least eight pound signs and/or commas. It is useful, however, to position commas in the specification where they will appear when they are output, e.g., ##,###,###. A comma that is not part of a numeric image specification is treated as a printing character.

Example:

```
10      PRINT USING " ####,###",1E4,1E5,1E6
20      PRINT
30      PRINT USING 40, -141516.8
40      :###,#####.#
50      END

RUNNH

   10,000
  100,000
 &1,000,000

 -141,516.8
```

Trailing Minus Sign

A trailing minus sign in a numeric image specification causes the number to have its sign printed at its end, rather than at its beginning (e.g., 27-). A trailing minus sign in a number is often used in a report to indicate a debit. When a trailing minus sign is present in the image specification, a position need not be saved at the beginning of the image specification for the sign of the number, since the minus sign reserves a place for the sign. When the trailing minus sign is present in the image specification and the output number is positive or zero, the sign field on output is blank. A minus sign that does not end a numeric image specification is treated as a printing character.

Example:

```
10      PRINT USING "###-",10,-14,137.8
20      PRINT
30      PRINT USING 40, -141516.8, -14
40      :##,#####.#- ##.↑↑↑↑-
50      END

RUNNH

  10
  14-
 137

 141,516.8- 14.E+00-
```

<u>Leading Asterisk</u>

If a numeric image specification begins with two or more asterisks instead of number signs, the number is output with leading asterisks filling any unused positions in the output field. Leading asterisks are often used when printing checks or in any application that requires that the numbers be protected (i.e., so that no additional digits can be added).

Within an image specification, an asterisk can replace one or all of the number signs. In image specifications with leading asterisks, negative numbers can be output only if there is a trailing minus sign in the image specification. If a trailing minus sign is not present in the image specification and an attempt is made to output a negative number, an error message will be issued. Thus, an additional position need not be saved for a leading sign. Four up-arrows cannot be present in an image specification that contains leading asterisks. Thus, numbers with explicit exponents cannot be output with leading asterisks. An isolated asterisk in an image is treated as a printing character.

An example showing image specifications with leading asterisks follows.

```
10      A$="***.**"
20      READ X,Y,Z,W,U
25      DATA 13.56, 4.577, 3.1, 19.612, 100.50
30      PRINT USING A$, X,Y,Z,W,U
35      PRINT
40      PRINT USING "*****,*** ***#-", 1E6,-1E3
50      END

RUNNH

*13.56
**4.58
**3.10
*19.61
100.50

1,000,000 1000-
```

Floating Dollar Sign

If a numeric image specification begins with two or more dollar signs instead of number signs, the number is output with a floating dollar sign. That is, a dollar sign is always output in the position immediately preceding the first digit of the number, even if there are fewer digits in the number than there positions specified in the image specification. This capability is often used to protect checks so that there are never any spaces left between the dollar sign and the number.

The dollar sign can replace any or all of the number signs in the image specification (i.e., $$$$.$$ is exactly the same as $$##.##). An additional position at the beginning of the image specification must be indicated to save a place for the dollar sign in the output field.

When the floating dollar sign is used in the image specification, negative numbers can be output only if there is a trailing minus sign. If a trailing minus sign is not present in the image specification and an attempt is made to output a negative number, an error message is issued. Thus, a space need not be reserved for a leading sign in the output field. Four up-arrows cannot be present in a numeric image specification that contains dollar signs. Thus, numbers with explicit exponents cannot be output with floating dollar signs. An isolated dollar sign in an image is treated as a printing character.

An example showing floating dollar sign specifications follows. Note that the image in line 10 contains a decimal numeric image specification that is preceded by a dollar sign. This single dollar sign is treated as a printing character and, as shown in the example, is fixed in the output field.

```
10      :$$$$.$$  $####.##
20      READ A,B,C
25      DATA 100.43, 19.678, 0.97
30      PRINT USING 10, A,A,B,B,C,C
35      PRINT
40      PRINT USING "$$,,,,",1000
50      END

RUNNH

$100.43  $ 100.43
 $19.68  $  19.68
  $0.97  $   0.97

$1,000
```

11.2.3 String Image Specifications

The string image specifications allow the user to right-justify, left-justify, or center strings in the output field. In addition, the user can specify an image that causes the width of the output field to be extended if the string is larger than the image specifies. The format characters used for string output are:

'	(apostrophe)
C	(center)
L	(left-justify)
R	(right-justify)
E	(extend)

A string image specification contains one apostrophe (') and as many of the format characters C, L, R, or E as are necessary to output the string. The apostrophe is counted with the format characters when BASIC determines the length of the output field. The format characters cannot be mixed within an image specification. If the image specification contains only the apostrophe, only the first character in the string is output. The characters in a string image specification must be contiguous.

C Format Character

C format characters following the apostrophe in a string image specification cause the string to be centered in the output field. If a string cannot be exactly centered (e.g., a two-character string in a three-character field), it will be off-center one character position to the left. If the string to be output is longer than the image specification, the string is left-justified in the output field and the rightmost characters that overflow are truncated.

L Format Character

L format characters following the apostrophe in a string image specification cause the string to be left-justified in the output field. If the string to be output is longer than the image specification, the string is left-justified in the field and the rightmost characters that overflow are truncated.

R Format Character

R format characters following the apostrophe in a string image specification cause the string to be right-justified in the output field. If the string to be output is longer than the image specification, the string is left-justified in the field and the rightmost characters that overflow are truncated.

E Format Character

E format characters following the apostrophe in a string image specification cause the string to be left-justified in the output field. If the string to be output is longer than the image specification, the output field is widened (extended) to the right to accommodate all the characters in the string.

The following example shows the use of string image specifications.

```
100     : 'CCCC   'EFFE   'LLLL   'RRRR   '
110     INPUT A$
120     IF A$="STOP" GO TO 150
130     PRINT USING 100, A$,A$,A$,A$,A$
140     GO TO 110
150     END

RUNNH

 ?ABCD
ABCD   ABCD    ABCD     ABCD   A
?ABCDEF
ABCDE  ABCDEF  ABCDE   ABCDE   A
?A
  A     A       A             A   A
?STOP
```

Note that the last three fields in the second line printed are displaced one position because of the field extension necessary in the second field of the line.

11.2.4 Printing Characters

All characters in an image that are not format control characters are printing characters. Printing characters are output exactly as they appear in the image. Format control characters only appear as part of image specifications; if a character used as a format control character (e.g., $, E, *) does not appear as part of an image specification, it is treated as a printing character. If the USING statement does not use all of the specifications in an image, all of the printing characters except those following the unused specifications are printed. Similarly, if the USING statement uses the specifications in an image more than once, the printing characters in the image will be output as many times as the image is used. An example showing the use of printing characters in images follows.

```
10      : 'E TRUNCATED          'RRRR ROUNDED
20      : THE DATE IS:     'RRRRRRR1972
30      :A=### AND THE SQUARE ROOT OF A=##
40      PRINT USING 20,"1-JULY-"
50      PRINT
60      PRINT USING 10, "ALL NUMERIC OUTPUT FROM THIS PROGRAM IS"
70      PRINT
80      A=25
90      PRINT USING 30, A, SQR(A)
100     END

READY
RUNNH


 THE DATE IS:      1-JULY-1972

 ALL NUMERIC OUTPUT FROM THIS PROGRAM IS TRUNCATED

A= 25 AND THE SQUARE ROOT OF A= 5
```

APPENDIX A
SUMMARY OF BASIC STATEMENTS

A.1 ELEMENTARY BASIC STATEMENTS

The following subset of the BASIC command repertoire includes the most commonly used commands and is sufficient for solving most problems.

DATA [data list] READ [sequence of variables]	DATA statements are used to supply one or more numbers or alphanumeric strings to be accessed by READ statements. READ statements, in turn, assign the next available data, numeric or string as appropriate, in the DATA statement to the variables listed.
PRINT [sequence of formulas and format control characters]	Types the values of the specified formulas, which may be separated by format control characters. If two formulas are not separated by one or more format control characters, they are treated as though they were separated by a semicolon.
LET [variable] = [formula] or [variable] = [formula]	Assigns the value of the formula to the specified variable.
GO TO [line number]	Transfers control to the line number specified and continues execution from that point.
IF [formula] [relation] [formula] , $\begin{Bmatrix}\text{THEN}\\ \text{GO TO}\end{Bmatrix}$ [line number]	If the stated relationship is true, then transfers control to the line number specified; if not, continues in sequence. The comma preceding THEN and GO TO is optional.
FOR $\begin{bmatrix}\text{numeric}\\ \text{variable}\end{bmatrix}$ = $[\text{formula}_1]$ TO $[\text{formula}_2]$ $\begin{Bmatrix}\text{STEP}\\ \text{BY}\end{Bmatrix}$ $[\text{formula}_3]$ NEXT $\begin{bmatrix}\text{numeric}\\ \text{variable}\end{bmatrix}$ or NEXT $\begin{bmatrix}\text{numeric} & \text{numeric} & & \text{numeric}\\ \text{variable,} & \text{variable,} & \dots & \text{variable}\end{bmatrix}$	Used for looping repetitively through a series of steps. The FOR statement initializes the variable to the value of formula_1 and then performs the following steps until the NEXT statement is encountered. The NEXT statement increments the variable by the value of formula_3. (If omitted, the increment value is assumed to be +1.) The resultant value is then compared to the value of formula_2. If variable $<\text{formula}_2$, control is sent back to the step following the FOR statement and the sequence of steps is repeated; eventually, when variable $>\text{formula}_2$, control continues in sequence at the step following the current NEXT argument.

ON [x] , $\left\{\begin{matrix}\text{GO TO}\\ \text{THEN}\end{matrix}\right\}$ [line number$_1$,] [line number$_2$,] .... [line number$_n$]	If the integer portion of x = 1, transfers control to line number$_1$, if x = 2, to line number$_2$, etc. [x] may be a formula. The comma preceding GO TO and THEN is optional.
DIM [variable] (subscript) or DIMENSION [variable] (subscript)	Enables the user to enter a table or array with a subscript greater than 10 (i.e., more than 10 items).
END	Last statement to be executed in the program, and must be present.

A.2 ADVANCED BASIC STATEMENTS

GOSUB [line number] Subroutine $\left\{\begin{matrix}\text{[line number]} & . \\ & . \\ & . \\ & . \\ & . \\ & \text{RETURN}\end{matrix}\right.$	Simplifies the execution of a subroutine at several different points in the program by providing an automatic RETURN from the subroutine to the next sequential statement following the appropriate GOSUB (the GOSUB which sent control to the subroutine).
INPUT [variable(s)]	Causes typeout of a ? to the user and waits for user to respond by typing the value(s) of the variable(s).
STOP	Equivalent to GO TO [line number of END statement].
REM	Permits typing of remarks within the program. The insertion of short comments following any BASIC statement (except an image statement) is accomplished by preceding such comments with an apostrophe (').
RESTORE	Sets pointer back to beginning of string of DATA values.
CHANGE $\begin{bmatrix}\text{string formula}\\ \text{or}\\ \text{numeric vector}\end{bmatrix}$ TO $\begin{bmatrix}\text{numeric vector}\\ \text{or}\\ \text{string variable}\end{bmatrix}$	Changes a string formula to a numeric vector, or changes a numeric vector to a string variable.
CHAIN [string formula] or CHAIN [string formula] , [numeric formula]	Stops execution of the current program and begins execution of the new program at the beginning or at the specified line.
MARGIN [numeric formula]	Sets the Teletype to the specified output margin.
PAGE [numeric formula]	Output to the Teletype is divided into pages of the specified length.
NOPAGE	Output to the Teletype is not divided into pages.
QUOTE	The Teletype is set to quote mode (see Chapter 10).
NOQUOTE	The Teletype is set to noquote mode (see Chapter 10).

PRINT USING $\begin{bmatrix}\text{line number}\\ \text{or}\\ \text{string formula}\end{bmatrix}$, [sequence of formulas]	Types the values of the formulas in the format determined by the image specified by the line number or string formula.

A.3 MATRIX INSTRUCTIONS

NOTE

The word "vector" may be substituted for the word "matrix" in the following explanations.

MAT READ a, b, c	Read the three matrices, their dimensions having been previously specified.
MAT c = ZER	Fill out c with zeros.
MAT c = CON	Fill out c with ones.
MAT c = IDN	Set up c as an identity matrix.
MAT PRINT a, b, c	Print the three matrices.
MAT INPUT v	Input a vector.
MAT b = a	Set matrix b = matrix a.
MAT c = a + b	Add the two matrices, a and b.
MAT c = a - b	Subtract matrix b from matrix a.
MAT c = a * b	Multiply matrix a by matrix b.
MAT c = TRN(a)	Transpose matrix a.
MAT c = (k) * a	Multiply matrix a by the number k. (k, which must be in parentheses, may also be given by a numeric formula.)
MAT c = INV(a)	Invert matrix a.

(Refer to Section A.5 for the special matrix functions NUM and DET.)

A.4 DATA FILE STATEMENTS

FILE [sequence of [channel specifier] [filename arguments]]	Assigns files to channels during program execution.
FILES [sequence of filename arguments]	Assigns files to channels before program execution begins.
SCRATCH [sequence of channel specifiers]	Erases a sequential access file and puts it in write mode; or erases a random access file and sets the record pointer to the beginning of the file.
RESTORE [sequence of channel specifiers]	Puts a sequential access file in read mode or sets the record pointer for a random access file to the beginning of the file.

WRITE [channel specifier] [sequence of formulas]	Causes data to be output to a file on the specified channel. Used for sequential access files with line numbers, or for random access files.
READ [channel specifier] [sequence of variables]	Causes data to be input from a file on the specified channel. Used for sequential access files with line numbers or for random access files.
PRINT [channel specifier] [sequence of formulas]	Causes data to be output to a file on the specified channel. Used for sequential access files without line numbers or for random access files.
INPUT [channel specifier] [sequence of variables]	Causes data to be input from a file on the specified channel. Used for sequential access files without line numbers or for random access files.
IF END [channel specifier] , $\left\{\begin{matrix}\text{THEN}\\\text{GO TO}\end{matrix}\right\}$ [line number]	Determines whether or not there is data in a file between the current position and the end of the file. The comma preceding THEN or GO TO is optional.
MARGIN [sequence of [channel specifier] [numeric formula]]	Sets the specified output margins for the sequential access files on the specified channels.
MARGIN ALL [numeric formula]	Sets the specified output margin for the sequential access files on channels 1 through 9.
PAGE [sequence of [channel specifier] [numeric formula]]	Sets the specified output page sizes for the sequential access files on the specified channels.
PAGE ALL [numeric formula]	Sets the specified output page size for the sequential access files on channels 1 through 9.
NOPAGE [sequence of channel specifiers]	Output to the sequential access files on the specified channels is not divided into pages.
NOPAGE ALL	Output to the sequential access files on channels 1 through 9 is not divided into pages.
QUOTE [sequence of channel specifiers]	Puts the sequential access files on the specified channels into quote mode (see Chapter 10).
QUOTE ALL	Puts the sequential access files on channels 1 through 9 into quote mode (see Chapter 10).
NOQUOTE [sequence of channel specifiers]	Puts the sequential access files on the specified channels into noquote mode (see Chapter 10).
NOQUOTE ALL	Puts the sequential access files on channels 1 through 9 into noquote mode (see Chapter 10).
SET [sequence of [channel specifier] [numeric formula]]	Moves the record pointers for random access files.
PRINT [channel specifier] , USING $\left[\begin{matrix}\text{line number}\\\text{or}\\\text{string formula}\end{matrix}\right]$, [sequence of formulas] or PRINT USING [channel specifier] , $\left[\begin{matrix}\text{line number}\\\text{string formula}\end{matrix}\right]$, [sequence of formulas]	Causes data to be output to a sequential access file without line numbers on the specified channel. The data is output in the format determined by the image specified by the line number or string formula. In the first form, the comma following the channel specifier can be omitted.

WRITE [channel specifier] , USING [line number or string formula] , [sequence of formulas] or WRITE USING [channel specifier] , [line number or string formula] , [sequence of formulas]	Causes data to be output to a line-numbered sequential access file on the specified channel. The data is output in the format determined by the image specified by the line number or string formula. In the first form, the comma following the channel specifier can be omitted.

A.5 FUNCTIONS

In addition to the common arithmetic operators of addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (↑ or **), BASIC includes the following elementary numeric functions:

SIN (numeric formula)	COT	(numeric formula)	LOG, or LN, or LOGE	(numeric formula)
COS (numeric formula)	ATN	(numeric formula)	ABS	(numeric formula)
TAN (numeric formula)	EXP	(numeric formula)	SQR or SQRT	(numeric formula)
			CLOG or LOG10	(numeric formula)

Some advanced numeric functions include the following:

INT (numeric formula)	Finds the greatest integer not greater than its argument.
RND	Generates random numbers between 0 and 1. The same set of random numbers can be generated repeatedly for purposes of program testing and debugging. The statement RANDOMIZE can be used to cause the generation of new sets of random numbers.
SGN (numeric formula)	Assigns a value of 1 if its argument is positive, 0 if its argument is 0, or -1 if its argument is negative.
TIM	Returns the elapsed execution time, in seconds, since the program started execution. The time does not include compile and load time except when programs are chained. In such a case, the compile and load times of the programs after the first are included in the time returned.

Two functions used with matrix computations are as follows:

NUM	Equals number of components following an INPUT.
DET	Equals the determinant of a matrix after inversion.

Two functions for use with random access files are:

LOC (channel specifier)	Returns the number of the current record in the file on the specified channel.
LOF (channel specifier)	Returns the number of the last record in the file on the specified channel.

Functions for manipulating strings are:

ASC (one character or a 2- or 3-letter code)	Returns the decimal ASCII code for its argument. The two- or three-letter codes are listed in Table 8-1.
CHR$ (numeric formula)	The opposite function to ASC. The argument is truncated to an integer that is interpreted as an ASCII decimal number; a one-character string is returned.
INSTR (numeric formula, string formula, string formula) or INSTR (string formula, string formula)	Searches for the second string within the first string argument. In the first form, the search starts at the character position specified by the numeric formula, truncated to an integer. In the second form, the search starts at the beginning of the string. Returns zero if the substring not found; returns the position of the first character in the substring if it is found.
LEFT$ (string formula, numeric formula)	Returns a substring of the string formula, starting from the left. The substring contains the number of characters specified by the numeric formula truncated to an integer.
LEN (string formula)	Returns the number of characters in its argument.
MID$ (string formula, numeric formula, numeric formula) or MID$ (string formula, numeric formula)	Returns a substring of the string formula, starting at the character position specified by the first numeric formula truncated to an integer. In the first form, the substring contains the number of characters specified by the second numeric formula truncated to an integer. In the second form, the substring continues to the end of the string.
RIGHT$ (string formula, numeric formula)	Returns a substring of the string formula, starting from the right, containing the number of characters specified by the numeric formula truncated to an integer.
SPACE$ (numeric formula)	Returns a string of the number of spaces specified by the numeric formula truncated to an integer.
STR$ (numeric formula)	Returns a string representation of its argument.
VAL (string formula)	The opposite function to STR$. Returns the number that the string argument represents.

The user can also define his own functions by use of the DEFine statement. For example,

```
[line number]      DEF      FNC(x) = SIN (x) + TAN(x) - 10
```

where x is a dummy variable. (Define the user function FNC as the formula SIN(x) + TAN(x) - 10.)

NOTE

DEFine statements may be extended onto more than one line; all other statements are restricted to a single line (refer to Section 5.1.5).

APPENDIX B
BASIC DIAGNOSTIC MESSAGES

Most messages typed out by BASIC are self-explanatory. BASIC diagnostic messages are divided into three categories and listed in the three tables below:

a. Command errors in Table B-1

b. Compilation errors in Table B-2

c. Execution errors in Table B-3

Table B-1
Command Error Messages

Message	Explanation
?CANNOT INPUT FROM THIS DEVICE	An attempt has been made to input to a device that can only do output, or vice versa.
?CANNOT OUTPUT TO THIS DEVICE	
?CANNOT OUTPUT filenm.ext	A COPY, SAVE, or REPLACE command could not enter a file to output it. The actual name of the file is typed, not filenm.ext.
?CATALOG DEVICE MUST BE DISK OR DECtape	A device other than disk or DECtape was specified in a CATALOG command.
?COMMAND ERROR (YOU MAY NOT OVER-WRITE LINES OR CHANGE THEIR ORDER)	The given RESEQUENCE command would have changed the order of lines in the file. The command is ignored.
?COMMAND ERROR (LINE NUMBERS MAY NOT EXCEED 99999)	The given RESEQUENCE command is not executed for that reason.
?DELETE COMMAND MUST SPECIFY WHICH LINES TO DELETE	A DELETE command has no arguments.
?DUPLICATE FILENAME, REPLACE OR RENAME	User tried to SAVE a file that already exists.
?DUPLICATE SWITCH IN QUEUE ARGUMENT	Two switches of the same type have been specified for one QUEUE argument.
?FILE dev:filenm.ext COULD NOT BE UNSAVED	
?FILE dev:filenm.ext NOT FOUND	A file that was requested did not exist.

Table B-1 (Cont)
Command Error Messages

Message	Explanation
?LINE TOO LONG	A line of input is greater than 142 characters, not counting the terminating carriage return, line feed.
?LOGOUT FAILED -- TRY AGAIN	A BYE or GOODBYE command could not transfer control to the LOGOUT system program.
?MISSING LINE NUMBER FOLLOWING LINE nn*	During a WEAVE or OLD command, a line without a line number was found in the file. The line is thrown away.
?NO SUCH DEVICE, device	The device is not available.
?PAGE LIMIT >9999 OR < 1 IN QUEUE ARGUMENT	A LIMIT switch for a QUEUE argument was out of bounds.
?QUOTA EXCEEDED OR BLOCK NO. TOO LARGE ON OUTPUT DEVICE	Normally, this indicates that all of the space allowed on the output device has been used; no more can be output to this device unless some of the user's files are deleted from it. This can also mean that the block numer is too large for the output device.
?THIS COMMAND IS NOT IMPLEMENTED FOR THIS MONITOR	
?UNDEFINED LINE NUMBER mm IN LINE nn	
?UNSAVE DEVICE MUST BE DISK OR DECTAPE, FILE dev:filenm.ext	A device other than disk or DECtape was specified in an UNSAVE command.
?WHAT?	Catchall command error.
?>63 OR < 1 COPIES REQUESTED IN QUEUE ARGUMENT	A COPIES switch for a QUEUE argument was out of bounds.

*If the current program was called by a CHAIN statement, the name of the current program is appended to the error message.

Table B-2
Compilation Error Messages

Message	Explanation
?BAD DATA INTO LINE n	Input data is not in correct form.
?DATA IS NOT IN CORRECT FORM	Incorrect number or string data in a DATA statement.
?END IS NOT LAST IN nn	
?EOF IN LINE nn	An attempt was made to read data from a file after all data had been read.
?FAILURE ON ENTRY IN LINE nn	Channel is not available for SCRATCH command.
?FNEND BEFORE DEF IN LINE nn	FNEND occurs, but not in a function DEF.
?FNEND BEFORE NEXT IN LINE nn	A FOR occurred in a DEF, but its NEXT did not.
?FOR WITHOUT NEXT IN LINE nn	
?GOSUB WITHIN DEF IN LINE nn	A GOSUB statement is within a multiple line DEF.
?FUNCTION DEFINED TWICE IN LINE nn	
?ILLEGAL ARGUMENT FOR ASC FUNCTION IN LINE nn	
?ILLEGAL CHARACTER IN LINE nn	A meaningless character; e.g., DIM# (1).
?ILLEGAL CONSTANT IN LINE nn	
?ILLEGAL FORMAT IN LINE nn	Catchall for other syntax errors.
?ILLEGAL FORMAT WHERE THE WORDS THEN OR GO TO WERE EXPECTED IN LINE nn	
?ILLEGAL FORMULA IN LINE nn	Syntax error in arithmetic formula.
?ILLEGAL INSTRUCTION IN LINE nn	The first three non-blank, non-tab characters of the statement do not match the first three characters of any legal statement.
?ILLEGAL LINE REFERENCE IN LINE nn	BASIC syntax required an integer, but user typed something else; e.g., GO TO A.
?ILLEGAL LINE REFERENCE mm IN LINE nn	In line nn, line mm was referred to illegally because: a. Line mm is a REM b. The first character in line mm is an apostrophe ('). c. One of the lines nn or mm is inside a function; the other is not inside that function.
?ILLEGAL RELATION IN LINE nn	Incorrect IF relation.
?ILLEGAL VARIABLE IN LINE nn	

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all compilation error messages. For example, NO DATA IN TEST.BAK.

Table B-2 (Cont)
Compilation Error Messages

Message	Explanation
?INCORRECT NUMBER OF ARGUMENTS IN LINE nn	A function used with the wrong number of arguments.
?INITIAL PART OF STATEMENT NEITHER MATCHES A STATEMENT KEYWORD NOR HAS A FORM LEGAL FOR AN IMPLIED LET--CHECK FOR MISSPELLING IN LINE nn	
?MIXED STRINGS AND NUMBERS IN LINE nn	Line nn illegally contains a string variable or literal because: a. No element of this statement may be a string. b. All elements must be strings but some were not.
?NESTED DEF IN LINE nn	DEF within multiline DEF.
?NEXT WITHOUT FOR IN LINE nn	
?nn IS NOT AN IMAGE IN LINE nn	The specified line was referenced by a USING statement as an image, but it is not an image statement.
?NO CHARACTERS IN IMAGE IN LINE nn	
?NO DATA	Program contains READ but not DATA.
?NO END INSTRUCTION	
?NO FNEND FOR DEF FNx	The multiline DEF for FNx (the actual function name, not FNx is typed) has no FNEND.
?OUT OF ROOM	Cannot get more core to make room for: a. More compilation space. b. Maximum space for all the vectors and arrays c. Space to store another string during execution.
?RETURN WITHIN DEF IN LINE nn	A RETURN statement is within a multiple line DEF.
?SPECIFIED LINE IS NOT AN IMAGE IN LINE nn	
?STRING RECORD LENGTH >132 OR<1 IN LINE nn	The length of a record in a string random access file was specified as greater than 132 or less than 1.
?STRING VECTOR IS 2-DIM ARRAY	The user managed to do this error despite many other checks.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all compilation error messages. For example, NO DATA IN TEST.BAK.

Table B-2 (Cont)
Compilation Error Messages

Message	Explanation
?SYSTEM ERROR	An I/O error, or the UUO mechanism drops a bit, or something similar to those errors.
?TOO MANY FILES IN LINE nn	A maximum of nine files can be open at one time in a program.
?UNDEFINED FUNCTION -- FNx	The actual function name, not FNx, is typed.
?UNDEFINED LINE NUMBER mm IN LINE nn	In line nn, mm is used as a line number. Line number mm does not exist.
?USE VECTOR, NOT ARRAY IN LINE nn	A variable previously defined as a two-dimensional array is now used in MAT input or CHANGE.
?VARIABLE DIMENSIONED TWICE IN LINE nn	
?VECTOR CANNOT BE ARRAY IN LINE nn	A variable previously used in a MAT INPUT or CHANGE statement is now defined as a 2-dimensional array in a DIM statement.
?Character$_1$ WAS SEEN WHERE character$_2$ WAS EXPECTED IN LINE nn	An erroneous character was used in place of the correct character. Character$_1$ and character$_2$ are replaced by the appropriate characters or a phrase describing the characters when the message is issued.
NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all compilation error messages. For example, NO DATA IN TEST.BAK.	

Table B-3
Execution Error Messages

Message	Explanation
%ABSOLUTE VALUE RAISED TO POWER IN LINE nn	
?ATTEMPT TO OUTPUT A NEGATIVE NUMBER TO A $ OR * FIELD IN LINE nn	A USING statement attempted to output a negative number to a floating dollar sign or leading asterisk field that did not end in a minus sign.
?ATTEMPT TO OUTPUT A NUMBER TO A STRING FIELD OR A STRING TO A NUMERIC FIELD IN LINE nn	A USING statement attempted to output a number to a string field or a string to a numeric field.
?ATTEMPT TO READ# OR INPUT# FROM A FILE WHICH DOES NOT EXIST IN LINE nn	An attempt was made to read from a file that does not exist on the user's disk area.
?ATTEMPT TO READ# OR INPUT# FROM A FILE WHICH IS IN WRITE# OR PRINT# MODE IN LINE nn	An attempt was made to read from a sequential access file that is not in read mode.
NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.	

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
?ATTEMPT TO WRITE A LINE NUMBER >99,999 IN LINE nn	
?ATTEMPT TO WRITE# OR PRINT# TO A FILE WHICH HAS NOT BEEN SCRATCH#ED IN LINE nn	An attempt was made to write to a sequential access file that is not in write mode.
?ATTEMPT TO WRITE# OR PRINT# TO A FILE WHICH IS IN READ# OR INPUT# MODE IN LINE nn	An attempt was made to write to a sequential access file that is not in write mode.
?CHR$ ARGUMENT OUT OF BOUNDS IN LINE nn	The argument to the CHR$ function was less than zero or greater than 127.
?DATA FILE LINE TOO LONG IN LINE nn	An attempt has been made to read from a data file a line which is greater than 142 characters long.
?DIMENSION ERROR IN LINE nn	
%DIVISION BY ZERO IN LINE nn	†
?EXPONENT REQUESTED FOR * OR $ FIELD IN LINE nn	
?FILE IS NOT RANDOM ACCESS IN LINE nn	
?FILE NEVER ESTABLISHED -- REFERENCED IN LINE nn	
?FILE NOT FOUND BY RESTORE COMMAND IN LINE nn	
?FILE filenm.ext ON MORE THAN ONE CHANNEL IN LINE nn	The user tried to establish the same file on more than one channel. The actual filename and extension are typed, not filenm.ext.
?FILE NOT IN CORRECT FORM IN LINE nn	A data error has been detected in a string random access file.
?FILE RECORD LENGTH OR TYPE DOES NOT MATCH IN LINE nn	An existing random access file does match the type or record length specified for it in a FILE statement.
?IF END ASKED FOR UNREADABLE FILE IN LINE nn	

†An OVERFLOW error message means that an attempt has been made to create a number larger in magnitude than the largest number representable in the computer (approximately 1.7E + 38); when this occurs, the largest representable number is returned (with the correct sign) and execution continues. An UNDERFLOW error message means that an attempt has been made to create a nonzero number smaller in magnitude than the smallest representable positive number (approximately 1.4E -39); in this case, zero is returned and execution continues. Division by zero is considered overflow; the largest representable positive number is returned.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
?ILLEGAL CHARACTER IN STRING IN LINE nn	An attempt has been made to write onto a data file a string containing an embedded line terminator or quote.
?ILLEGAL CHARACTER SEEN IN LINE nn	An attempt has been made to create an illegal character in a CHANGE statement.
?ILLEGAL FILENAME IN LINE nn	The string argument is not in the correct form. If the argument is variable, check that it has been defined.
?ILLEGAL LINE REFERENCE IN RUN (NH) OR CHAIN	The line at which execution is to begin is inside a multiline DEF.
?IMPOSSIBLE VECTOR LENGTH IN LINE nn	In a CHANGE (to string) statement, the zero element of the number vector was negative or exceeded its maximum dimension.
?INPUT DATA NOT IN CORRECT FORM -- RETYPE LINE	
?INSTR ARGUMENT OUT OF BOUNDS IN LINE nn	
?LEFT$ ARGUMENT OUT OF BOUNDS IN LINE nn	
?LINE NUMBER OUT OF BOUNDS IN LINE nn	The line number argument is less than zero or greater than 99,999. The RUN (NH) commands return a ?WHAT? message in this situation.
%LOG OF NEGATIVE NUMBER IN LINE nn	
%LOG OF ZERO IN LINE nn	
%MAGNITUDE OF SIN OR COS ARG TOO LARGE TO BE SIGNIFICANT IN LINE nn	When the argument for COS or SIN is too large to be significant, this message is issued and an answer of 0 returned.
?MARGIN OUT OF BOUNDS IN LINE nn	A MARGIN or MARGIN ALL statement specified a margin greater than 132 characters or less than 1 character.
?MARGIN TOO SMALL IN LINE nn	A WRITE# statement referenced a file that has a margin of fewer than seven characters.
?MID$ ARGUMENT OUT OF BOUNDS IN LINE nn	
?MIXED RANDOM & SEQUENTIAL ACCESS IN LINE nn	A random access statement or function referenced a sequential access file, or vice versa.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN LINE 20 IN TEST.BAK.

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
?MIXED READ#/INPUT# IN LINE nn	An attempt was made to reference a file with both a READ# and an INPUT# statement without an intervening RESTORE# statement.
?MIXED WRITE#/PRINT# IN LINE nn	An attempt was made to reference a file with both a WRITE# and a PRINT# statement without an intervening SCRATCH# statement.
?NEGATIVE STRING LENGTH IN LINE nn	In a MID$, LEFT$, or RIGHT$ function, a negative number of characters was specified for a substring.
?NO FIELDS IN IMAGE IN LINE nn	An image contains neither string nor numeric fields.
?NO ROOM FOR STRING IN LINE nn	In a CHANGE A$ TO A, the number of characters in A$ exceeds the maximum size of A. A DIM statement appropriately increasing the size of A will cover this.
?NO SUCH LINE IN RUN (NH) OR CHAIN	The specified line does not exist in the program.
?NOT ENOUGH -- ADD MORE	
?ON EVALUATED OUT OF RANGE IN LINE nn	The value of the ON index was <1 or > the number of branches.
?OUT OF DATA IN LINE nn	
?OUTPUT ITEM TOO LONG FOR LINE IN LINE nn	In quote mode, an attempt was made to write a string or number that is too long to fit on one line.
?OUTPUT LINE 132 CHARACTERS IN LINE nn	A line of output created by a USING statement is greater than 132 characters.
?OUTPUT STRING LENGTH RECORD LENGTH IN LINE nn	An attempt has been made to output to a random access file a string which is too long to fit in one record.
%OVERFLOW IN LINE nn	†
%OVERFLOW IN EXP IN LINE nn	An exponent greater than 88.028 has been specified for the EXP function. An answer of the largest representable number is returned and execution continues.†

† An OVERFLOW error message means that an attempt has been made to create a number larger in magnitude than the largest number representable in the computer (approximately 1.7E + 38); when this occurs, the largest representable number is returned (with the correct sign) and execution continues. An UNDERFLOW error message means that an attempt has been made to create a nonzero number smaller in magnitude than the smallest representable positive number (approximately 1.4E - 39); in this case, zero is returned and execution continues. Division by zero is considered overflow; the largest representable positive number is returned.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
?PAGE LENGTH OUT OF BOUNDS IN LINE nn	A PAGE or PAGE ALL statement specified a page length of less than one line.
?QUOTA EXCEEDED OR BLOCK NO. TOO LARGE ON OUTPUT DEVICE	Normally, this indicates that all of the space allowed on the output device has been used; no more can be output to this device unless some of the user's files are deleted from it. It may also mean that the block number is too large for the output device.
?RETURN BEFORE GOSUB IN LINE nn	
?RIGHT$ ARGUMENT OUT OF BOUNDS IN LINE nn	
?SET ARGUMENT OUT OF BOUNDS IN LINE nn	The user attempted to set the value of the pointer to zero or to a negative number.
% SINGULAR MATRIX INVERTED IN LINE nn	
?SPACE$ ARGUMENT OUT OF BOUNDS IN LINE nn	The SPACE$ function was requested to return a string that was less than or equal to zero or greater than 132 characters.
%SQRT OF NEGATIVE NUMBER IN LINE nn	
?STRING FORMULA 132 CHARACTERS IN LINE nn	A string formula contains more than 132 characters.
?STRING RECORD LENGTH 132 OR 1 IN LINE nn	The record length for a string random access file was specified as less than one or greater than 132 characters.
?SUBROUTINE OR FUNCTION CALLS ITSELF IN LINE nn	FNA is defined in terms of FNB which is defined in terms of FNA, or a similar situation with FUNCTIONS or GOSUBS.
%TAN OF PI/2 OR COTAN OF ZERO IN LINE nn	
?TOO MANY ELEMENTS -- RETYPE LINE	

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
%UNDERFLOW IN EXP IN LINE nn	An exponent less than -88.028 has been specified for the EXP function. An answer of zero is returned and the execution continues. †
% UNDERFLOW IN LINE nn	†
?VAL ARGUMENT NOT IN CORRECT FORM IN LINE nn	The string argument to the VAL function does not represent a legal number.
%ZERO TO A NEGATIVE POWER IN LINE nn	

† An OVERFLOW error message means that an attempt has been made to create a number larger in magnitude than the largest number representable in the computer (approximately 1.7E + 38); when this occurs, the largest representable number is returned (with the correct sign) and execution continues. An UNDERFLOW error message means that an attempt has been made to create a nonzero number smaller in magnitude than the smallest representable positive number (approximately 1.4E - 39); in this case, zero is returned and execution continues. Division by zero is considered overflow; the largest representable positive number is returned.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.

APPENDIX C
TAPE AND KEY COMMANDS

The TAPE and KEY commands are designed for user Teletypes with attached paper-tape readers; for example, the LT33B shown in Figure C-1.

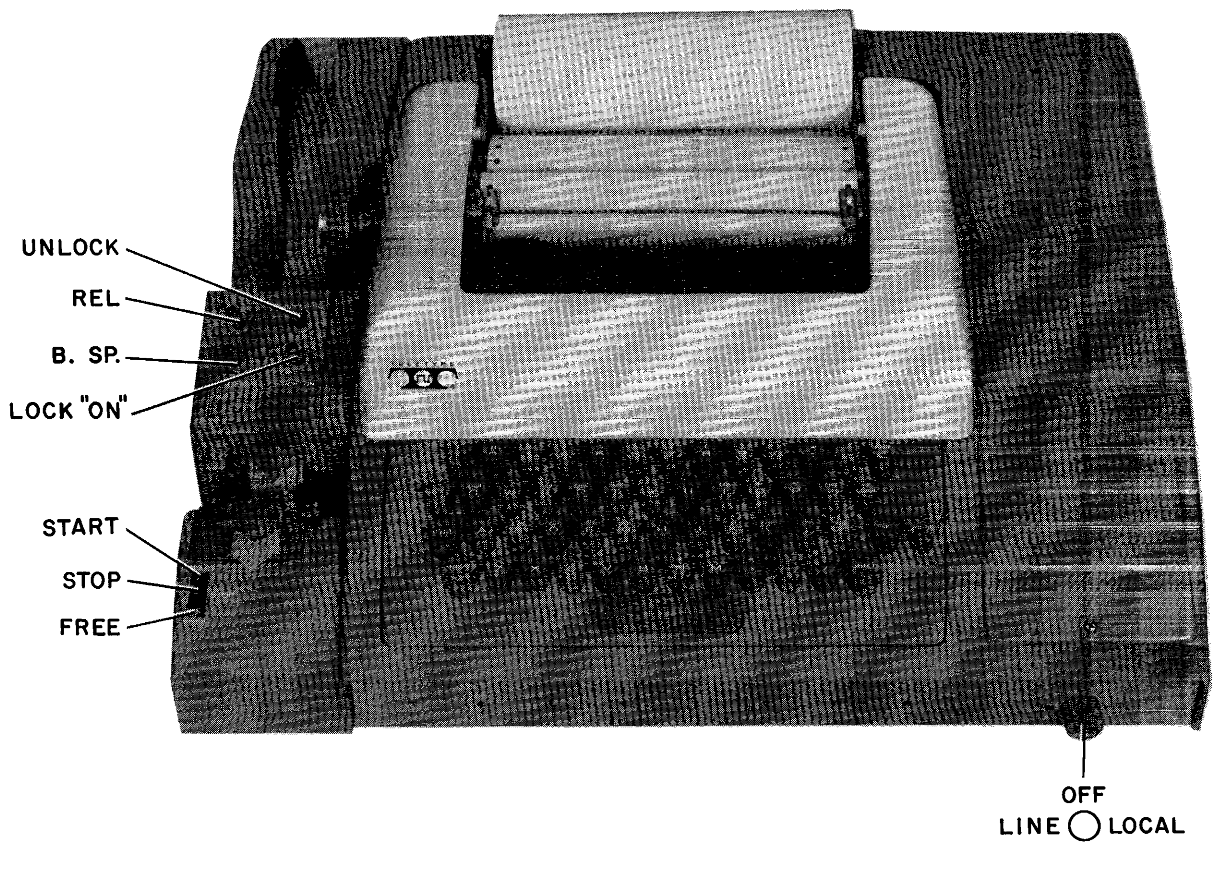


Figure C-1 LT33B Teletype

C.1 KEY AND TAPE MODES

KEY mode is produced by typing the KEY command to BASIC. In this mode, the user types input to BASIC on the keyboard in the normal manner. KEY mode is also the default mode.

TAPE mode is produced by typing the TAPE command to BASIC. The user initiates this mode whenever he wants to input from the paper tape reader while the Teletype is in LINE mode.

C.2 PREPARING AN INPUT TAPE IN LOCAL MODE

The following procedure should be followed for preparing an input tape.

Step	Procedure
1	Turn the Teletype control to LOCAL (see Figure C-1).
2	Feed blank tape into the punch.
3	Depress the LOCK "ON" control.
4	Generate the leader tape by doing the following: a. Depress the SPACE bar several times. b. Depress the RETURN key once. c. Depress the LINE FEED key once.
5	Type on the keyboard the commands and statements to be punched on the tape. a. At the end of each line, type both the RETURN and LINE FEED keys. b. If an incorrect character is typed, do the following: (1) Depress the BACKSPACE control (2) Depress the RUBOUT key. (3) Type the correct character. c. A TAB is received correctly when typed, even though the Teletype typewheel moves only one position to the right when TAB is typed. d. Any normal input to BASIC can be punched on the tape. A typical example is as follows: NEW TEST4 5 PRINT "THIS IS A TEST" 10 END LIS RUNNH
6	Generate a trailer tape by doing the following: a. Depress the SPACE bar several times. b. Depress the RETURN key once. c. Depress the LINE FEED key once.

Step	Procedure
7	Depress the UNLOCK control.
8	Remove the tape from the punch.
9	Depress the CTRL and T keys simultaneously.

C.3 SAVING AN EXISTING PROGRAM ON TAPE

The following procedure should be performed to save an existing program on tape.

Step	Procedure
1	Turn the Teletype control to LINE.
2	Depress the LOCK "ON" control.
3	Generate a leader tape by doing the following: a. Depress the SPACE bar several times. b. Depress the RETURN key once.
4	Turn the Teletype control to LOCAL.
5	Depress the UNLOCK control.
6	Depress the CTRL and T keys simultaneously.
7	Turn the Teletype control to LINE.
8	Type the LISTNH command, but do not depress the RETURN key.
9	Depress the LOCK "ON" control.
10	Depress the RETURN key.
11	Wait until the program has been listed and the READY message has been typed. NOTE The tape will contain not only your program but also an extra line at the end with the READY message on it. This is not important. Since READY is not a legal command, it will simply produce a WHAT? error message when the tape is input to BASIC, and then it will be ignored.
12	Generate a trailer tape by doing the following: a. Depress the SPACE bar several times. b. Depress the RETURN key once.
13	Turn the Teletype control to LOCAL.
14	Depress the UNLOCK control.
15	Remove the tape from the punch.
16	Depress the CTRL and T keys simultaneously.
17	Turn the Teletype control to LINE; now you are back in BASIC.

Step	Procedure
18	To stop the tape output while the program is being listed, do the following: a. Depress the UNLOCK control. b. Depress the CTRL and T keys simultaneously. c. Twice depress the CTRL and C keys simultaneously.

C.4 INPUTTING TO BASIC FROM THE READER

The following procedure should be followed for inputting to BASIC from the reader.

Step	Procedure
1	Turn the Teletype control to LINE.
2	With the reader control on STOP (see Figure C-1), position the tape on the sprocket wheel and close the tape retainer cover.
3	Type the command TAPE to BASIC.
4	Depress the RETURN key.
5	When BASIC answers READY, set the reader control to START.
6	When the tape has been read in, set the reader control to STOP.
7	Type KEY.
8	Depress the RETURN key.
9	Depress the LINE FEED key. (The Key mode is now restored.)
10	To stop the tape input while it is in progress, do the following: a. Switch the reader control to STOP. b. Twice depress the CTRL and C keys simultaneously. c. Type KEY. d. Depress RETURN. e. Depress LINE FEED.

NOTE

Do not type on the keyboard without first stopping the tape.

C.5 LISTING AN INPUT TAPE

An input tape is listed in the following manner:

Step	Procedure
1	Turn the Teletype control to LOCAL. (In LOCAL mode the tape is not inputted to the computer.)

Step	Procedure
2	Set the reader to STOP.
3	Put the tape in the reader.
4	Set the reader to START. (The contents of the tape is then printed on the console.)

INDEX