

Mark II  
Time-Sharing Service

Reference Manual



**BASIC**  
**Language**

DEC 68

**GENERAL**  **ELECTRIC**  
INFORMATION SERVICE DEPARTMENT

# BASIC Language

November 1967  
Reprinted 1-68, 7-68  
Revised 12-68

INFORMATION SYSTEMS

**GENERAL**  **ELECTRIC**

# Preface

---

This manual, which supersedes the Preliminary Reference Manual of the same title and number, describes the version of the BASIC language used with the Mark II Time-Sharing Service. Mark II includes the following extensions and additions to the versatile BASIC language:

- Data files
- String processing, which permits manipulation of alphanumeric data
- Chaining, which permits a program to stop and begin execution of another program without direct intervention
- Liberal definitions of variables in a function statement
- Capability for formatting output
- Ability to initialize all variables, lists, and tables to zero

Mark II BASIC also incorporates instructions such as a compound version of LET; ON, which provides a powerful conditional branch; and RANDOMIZE, for initializing the random number generator.

Another manual, Mark II Time-Sharing Service Command System: Reference Manual (711223) explains all of the system commands that are a part of the Mark II Time-Sharing Service.

The development of the BASIC language was supported by the National Science Foundation under the terms of a grant to Dartmouth College. Under this grant, Dartmouth College developed, under the direction of Professors John G. Kemeny and Thomas E. Kurtz, the BASIC language compiler. Since that development, BASIC has been offered as part of the Time-Sharing Service of General Electric's Information Service Department.

# Contents

---

	Page
1. WHAT IS A PROGRAM? .....	1
2. A BASIC PRIMER	
An Example .....	2
Formulas .....	5
Numbers .....	7
Variables .....	7
Loops .....	7
Lists and Tables .....	9
Errors and Debugging .....	11
Summary of Elementary Basic Statements .....	15
LET .....	15
READ and DATA .....	15
PRINT .....	16
GØ TØ .....	16
IF-THEN or IF-GØ TØ .....	16
FØR and NEXT .....	17
DIM .....	17
END .....	17
3. ADVANCED BASIC	
More About Print .....	18
Functions .....	20
RANDØMIZE .....	29
DEF .....	29
GØSUB and RETURN .....	30
ØN .....	31
INPUT .....	32
CHAIN .....	32
Other Useful Statements .....	34
STØP .....	34
REM .....	34
RESTØRE .....	35
TRACE ØN and TRACE ØFF .....	35
Matrices .....	36
Alphanumeric Data and String Manipulation .....	41
DIM Statement .....	41
LET Statement .....	41
IF-THEN Statement .....	42
CHANGE Statement .....	42
DATA, INPUT, and MAT INPUT Statements .....	44
READ and PRINT Statements .....	44
Data Files .....	44
Initial File Preparation .....	45
File Reference .....	45
File Designator .....	46
File Modes .....	46
File Read .....	46
Reading Internal Data .....	48
File Write .....	49
Reading with INPUT Statement .....	51
Writing with PRINT Statement .....	52

CONTENTS (Cont'd)

	Page
3. ADVANCED BASIC (Cont'd)	
MAT READ Statement . . . . .	53
MAT WRITE Statement . . . . .	54
RESTORE Statement . . . . .	55
SCRATCH Statement . . . . .	55
FILE Statement . . . . .	56
DELIMIT Statement . . . . .	57
APPEND Statement . . . . .	59
MARGIN Statement . . . . .	59
IF END Statement . . . . .	60
IF MORE Statement . . . . .	61
BACKSPACE Statement . . . . .	63

APPENDIXES

A. ERROR MESSAGES . . . . .	65
B. LIMITATIONS ON BASIC . . . . .	70

# 1. What Is A Program?

---

A program is a set of directions that is used to tell a computer how to provide an answer to some problem. It usually starts with the given data, contains a set of instructions to be performed or carried out in a certain order, and ends up with a set of answers.

Any program must meet two requirements before it can be carried out. The first is that it must be presented in a language that is understood by the computer. If the program is a set of instructions for solving a system of linear equations and the computer is an English-speaking person, the program will be presented in some combination of mathematical notation and English. If the computer is a French-speaking person, the program must be in his language; and if the computer is a high-speed digital computer, the program must be presented in a language which the computer understands.

The second requirement for all programs is that they must be completely and precisely stated. This requirement is crucial when dealing with a digital computer, which has no ability to infer what you mean--it does what you tell it to do, not what you meant to tell it.

We are, of course, talking about programs which provide numerical answers to numerical problems. It is easy for a programmer to present a program in the English language, but such a program poses great difficulties for the computer because English is rich with ambiguities and redundancies, those qualities which make computing impossible. Instead, you present your program in a language which resembles ordinary mathematical notation, which has a simple vocabulary and grammar, and which permits a complete and precise specification of your program. The language you will use is BASIC (Beginner's All-purpose Symbolic Instruction Code) which is, at the same time, precise, simple, and easy to understand.

A first introduction to writing a BASIC program is given in Chapter 2. This chapter includes all that you will need to know to write a wide variety of useful and interesting programs. Chapter 3 deals with more advanced computer techniques, and the Appendixes contain a variety of reference materials.

## 2. A Basic Primer

---

### AN EXAMPLE

The following example is a complete BASIC program for solving a system of two simultaneous linear equations in two variables:

$$\begin{aligned}ax + by &= c \\dx + ey &= f\end{aligned}$$

and then solving two different systems, each differing from this system only in the constants  $c$  and  $f$ .

You should be able to solve this system, if  $ae - bd$  is not equal to 0, to find that:

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

If  $ae - bd = 0$ , there is either no solution or there are infinitely many, but there is no unique solution. If you are rusty on solving such systems, take our word for it that this is correct. For now, we want you to understand the BASIC program for solving this system.

Study this example carefully – in most cases the purpose of each line in the program is self-evident – and then read the commentary and explanation.

```
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 GOTO 30
65 PRINT 'NO UNIQUE SOLUTION'
70 DATA 1, 2, 4
80 DATA 2, -7, 5
85 DATA 1, 3, 4, -7
90 END
```

A first observation is that each line of the program begins with a number. These numbers are called line numbers and serve to identify the lines, each of which is called a statement. Thus, a program is made up of statements, most of which are instructions to the computer. Line numbers also serve to specify the order in which the statements are to be performed by the computer. This means that you may type your program in any order. Before the program is run, the computer sorts out and edits the program, putting the statements into the order specified by their line numbers. (This editing process facilitates the correcting and changing of programs, as we shall explain later.)

A second observation is that each statement starts, after its line number, with an English word. This word denotes the type of the statement. There are several types of statements in BASIC, nine of which are discussed in this chapter. Seven of these nine appear in the sample program of this section.

A third observation, not at all obvious from the program, is that spaces have no significance in BASIC, except in messages which are to be printed out, as in line number 65 on the preceding page. Thus, spaces may be used at will to make a program more readable. Statement 10 could have been typed as 10READA,B,D,E, and statement 15 as 15LETG=A\*E-B\*D.

With this preface, let us go through the example, step by step. The first statement, 10, is a READ statement. It must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing your program, it will cause the variables listed after the READ to be given values according to the next available numbers in the DATA statements. In the 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, we have exhausted the available data in statement 70, but there is more in statement 80, and we pick up from it the number 2 to be assigned to E.

We next go to statement 15, which is a LET statement, and first encounter a formula to be evaluated. (The asterisk is used to denote multiplication.) In this statement we direct the computer to compute the value of  $AE - BD$ , and to call the result G. In general, a LET statement directs the computer to set a variable equal to the value of the formula on the right side of the equals sign. We know that 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. If the computer discovers a "yes" answer to the question, it is directed to go to line 65, where it prints "NO UNIQUE SOLUTION." From this point, it would go to the next statement. But lines 70, 80, and 85 give it no instructions, since DATA statements are not "executed," and it then goes to line 90 which tells it to "END" the program.

If the answer to the question "Is G equal to zero?" is "no," as it is in this example, the computer goes on to the next statement, in this case 30. (Thus, an IF-THEN tells the computer where to go if the IF condition is met, but to go on to the next statement if it is not met.) The computer is now directed to read the next two entries from the DATA statements, -7 and 5, (both are in statement 80) and to assign them to C and F respectively. The computer is now ready to solve the system

$$x + 2y = -7 \qquad 4x + 2y = 5$$

In statements 37 and 42, we direct the computer to compute the value of X and Y according to the formulas provided. Note that we must use parentheses to indicate that  $CE - BF$  is divided by G; without parentheses, only BF would be divided by G and the computer would let  $X = CE - BF/G$ .

The computer is told to print the two values computed, that of X and that of Y, in line 55. Having done this, it moves on to line 60 where it is directed back to line 30. If there are additional numbers in the DATA statements, as there are here in 85, 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, the computer is now ready to solve the system

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

As before, it finds the solution in 37 and 42 and prints the values of X and Y in 55, and then is directed in 60 to go back 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 to print out the solutions. It is directed back again to 30, but there are no more pairs of numbers available for C and F in the DATA statements. The computer then informs you that it is out of data, printing on the paper in your teletypewriter `OUT OF DATA IN 30`, and stops.

For a moment, let us look at the importance of the various statements. For example, what would have happened if we had omitted line number 55? The answer is simple: the computer would have solved the three systems and told us when it was out of data. However, since it was not asked to tell us (PRINT) its answers, it would not do it, and the solutions would be the computer's secret. What would have happened if we had left out line 20? In this problem just solved, nothing would have happened. But, if G were equal to zero, we would have set the computer the impossible task of dividing by zero in 37 and 42, and it would tell us so emphatically, printing `DIVISION BY ZERO IN 37` and `DIVISION BY ZERO IN 42`. Had we left out statement 60, the computer would have solved the first system, printed out the values of X and Y, and then gone on to line 65 where it would be directed to print `NO UNIQUE SOLUTION`. It would do this and then stop.

One very natural question arises from the seemingly arbitrary numbering of the statements: why this selection of line numbers? The answer is that the particular choice of line numbers is arbitrary, as long as the statements are numbered in the order which we want the machine to follow in executing the program. We could have numbered the statements 1, 2, 3, ..., 13, although we do not recommend this numbering. We would normally number the statements 10, 20, 30, ..., 130. We put the numbers such a distance apart so that we can later insert additional statements if we find that we have forgotten them in writing the program originally. Thus, if we find that we have left out two statements between those numbered 40 and 50, we can give them any two numbers between 40 and 50 - say 44 and 46; and in the editing and sorting process, the computer will put them in their proper place.

Another question arises from the seemingly arbitrary placing of the elements of data in the DATA statements: why place them as they have been in the sample program? Here again, the choice is arbitrary and 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 put

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

or we could have written, perhaps 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 exactly as it appears on the teletypewriter:

```
10 READ A,B,D,E
15 LET G = A*B - B*D
20 IF G = 0 THEN 65
30 READ C,F
37 LET X = (C*B - 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
99 END
```

```

RUN
LINEAR      15:07      12/06/68
  4          -5.5
0.666667    0.166667
-3.66667    3.83333
OUT OF DATA IN 30

```

After typing the program, we type RUN followed by a carriage return. Up to this point the computer stores the program and does nothing with it. It is this command which directs the computer to execute your program.

Note that the computer, before printing out the answers, printed the name which we gave to the problem (LINEAR) and the time and date of the computation.

The message OUT OF DATA IN 30 here may be ignored. But sometimes it would indicate an error in the program. For more detail see the paragraph "READ and DATA."

## FORMULAS

The computer can do a great many things – it can add, subtract, multiply, divide, extract square roots, raise a number to a power, and find the sine of a number (on an angle measured in radians), etc. We shall now learn how to tell the computer to do these things in the order that we want them done.

The computer computes by evaluating formulas which are supplied in a program. These formulas are similar to those used in standard mathematical calculation, except that all BASIC formulas must be written on a single line. Five arithmetic operations can be used to write a formula. These are listed in the following table:

<u>Symbol</u>	<u>Example</u>	<u>Meaning</u>
+	A + B	Addition (add B to A)
-	A - B	Subtraction (subtract B from A)
*	A * B	Multiplication (multiply B by A)
/	A / B	Division (divide A by B)
↑	X ↑ B	Raise to the power (find X <sup>B</sup> )

Note: Some teletypewriters use a ^ in place of the ↑

We must be careful with parentheses to make sure that we group together those things which we want together. We must also understand the order in which the computer does its work. For example, if we type A + B \* C ↑ D, the computer will first raise C to the power D, multiply this result by B, and then add A to the resulting product. This is the same convention as is usual for A + B C<sup>D</sup>. If this is not the order intended, then we must use parentheses to indicate a different 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 even add A to B, multiple their sum by C, and raise the product to the power D by writing ((A+B) \* C) ↑ D.

The order of priorities is summarized in the following rules:

1. The formula inside parentheses is computed before the parenthesized quantity is used in further computations.
2. In the absence of parentheses in a formula involving addition, multiplication, and the raising of a number to a power, the computer first raises the number to the power, then multiplies, then adds. Division has the same priority as multiplication, and subtraction the same as addition.
3. In the absence of parentheses in a formula involving only multiplication and division, the operations are done from left to right, as they are read. Addition and subtraction are also done from left to right.

These rules are illustrated in the previous example. The rules also tell us that the computer, faced with  $A - B - C$ , will (as usual) subtract  $B$  from  $A$  and then  $C$  from their difference; faced with  $A/B/C$ , it will divide  $A$  by  $B$  and that quotient by  $C$ . Given  $A \uparrow B \uparrow C$ , the computer will raise the number  $A$  to the power  $B$  and take the resulting number and raise it to the power  $C$ . If there is any question in your mind about the priority, put in more parentheses to eliminate possible ambiguities.

In addition to these five arithmetic operations, the computer can evaluate several mathematical functions. These functions are given special 3-letter English names, as the following list shows:

<u>Functions</u>	<u>Interpretation</u>	
SIN (X)	Find the sine of X	} X interpreted as a number, or 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 $e^x$	
LOG (X)	Find the natural logarithm of X (ln X)	
ABS (X)	Find the absolute value of X ( $ X $ )	
SQR (X)	Find the square root of X ( $\sqrt{X}$ )	

Two special functions, NUM and DET, are explained under "Matrices" in Chapter 3. Three other mathematical functions are also available in BASIC: INT, SGN, and RND. These are explained under "Functions" in Chapter 3. In place of X, we may substitute any formula or any number in parentheses following any of these formulas. For example, we may ask the computer to find  $\sqrt{4 + X^3}$  by writing SQR (4 + X  $\uparrow$  3), or the arctangent of  $3X - 2e^x + 8$  by writing ATN (3\*X-2\*EXP(X) + 8).

If, sitting at the teletypewriter, you need the value of  $(\frac{5}{6})^{17}$ , you can run the two-line program:

```
10 PRINT (5/6)  $\uparrow$  17
20 END
```

Since we have mentioned numbers and variables, we should be sure that we understand how to write numbers for the computer and what variables are allowed.

## Numbers

A number may be positive or negative and it may contain up to nine digits, but it must be expressed in decimal form. For example, all of the following are numbers in BASIC: 2, -3.675, 123456789, -.987654321, and 483.4156. The following are not numbers in BASIC: 14/3,  $\sqrt{7}$  and .00123456789. The first two are formulas, but not numbers, and the last one has more than nine digits. We may ask the computer to find the value of 14/3 or  $\sqrt{7}$  and to do something with the resulting number, but we may not include either in a list of DATA.

We gain further flexibility by use of the letter E, which stands for "times ten to the power." Thus, we may write .00123456789 in a form acceptable to the computer in any of several forms: .123456789E-2 or 123456789E-11 or 1234.56789E-6. We may write ten million as 1E7 and 1965 as 1.965E3. We do not write E7 as a number, but must write 1E7 to indicate that it is 1 that is multiplied by  $10^7$ . Numbers cannot be larger than 1.70141E38 or smaller than 1.49637E-39.

When entering a series of numbers, separate them by commas. The comma following the last number is optional.

## Variables

A variable in BASIC is denoted by any letter, or by any letter followed by a single digit. Thus, the computer will interpret E7 as a variable, along with A, X, N5, I0, and Ø1. A variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program was written. Variables are given or assigned values by FØR, LET, READ, or INPUT statements. The value assigned will not change until the next time a FØR, LET, READ, or INPUT statement is encountered with a value for that variable.

Note that, since all variables are set to zero before a RUN, it is necessary to assign a value to a variable only when you do not want it to be zero.

Six mathematical symbols are provided for in BASIC. These are symbols of relation 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 in Chapter 1. Any of the following six standard relations may be used:

<u>Symbol</u>	<u>Example</u>	<u>Meaning</u>
=	A = B	Is equal to (A is equal to B)
<	A < B	Is less than (A is less than B)
<=	A <= B	Is less than or equal to (A is less than or equal to B)
>	A > B	Is greater than (A is greater than B)
>=	A >= B	Is greater than or equal to (A is greater than or equal to B)
<>	A <> B	Is not equal to (A is not equal to B)

## LOOPS

We are frequently interested in writing a program in which one or more parts are done not just once but a number of times, perhaps with slight changes each time. In order to write the simplest program, the one in which the part to be repeated is written just once, we use the programming device known as a loop.

Programs which use loops can be illustrated and explained by two programs for the simple task of printing out a table of the first 100 positive integers together with the square root of each. Without a loop, our program would be 101 lines long and read:

```
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
```

With the following program, using one type of loop, we can obtain the same table with far fewer lines of instruction, 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 the line 20 is printed both 1 and its square root. Then, in line 30, X is increased by 1, to 2. Line 40 asks whether X is less than or equal to 100; an affirmative answer directs the computer back to line 20. Here 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  $4 \leq 100$  go back to line 20), etc.--until the loop has been traversed 100 times. Then, after it has printed 100 and its square root has been printed, 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: initialization (line 10), the body (line 20), modification (line 30), and an exit test (line 40).

Because loops are so important and because loops of the type just illustrated arise so often, BASIC provides two statements to specify a loop even more simply. They are the FOR and NEXT statements and their use is illustrated in the program:

```
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 set up, like that of line 40 above. Line 30 carries out two tasks: X is increased by 1, and the test is carried out to determine whether to go back to 20 or go on. Thus lines 10 and 30 take the place of lines 10, 30, and 40 in the previous program--and they are easier to use.

Note that the value of X is increased by 1 each time we go through the loop. If we wanted a different increase, say 5, we could specify it by writing

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

and the computer would assign 1 to X on the first time through the loop, 6 to X on the second time through, 11 on the third time, and 96 on the last time. Another step of 5 would take X beyond 100, so the program would proceed to the end after printing 96 and its square root. The STEP may be positive or negative, and we could have obtained the first table, printed in reverse order, by writing line 10 as

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

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

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, if N and Z have been specified earlier in the program, we could write

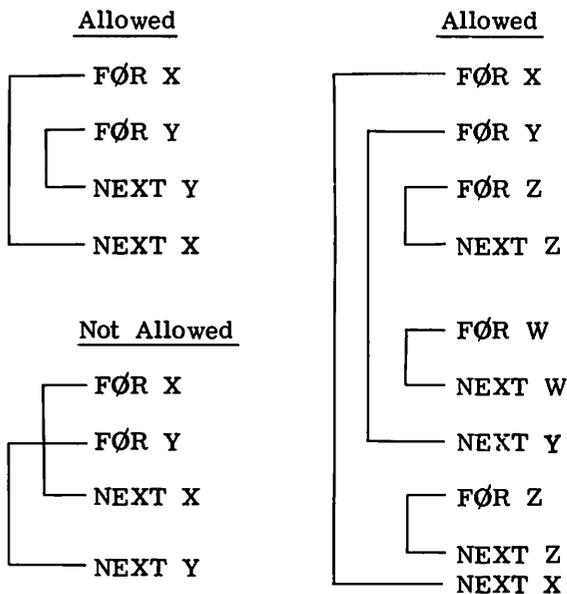
```
FOR X = N + 7*Z TO (Z-N) / 3 STEP (N-4*Z) / 10
```

For a positive 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 for negative step-size), then the body of the loop will not be performed at all, but the computer will immediately pass to the statement following the NEXT. As an example, the following program for adding up the first n integers will give 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
```

It is often useful to have loops within loops. These are called nested loops and can be expressed with FOR and NEXT statements. However, they must actually be nested and must not cross, as the following skeleton examples illustrate:



## LISTS AND TABLES

In addition to the ordinary variables used by BASIC, there are variables which can be used to designate the elements of a list or of a table. These are used where we might ordinarily use a subscript or a double subscript, for example the coefficients of a polynomial ( $a_0, a_1, a_2, \dots$ ) or the elements of a matrix ( $b_{i,j}$ ). The variables which we use in BASIC consist of a single letter, which we call the name of the list, followed by the subscripts in parentheses. Thus, we might write A(1), A(2), etc. for the coefficients of the polynomial and B(1,1), B(1,2), etc. for the elements of the matrix.

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

```
10 FOR I = 0 TO 10
20 READ A(I)
30 NEXT I
40 DATA 2, 3, -5, 7, 2.2, 4, -9, 123, 4, -4, 3
```

We need no special instruction to the computer if no subscript greater than 10 occurs. However, if we want larger subscripts, we must use a dimension (DIM) statement, to indicate to the computer that it has to save extra space extra space for the list or table. When in doubt, indicate a larger dimension than you expect to use. For example, if we want a list of 15 numbers entered, we might write:

```
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 form as typed would allow for the lengthening of the list by changing only statement 60, so long as it did not exceed 25.

We would enter a 3x5 table into a program by writing:

```
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
```

Here again, we may enter a table with no dimension statement, and it will handle all the entries from B(0,0) to B(10,10). If you try to enter a table with a subscript greater than 10, without a DIM statement, you will get an error message telling you that you have a subscript error. This is easily rectified by entering the line:

```
5 DIM B(20, 30)
```

if for instance, we need a 20 by 30 table.

The single letter denoting a list or a table name may also be used to denote a simple variable without confusion. However, the same letter may not be used to denote both a list and a table in the same program. The form of the subscript is quite flexible, and you might have the list item B(I + K) or the table items B(I,K) or Q(A(3,7), B - C).

The sample program which follows illustrates a LIST and RUN of a problem which uses both a list and a table. The program computes the total sales of each of five salesmen, all of whom sell the same three products. The list P gives the price/item of the three products and the table S shows the quantity of each item sold by each man. You can see from the program the product number 1 sells for \$1.25, number 2 for \$4.30, and number 3 for \$2.50 per item; and also that salesman number 1 sold 40 items of the first product, 10 of the second, and 35 of the third, and so on. The program reads in the price list in lines 10, 20, 30, using data in line 900, and the sales table in lines 40-80, using data in lines 910-930. The same program could be used again, modifying only line 900 if the prices change, and only lines 910-930 to enter the sales in another month.

This sample program did not need a dimension statement, since the computer automatically saves enough space to allow all subscripts to run from 0 to 10. A DIM statement is normally used to save more space. But in a long program, requiring many small tables, DIM may be used to save less space for tables, in order to leave more for the program.

Since a DIM statement is not executed, it may be entered into the program on any line before END; it is convenient, however, to place DIM statements near the beginning of the program.

```
SALES1          08:59

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
```

READY

RUN

```
SALES1          09:00
```

```
TOTAL SALES FOR SALESMAN 1    $ 180.5
TOTAL SALES FOR SALESMAN 2    $ 211.3
TOTAL SALES FOR SALESMAN 3    $ 131.65
TOTAL SALES FOR SALESMAN 4    $ 166.55
TOTAL SALES FOR SALESMAN 5    $ 169.4
```

## ERRORS AND DEBUGGING

It may occasionally happen that the first run of a new problem will be free of errors and give the correct answers. But it is much more common that errors will be present and will have to be corrected. Errors are of two types: errors of form (or 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 will cause error messages to be printed, and the various types of error messages are listed and explained in Appendix A. Logical errors are often much harder 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 in the last section, 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. Notice that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers

will start out using line numbers that are multiples of five or ten, but that is a matter of choice.

These corrections can be made at any time--whenever you notice them--either before or after a run. Since the computer sorts lines out and arranges them in order, a line may be retyped out of sequence. Simply retype the incorrect line with its original line number.

Although the computer does little in the way of "correcting," during computation, it will sometimes help you when you forget to indicate absolute value. For example, if you ask for the square root of -7 or the logarithm of -5, the computer will give you the square root of 7 with the error message that you have asked for the square root of a negative number, or the logarithm of 5 with the error message that you have asked for the logarithm of a negative number.

As with most problems in computing, we can best illustrate the process of finding the errors (or bugs) in a program, and correcting (or debugging) it, by an example. Let us consider the problem of finding the value of X between 0 and 3 for which the sine of X is a maximum, and ask the system to print out this value of X and the value of its sine. If you have studied trigonometry, you know that  $\pi/2$  is the correct value; but we shall use the system to test successive values of X from 0 to 3, first using intervals of .1, then of .01, and finally of .001. Thus, we shall ask the system 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 will do it by testing SIN (0) and SIN (.1) to see which is larger, and calling the larger of these two numbers M. Then it will pick the larger of M and SIN (.2) and call it M. This number will be checked against SIN (.3) and so on down the line. 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 of the 31 sines, and X0 will be the argument that produced that largest value. It will then repeat 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 sine is the largest. Lastly, it will check the 3001 numbers 0, .001, .002, .003, . . . ., 2.998, 2.999, and 3, to find 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 teletypewriter, we write a program and let us assume that it is 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
```

We shall list the entire sequence on the teletypewriter and make explanatory comments on the right side.

```
NEW ØR ØLD--NEW
NEW FILE NAME-- MAXSIN
READY
```

```
10 READ D
20 LWR XØ= 0
30 FØR X = 0 TØ 3 STEP D
40 IF SINE*(X) <= M THEN 100
50 LET XØ=X
60 LET M = SIN(X)
70 PRINT XØ, X, D
80 NEXT Z*XØ
90 GØ TØ 20
20 LET XØ=0
100 DATA .1, .01, .001
110 END
RUN
```

```
MAXSIN      11:10
```

```
ILLEGAL VARIABLE IN 70
NEXT WITHOUT FØR IN 80
FØR WITHOUT NEXT IN 30
```

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

```
MAXSIN      11:12
```

```
0.1          0.1
0.2          0.2
0.3
```

```
TIME:      .07 SECS.
```

```
20
RUN
```

```
MAXSIN      11:13
```

```
UNDEFINED LINE NUMBER 20 IN 90
```

```
TIME:      .07 SECS.
```

```
90 GØ TØ 10
RUN
```

Notice the use of the backwards arrow (on some teletypewriters, an underline) to erase a character in line 40, which should have started IF SIN(X) etc., and in line 80.

After typing line 90, we notice that LET was mistyped in line 20, so we retype it, this time correctly.

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

We make both of these changes by retyping lines 70 and 80. In looking over the program, we also notice that the IF-THEN statement in 40 directed the computer to a DATA statement and not to line 80 where it should go.

```
0.1 This is obviously incorrect. We are having
0.1 every value of X printed, so we direct the
machine to cease operations by pressing control,
shift, and P, even while it is running. We ponder
the program for a while, trying to figure out
what is wrong with it. We notice that SIN(0) is
compared with M on the first time through the
loop, but we had assigned a value to XØ but
not to M. However we recall that all variables
are set equal to zero before a RUN so that
line 20 is unnecessary.
```

Of course, line 90 sent us back to line 20 to repeat the operation and not back to line 10 to pick up a new value for D.

```

MAXSIN      11:13
0.1         0.1         0.1
0.2         0.2         0.1
0.3
TIME:      .07 SECS.

```

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

```

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

```

We fix this 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 and not X. We also decide to put in headings for our columns by a PRINT statement.

```

MAXSIN      11:18

ILLEGAL VARIABLE IN 5

```

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

```

TIME:      .06 SECS.

5 PRINT "X VALUE", "SINE", "RESOLUTION"
RUN

MAXSIN      11:19

```

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

```

X VALUE      SINE      RESOLUTION
1.6          0.999574   0.1
1.57         1.         0.01
1.57099      1.         0.001
OUT OF DATA IN 10

```

Exactly 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. Similarly for the finer subdivisions.

```

TIME:      1.68 SECS.

```

The whole process took 1.68 seconds of the computer's time.

```

LIST

MAXSIN      11:19

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 XO=X
60 LET M = SIN(X)
80 NEXT X
85 PRINT XO, M, D
90 GO TO 10
100 DATA .1, .01, .001
110 END

```

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

```

READY

SAVE
READY

```

The program is saved for later use. This should not be done unless future use is necessary.

In solving this problem, there are two common devices which we did not use. One is the insertion of a PRINT statement when we wonder if the machine is computing what we think we asked it to compute. For example, if we wondered about M, we could have inserted 65 PRINT M, and we would have seen the values. The other device is used after several corrections have been made and you are not sure just what the program looks like at this stage - in this case type LIST, and the computer will type out the program in its current form for you to inspect.

## SUMMARY OF ELEMENTARY BASIC STATEMENTS

In this section we shall give a short and concise description of each of the types of BASIC statements discussed earlier in this chapter. In each form, we shall assume a line number, and shall use underlining to denote a general type. Thus, variable refers to a variable, which is a single letter, possibly followed by a single digit.

### LET

This statement is not a statement of algebraic equality, but is rather a command to the system to perform certain computations and to assign the answer to a certain variable. LET statements may be of the form: LET variable = formula.

Examples:     100 LET X = X + 1  
              259 LET W7 = (W-X4 3)\*(Z - A/(A - B)) - 17

Multiple assignments may be made with a single LET statement. For example,

LET X = Y3 = A(3,I) = 1

### READ and DATA

We use a READ statement to assign to the listed variables values obtained from a DATA statement. Neither statement is used without one of the other type. 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 system takes all of the DATA statements in the order in which they appear and creates a 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, with a READ statement still asking for more, the program is assumed to be done 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 and each DATA statement of the form: DATA sequence of numbers

Examples:     150   READ X, Y, Z, X1, Y2, Q9  
              330   DATA 4, 2, 1.7  
              340   DATA 6.734E-3, -174.321, 3.14159265  
  
              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

When entering numeric values, remember that only numbers are put in a DATA statement, and that  $15/7$  and  $\sqrt{3}$  are expressions, not numbers.

## PRINT

The PRINT statement has a number of different uses and is discussed in more detail in Chapter 3. The common uses are:

- a. To print out the result of some computations
- b. To print out verbatim a message included in the program
- c. To perform a combination of a and b
- d. To skip a line

We have seen examples of only the first two in our sample programs. Each type is slightly different in form, but all start with PRINT after the line number.

Examples of type a:   100 PRINT X, SQR (X)  
                          135 PRINT X, Y, Z, B\*B - 4\*A\*C, EXP (A - B)

The first will print X and then, a few spaces to the right of that number, its square root. The second will print five different numbers: X, Y, Z,  $B^2 - 4AC$ , and  $e^{A-B}$ . The system will compute the two formulas and print them for you, as long as you have already given values to A, B, and C. It can print up to five numbers per line in this format.

Examples of type b:   100 PRINT "NO UNIQUE SOLUTION"  
                          430 PRINT "X VALUE", "SINE", "RESOLUTION"

Both have been encountered in the sample programs. The first prints that simple statement; the second prints the three labels with spaces between them. The labels in 430 automatically line up with three numbers called for in a PRINT statement - as seen in MAXSIN.

Examples 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, the system will print out: THE VALUE OF X IS 3. If the second has computed the value of X to be 625, the system will print out: THE SQUARE ROOT OF 625 IS 25.

Example of type d:   250 PRINT

The system will advance the paper one line when it encounters this command.

## GØ TØ

There are times in a program when you do not want all commands executed in the order that they appear in the program. An example of this occurs in the MAXSIN problem where the system has computed X0, M, and D and printed them out in line 85. We did not want the program to go on to the END statement yet, but to go through the same process for a different value of D. So we directed the system to go back to line 10 with a GØ TØ statement. Each is of the form GØ TØ line number.

Example:   150   GØ TØ 75

## IF—THEN or IF—GØ TØ

There are times that we are interested in jumping the normal sequence of commands, if a certain relationship holds. For this we use an IF--THEN statement, sometimes called a conditional GØ TØ statement. Such a statement occurred at line 40 of MAXSIN. Each such statement is of the form

IF formula relation formula THEN line number   or IF formula relation formula GØ TØ  
line number

## SUMMARY OF ELEMENTARY BASIC STATEMENTS

Examples: 40 IF SIN (X) <= M THEN 80 or 40 IF SIN (X) <= M GØ TØ 80  
20 IF G = 0 THEN 65 or 20 IF G = 0 GØ TØ 65

Line 40 asks if the sine of X is less than or equal to M, and directs the system to skip to line 80 if it is. Line 20 asks if G is equal to 0, and directs the system to skip to line 65 if it is. In each case, if the answer to the question is No, the system will go to the next line of the program.

### FØR and NEXT

We have already encountered the FØR and NEXT statements in our loops, and have seen that they go together, one at the entrance to the loop and one at the exit, directing the system back to the entrance again. Every FØR statement is of the form

FØR variable = formula TØ formula STEP formula

Most commonly, the formulas will be integers and the STEP omitted. In the latter case, a step size of one is assumed. The accompanying NEXT statement is simple in form, but the variable must be precisely the same one as that following FØR in the FØR statement. Its form is NEXT variable.

Examples: 30 FØR X = 0 TØ 3 STEP D  
80 NEXT X  
  
120 FØR X4 = (17 + CØS (Z))/3 TØ 3\*SQR (10) STEP 1/4  
235 NEXT X4  
  
240 FØR X = 8 TØ 3 STEP -1  
  
456 FØR J = -3 TØ 12 STEP 2

Notice that the step size may be a formula (1/4), a negative number (-1), or a positive number (2). In the example with lines 120 and 235, the successive values of X4 will be .25 apart, in increasing order. In the next example, the successive values of X will be 8, 7, 6, 5, 4, 3. In the last example, on successive trips through the loop, J will take on values -3, -1, 1, 3, 5, 7, 9, and 11.

If the initial, final, or step-size values are given as formulas, these formulas are evaluated only once, upon entering the FØR statement. The control variable can be changed in the body of the loop; of course, the exit test always uses the latest value of this variable.

If you write 50 FØR Z = 2 TØ -2, without a negative step size, the body of the loop will not be performed and the system will proceed to the statement immediately following the corresponding NEXT statement.

FØR statements cannot be nested to a depth greater than 20 levels.

### DIM

Whenever we want to enter a list or a table with a subscript greater than 10, we must use a DIM statement to instruct the system to save us sufficient room for the list or the table.

Examples: 20 DIM H (35)  
35 DIM Q (5,25)

The first would enable us to enter a list of 36 items and the second a table 6 x 26.

### END

Every program must have an END statement, and it must be the statement with the highest line number in the program. Its form is simple: a line number with END.

Example: 999 END

## 3. Advanced Basic

---

### MORE ABOUT PRINT

The uses of the PRINT statement were described in Chapter 2, but we shall give more detail here. Although the format of answers is automatically supplied for the beginner, the PRINT statement permits a greater flexibility for the more advanced programmer who wishes a different format for his output.

The teletypewriter line is divided into five zones of fifteen spaces each. Some control of use of these comes from the use of the comma: a comma is a signal to move to the next print zone or, if the fifth print zone has just been filled, to move to the first print zone of the next line.

A closer grouping of numbers can be obtained by use of the semicolon. Numbers printed next to each other by use of the semicolon will be in closest readable format. For example, if you were to type the program

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

the teletypewriter would print 1 at the beginning of a line, 2 at the beginning of the next line, and so on, finally printing 15 on the fifteenth line. But, by changing line 20 to read

```
20 PRINT I,
```

you would have the numbers printed in the zones, reading

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

If you wanted the numbers printed in this fashion, but more tightly packed, you would change line 20 to replace the comma by a semicolon:

```
20 PRINT I;
```

and the result would be printed

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

You should remember that a label inside quotation marks is printed just as it appears and also that the end of a PRINT line signals a new line, unless a comma or semicolon is the last symbol. When a label is followed by a semicolon, the label is printed with no space after it.

Thus, the instruction

```
50 PRINT X, Y
```

will result in the printing of two numbers and the return to the next line, while

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

will result in the printing of these two values and no return--the next number to be printed will occur in the third zone, after the values of X and Y in the first two.

Since the end of a PRINT statement signals a new line, you will remember that

```
250 PRINT
```

will cause the teletypewriter to advance the paper one line. It puts a blank line in your program, if you want to use it for vertical spacing of your results, or it causes the completion of a partially filled line, as illustrated in the following fragment of a program:

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

This program will print B(1,1). Without line 140, the teletypewriter would then go on printing B(2,1), and B(2,2) on the same line, and even B(3,1), B(3,2), etc., if there were room. Line 140 directs the teletypewriter, after printing the B(1,1) value corresponding to M = 1, to start a new line and to do the same thing after printing the value of B(2,2) corresponding to M = 2, etc.

The following rules for the printing of numbers will help you in interpreting your printed results:

1. If a number is an integer, the decimal point is not printed. If the integer contains more than nine digits, the teletypewriter will give you the first digit, followed by (a) a decimal point, (b) the next five digits, and (c) and E followed by the appropriate integer. For example, it will take 32, 437, 580, 259 and write it as 3.24376 E 10.
2. For any decimal number, no more than six significant digits are printed.
3. 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 six decimal number. Thus, .03456 means that the number is exactly .034560000, while 3.45600 E -2 means that the number has been rounded to .0345600.
4. Trailing zeros after the decimal point are not printed.

The following program, in which we print out the first 45 powers of 2, shows how numbers are printed. Note that the semicolon "packed" form sometimes causes the last few characters in a number to be printed on top of each other. BASIC checks to see if there are 12 or more spaces at the end of a line before printing a number there, but some numbers require 15 spaces.

```
10 FOR I=1 TO 45
20 PRINT 2*I;
30 NEXT I
40 END
```

READY

RUN

PRINT 09:04

```
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 1.34218 E+8 2.68435 E+8 5.36871 E+8
1.07374 E+9 2.14748 E+9 4.29497 E+9 8.58993 E+9 1.71799 E+10
3.43597 E+10 6.87195 E+10 1.37439 E+11 2.74878 E+11 5.49756 E+11
1.09951 E+12 2.19902 E+12 4.39805 E+12 8.79609 E+12 1.75922 E+13
3.51844 E+13
```

The TAB function permits you to specify tabulated formatting. For example, TAB (17) would cause the teletypewriter to move to column 17. Positions on a line are numbered 0 through 74; 75 is assumed to be position 0 again.

TAB may contain any formula as its argument. The value of the formula is computed, and its integer part is taken. This in turn is treated modulo 75, to obtain a value 0 through 74. The teletypewriter is then moved to this position. If it has already passed this position, the TAB is ignored. For example, inserting the following line in a loop:

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

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

## FUNCTIONS

There are three functions which were listed in Chapter 2 but not described. These are INT, SGN, and RND. Also there are ten other functions that will sometimes be useful: TIM, CLK\$, DAT\$, HPS, VPS, LIN, ASC, STR\$, VAL, and LEN.

### INT

The INT function is the function which frequently appears in algebraic computation  $[x]$ , and it gives the greatest integer not greater than  $x$ . Thus  $\text{INT}(2.35) = 2$ ,  $\text{INT}(-2.35) = -3$ , and  $\text{INT}(12) = 12$ .

One use of the INT function is to round numbers. We may use it to round to the nearest integer by asking for  $\text{INT}(X + .5)$ . This will round 2.9, for example, to 3, by finding:

$$\text{INT}(2.9 + .5) = \text{INT}(3.4) = 3$$

You should convince yourself that this will indeed do the rounding guaranteed for it (it will round a number midway between two integers up to the larger of the integers).

It can also be used to round to any specific number of decimal places. For example,  $\text{INT}(10*X + .5)/10$  will round X correct to one decimal place,  $\text{INT}(100*X + .5)/100$  will round X correct to two decimal places, and  $\text{INT}(X*10^D + .5)/10^D$  will round X correct to D decimal places.

### SGN

The function SGN (argument) yields +1, -1, or 0 depending on the value of the argument. These are the options:

<u>Function</u>	<u>Argument Value</u>	<u>Yield</u>
SGN	Zero	0
SGN	Positive, not zero	+1
SGN	Negative, not zero	-1

## Examples:

SGN (0) yields 0  
 SGN (-1.82) yields -1  
 SGN (989) yields +1  
 SGN (-.001) yields -1  
 SGN (-0) yields 0

## RND

The function RND produces a random number between (not including) 0 and 1. No argument is required. For example you may type: LET A = RND.

If we want the first twenty random numbers, we write the program below and we get twenty six-digit decimals. This is illustrated in the following program.

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

READY

RUN

RNDTES 09:07

0.406533	0.88445	0.681969	0.939462	0.253358
0.863799	0.880238	0.638311	0.602898	0.990032
0.570427	0.897931	0.628126	0.613262	0.303217
5.00548 E-2	0.393226	0.680219	0.632246	0.668218

On the other hand, if we want twenty random one-digit integers, we could change line 20 to read

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

and we would then obtain

RNDTES 10:57

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

We can vary the type of random numbers we want. For example, if we want 20 random numbers ranging from 1 to 9 inclusive, we could change line 20 as shown

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

RNDTES 10:53

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

or we can obtain random numbers which are the integers from 5 to 13 inclusive by changing line 20 as in the example which follows:

```
20 PRINT INT(9*RND+5);  
RUN
```

```
RNDTRES      10:59
```

```
3 12 11 13 7 12 12 10 10 13 10 13 10 10 7 5 8 11 10
```

In general, if we want our random numbers to be chosen from the A integers of which B is the smallest, we would call for

```
INT (A*RND + B).
```

## TIM

The TIM function provides the elapsed execution time in seconds.

Example:

```
10 FOR X = 1 TO 50  
20 LET A = X  
30 NEXT X  
40 PRINT "ELAPSED TIME IS: ";TIM  
99 END
```

```
RUN
```

```
EXAMP      17:27
```

```
ELAPSED TIME IS: 14.95
```

The execution time may be assigned a variable name.

Example:

```
10 FOR X = 1 TO 50  
20 LET A = X  
30 NEXT X  
40 LET B = TIM  
50 PRINT "ELAPSED TIME IS: ";B  
99 END
```

```
RUN
```

```
EXAMP      15:04
```

```
ELAPSED TIME IS: 14.9508
```

The execution times provided by the TIM function will not agree with the total time printed out at the end of a program run. This is so because TIM provides only execution time, but the total time includes compilation, execution, and termination times.

## CLK\$

The CLK\$ function provides the time of day as a string.

Example:

```
10 LET A$ = CLK$
20 PRINT A$;
99 END
```

This program will print the time of day, such as

10:34

The CLK\$ function may be printed out without assigning it to a string variable.

Example:

```
10 PRINT CLK$;
99 END
```

will print out the time of day as a string, exactly as in the previous example.

## DAT\$

The DAT\$ function provides the calendar date as a string.

Example:

```
10 LET A$ = DAT$
20 PRINT A$;
99 END
```

will print the date in the form

10/08/68

Like the CLK\$ function, the DAT\$ function need not be assigned to a string variable.

Example:

```
10 PRINT DAT$;
99 END
```

will also print the date, exactly as in the previous example.

## HPS

The form of the HPS function is

HPS (file designator)

and it gives the character position in the current line of the file being read or written, starting after the line number.

Example:

The program

```
10 FOR X = 1 TO 20
20 PRINT X;
30 NEXT X
40 PRINT HPS(0)
99 END
```

RUN

POSIT 17:33

```
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
71
```

The character position when the HPS(0) statement was reached was 71.

The character position may be assigned a variable name.

Example:

```
10 FOR X = 1 TO 25
20 PRINT X;
30 NEXT X
40 LET B = HPS(0)
50 PRINT
60 PRINT "CHARACTER POSITION IS: "; B
99 END
```

RUN

VARNAME 17:36

```
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25
CHARACTER POSITION IS: 20
```

## VPS

The form of the VPS function is

VPS (file designator)

and it gives the current number of lines in a file being read or written.

Example:

The program

```
10 FILES A
20 SCRATCH #1
30 FOR X = 1 TO 60
40 WRITE #1,X
50 NEXT X
60 LET N = VPS(1)
70 PRINT "NO. OF LINES IN A = "; N
99 END
```

prints out

```
NO. OF LINES IN A = 60
```

A zero file designator will refer to the teletypewriter.

## LIN

The form of the LIN function is

LIN (file designator)

and it provides the last line number encountered in reading from or writing to a file.

## Example:

```
10 FILES A
20 SCRATCH #1
30 FOR X = 1 TO 60
40 WRITE #1,X
50 NEXT X
60 PRINT LIN(1)
99 END
```

This program will print out the last line number written to file A:

690

The listing of file A

```
A                17:44

100 1 ,
110 2 ,
120 3 ,
130 4 ,
140 5 ,
150 6 ,
160 7 ,
170 8 ,
180 9 ,
190 10 ,
200 11 ,
210 12 ,
220 13 ,
230 14 ,
240 15 ,
250 16 ,
260 17 ,
270 18 ,
280 19 ,
290 20 ,
300 21 ,
310 22 ,
320 23 ,
330 24 ,
340 25 ,
350 26 ,
360 27 ,
370 28 ,
380 29 ,
390 30 ,
400 31 ,
410 32 ,
420 33 ,
430 34 ,
440 35 ,
450 36 ,
460 37 ,
470 38 ,
480 39 ,
490 40 ,
500 41 ,
510 42 ,
520 43 ,
530 44 ,
540 45 ,
550 46 ,
560 47 ,
570 48 ,
580 49 ,
590 50 ,
600 51 ,
```

```
610 52 ,
620 53 ,
630 54 ,
640 55 ,
650 56 ,
660 57 ,
670 58 ,
680 59 ,
690 60 ,
```

verifies that the last line number in the file is 690.

A variable may be assigned the value of the function LIN.

Example:

```
10 FILES A
20 SCRATCH #1
30 FOR X = 1 TO 60
40 WRITE #1,X
50 NEXT X
60 LET A = LIN(1)
70 PRINT A;
99 END
```

In this program the variable named A is assigned the value of LIN(1). The program when run would produce the same results as the previous example.

## ASC

The form of the ASC function is

ASC (character)

or

ASC (abbreviation)

and it provides the numeric value of the specified ASCII character or (for non-printing characters) abbreviation.

Example:

The program

```
10 PRINT ASC(?)
20 PRINT ASC(CR)
30 PRINT ASC(LF)
40 PRINT ASC(R)
99 END
```

will produce results of

63

13

10

82

which are the numeric representations of the specified characters, ?, carriage return, line feed, and R.

A variable may be assigned the value of the function ASC.

Example:

```
10 LET C = ASC(CR)
20 PRINT C;
99 END
```

This program will assign the variable named C the numeric representation of the ASCII carriage return. The program when run will print out

13

## STR\$

The STR\$ function has the form

STR\$(N)

where N is a number. The function produces a string corresponding to the value of the number N.

Example:

The program

```
10 INPUT N
20 LET X$ = STR$(N)
30 PRINT X$
99 END
```

will print out, if the value 12 is entered when input is requested, the string

12

Example:

The program

```
10 INPUT X,Y
20 LET N = X*Y
30 LET C$ = STR$(N)
40 PRINT "THE STRING CORRESPONDING TO N IS:";C$
99 END
```

produces the output

```
? 20,35
THE STRING CORRESPONDING TO N IS:700
```

In this example the number N, computed by the program, is converted to a string.

## VAL

The form of the VAL function is

VAL(S\$)

where S\$ is a proper number. This function produces a number corresponding to the value of the string represented by S\$. This allows string variables to be used in arithmetic expressions.

Example:

The program

```
10 LET A$ = "12"  
20 LET B = VAL(A$)  
30 LET Q = 2*VAL(A$)  
40 PRINT Q;  
50 PRINT B;  
99 END
```

will give results of

```
24    12
```

for Q and B respectively.

Example:

The program

```
10 INPUT A$  
20 LET Z = VAL(A$)  
30 PRINT Z;  
99 END
```

will produce, with the input 5E5, the corresponding number

```
500000
```

If the string does not represent a number, an error message is printed out.

## LEN

The form of the LEN function is

```
LEN(X$)
```

where X\$ is the name of any string. The function gives the number of characters in the specified string. The value of LEN may be assigned to a variable or used directly.

Example:

The program

```
10 READ A$, B$, C$  
20 LET A = LEN(A$)  
30 LET B = LEN(B$)  
40 LET C = LEN(C$)  
50 PRINT "A$ ="; A$; "    LENGTH OF A$ ="; A  
60 PRINT "B$ ="; B$; "    LENGTH OF B$ ="; B  
70 PRINT "C$ ="; C$; "    LENGTH OF C$ ="; C  
80 DATA ABC, DEFGH, IJKLMNOPQRST  
99 END
```

produces the results

```
A$ =ABC    LENGTH OF A$ = 3  
B$ =DEFGH  LENGTH OF B$ = 5  
C$ =IJKLMNOPQRST  LENGTH OF C$ = 12
```

Example:

The program

```
10 READ A$, B$, C$
20 PRINT LEN(A$)
30 PRINT LEN(B$)
40 PRINT LEN(C$)
50 DATA ABC, DEFG, HIJKLMN
99 END
```

produces

```
3
4
7
```

## RANDØMIZE

The RANDØMIZE (or RANDØM) statement can be used in conjunction with the random number function to induce variance. It augments the function RND by causing it to produce different sets of random numbers. For example, if this is the first instruction in the program using random numbers, then repeated program execution will generally produce different results. When this instruction is omitted, the "standard list" of random numbers is obtained.

It is suggested that a simulation model should be debugged without RANDØM, so that you always obtain the same random numbers for test runs. After your program is debugged, you may insert

```
1 RANDØM
```

before starting execution runs.

## DEF

In addition to the standard functions, you can define other functions which you expect to use a number of times in your program by use of a DEF statement. The name of the defined function must be three letters, the first two of which are FN. Hence, you may define up to 26 functions, e.g., FNA, FNB, etc.

If a function can be defined in a single line, it takes the following form:

```
30 DEF FNE (X) = EXP(-X↑2)
```

Later on you can call for various values of the function by FNE(.1), FNE(3.45), FNE(A + 2), etc. Such a definition can be a great labor saver when you want values of some function for a number of different values of the variable.

Each function may have zero, one, two, or more variables, providing the definition fits on one line. For example, we may type:

```
DEF FNB(X,Y) = 3*X*Y - Y↑3
DEF FNC(X,Y,Z,W) = FNB(X,Y)/FNB(Z,W)
```

The DEF statement may occur anywhere in the program, and the expression to the right of the equal sign may be any formula which can be fitted onto one line. It may include any combination of other functions, including ones defined by different DEF statements, and it can involve other variables besides the one denoting the argument of the function. Thus, assuming FNR is defined by

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

if you have previously assigned values to Y and Z, you can ask for FNR (2.175). You can give new values to Y and Z before the next use of FNR.

If a function requires more than one line for its definition, introduce the function with a DEF statement containing no = and end the definition with a statement FNEND. For example:

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

The function will assume the last value assigned to the function name (i.e. lines 20 or 40).

Multiple line DEF's may not be nested, and there must not be a transfer from inside the DEF to outside its range, nor vice-versa.

Variables other than the arguments can be used and assigned values in multistatement functions. These variables may be global, which means that they can be used both inside and outside the function definition on either side of the equals sign; or they may be local, which means they are defined only within the function definition. Normally they are global. To specify a variable as being local, list the variable name in the DEF statement following the function name and arguments, for example:

```
DEF FNM(X,Y) P,Q
```

P and Q are specified as local variables, and bear no relation to P or Q used outside of the function definition.

## GOSUB and RETURN

When a particular part of a program is to be performed more than one time, or possibly at several different places in the over-all program, it is most efficiently programmed as a subroutine. The subroutine is entered with a GOSUB statement, where the number is the line number of the first statement in the subroutine. For example,

```
80 GOSUB 200
```

directs the system to line 200, the first line of the subroutine. The last line of the subroutine should be a RETURN statement directing the system to return to the earlier part of the program. For example,

```
310 RETURN
```

will tell the system to go back to the first line numbered greater than 80 and to continue the program there.

The following 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 200-310. 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.

You may use a GOSUB inside a subroutine to perform yet another subroutine. This would be called "nested GOSUBs." In any case, it is absolutely necessary that a subroutine be left only with a RETURN statement; using a GOTO or an IF-THEN to get out of a subroutine will not work properly. You may have several RETURNS in the subroutine. The first RETURN statement executed in a subroutine causes a return to the earlier part of the program.

You must be very careful not to write a program in which a `GOSUB` appears inside a subroutine which refers to one of the subroutines already entered. (Recursion is not allowed!)

```

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 GOT 20
110 DATA 60,90,120
120 DATA 38456,64872,98765
130 DATA 32,384,72
200 LET Q=INT(X/Y)
210 LET R=X-Q*Y
220 IF R=0 GOTO 300
230 LET X=Y
240 LET Y=R
250 GOTO 200
300 LET G=Y
310 RETURN
320 END

```

READY

RUN

GCN3N0            09:14

A	B	C	GCD
60	90	120	30
38456	64872	98765	1
32	384	72	8

OUT OF DATA IN 20

## ON

The `IF ... THEN` statement discussed in Chapter 2 allows a two-way conditional switch in a program. The `ON` statement provides a multiple switch. For example, consider the following:

```
ON X GOTO 100, 200, 150
```

- If `X = 1` the program branches to line number 100.
- If `X = 2` it goes to line 200.
- If `X = 3` it goes to line 150.

Any formula may replace `X` and there may be any number of line numbers in the instruction providing it fits on one line.

The value of the formula is computed and its integer part is taken. If this equals 1, the program transfers to the first specified number in the list.

If the integer part equals 2 the program transfers to the second number and so forth. If the integer part is less than 1 or larger than the number of line numbers specified, an error message is printed.

Note the use of `ON - GO TO` in line 120 of the following example:

```
100 IF X>0 GO TO 900
110 FOR X=1 TO 3
120 ON X GO TO 200,300,400
200 PRINT 200
210 GO TO 500
300 PRINT 300
310 GO TO 500
400 PRINT 400
500 NEXT X
600 STOP
900 END
```

READY

RUN

ONGOTO 09:16

```
200
300
400
```

## INPUT

There are times when it is desirable to have data entered during running of a program. This is particularly true when one person writes the program and enters it into memory, and other persons are to supply the data. This may be done by an INPUT statement, which acts as a READ statement but does not draw numbers from a DATA statement. If, for example, you want the user to supply values for X and Y into a program, you will type

```
40 INPUT X,Y
```

before the first statement which is to use either of these numbers. When it encounters this statement, the system will type a question mark. The user types two numbers, separated by a comma, presses the return key, and the system goes on with the rest of the program.

Frequently an INPUT statement is combined with a PRINT statement to make sure that the user knows what values to put in. You might type

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

and the system will type

```
WHAT ARE YOUR VALUES OF X, Y, AND Z?
```

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. Furthermore, it may take a long time to enter a large amount of data using INPUT. Therefore, INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the running of the program such as with game-playing programs.

## CHAIN

The CHAIN statement allows the user to stop execution of the current program and begin

compilation and execution of another program without direct intervention. It has the same effect as giving the commands `STOP`, `OLD`, a program name, and `RUN`. The form of the statement is

```
CHAIN "new file name"
```

or

```
CHAIN X$
```

The name of the file to be accessed is enclosed in quotation marks unless a string variable is used. Both the file containing the `CHAIN` statement and the file to be accessed must be saved before chaining can be done.

Example:

```
NEW
```

```
NEW FILE NAME-- CHAIN
```

```
READY
```

```
10 CHAIN "NEXT1"
```

or

```
10 LET A$ = "NEXT1"
```

```
20 END
```

```
20 CHAIN A$
```

```
SAVE
```

```
30 END
```

```
READY
```

```
RUN
```

In this example, the program `CHAIN` chains to the program `NEXT1`. Both programs must have been saved. If `SAVE` had been omitted in the example, the error message

```
CURRENT FILE MUST BE SAVED BEFORE CHAIN CAN BE PERFORMED
```

would have been printed out. If the file `NEXT1` had not been saved, the error message

```
FILE NOT SAVED
```

would have been printed out.

In the above example, the file `CHAIN` would now no longer be active, and would have to be re-called if corrections to it were necessary.

If a file with a password is named in a `CHAIN` statement, the password must be given before execution of the named program can begin.

Example:

```
10 CHAIN "FAT"  
20 END
```

```
RUN
```

```
PASS          18:30
```

```
GIVE PASSWORD RAT
```

Be careful when specifying a file in a `CHAIN` statement. No leading blanks are allowed in the

file name, but trailing blanks are allowed.

Example:

The program

```
10 CHAIN "NEXT1"  
20 END
```

will successfully chain to NEXT1, but a program such as

```
10 CHAIN " NEXT1"  
20 END
```

will produce the error message

```
FILE NOT SAVED
```

The CHAIN function does not allow you to chain to a line number within a file. Because there is no logic path to any statement following the CHAIN statement, all needed current program statements must be executed before the CHAIN statement.

## OTHER USEFUL STATEMENTS

Several other BASIC statements that may be useful from time to time are STOP, REM, RESTORE, TRACE ON, and TRACE OFF.

### STOP

STOP is equivalent to GOTO xxxxx, where xxxxx is the line number of the END statement in the program. It is useful in programs having more than one natural finishing point. For example, the following two program portions are exactly equivalent.

```
250 GOTO 999          250 STOP  
   :                 :  
340 GOTO 999          340 STOP  
   :                 :  
999 END              999 END
```

### REM

REM provides a means for inserting explanatory remarks in a program. The system completely ignores the remainder of that line, allowing the programmer to follow the REM with directions for using the program, with identifications of the parts of a long program, or with anything else that he wants. Although what follows REM is ignored, its line number may be used in a GOSUB, IF-THEN, GOTO, or ON-GOTO statement.

```
100 REM INSERT DATA 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
```

Explanatory remarks may be located following a statement on a line, by using the character " ' ". Anything on the line following the " ' " will be treated as an explanatory remark. For example, the statement

```
250 LET Y = 1 ' INITIALIZE Y TO ONE
```

includes the remark INITIALIZE Y TO ONE without affecting the running of the program.

## RESTORE

Sometimes it is necessary to use the data in a program more than once. The RESTORE statement permits reading the data as many additional times as it is used. Whenever RESTORE is encountered in a program, the system restores the data block pointer to the first number. A subsequent READ statement will then start reading the data all over again. A word of warning--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 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
```

## TRACE ON and TRACE OFF

TRACE ON and TRACE OFF statements may be useful in debugging programs. The TRACE ON statement causes the line number of each subsequent statement that is executed to be printed out. The TRACE OFF statement causes the printing out of line numbers to stop.

Example:

```
10 READ X, Y, Z
20 TRACE ON
30 LET A = X+(Y*Z)
40 IF A = 0 THEN 99
50 PRINT "    A NOT ZERO"
60 GO TO 10
88 DATA -9, 3, 4, -12, 3, 4
99 TRACE OFF
100 END
```

RUN

TRC 18:41

```
*AT 30
*AT 40
*AT 50
    A NOT ZERO
*AT 60
*AT 30
*AT 40
*AT 99
```

TRACE ON may be used without TRACE OFF.

Example:

```
10 READ X,Y,Z
20 TRACE ON
30 LET A = X+(Y*Z)
40 IF A = 0 THEN 99
50 PRINT "      A NOT ZERO"
60 GO TO 10
88 DATA -9, 3, 4, -12, 3, 4
99 END
```

RUN

TRC            18:42

```
*AT 30
*AT 40
*AT 50
   A NOT ZERO
*AT 60
*AT 30
*AT 40
*AT 99
```

## MATRICES

Although you can work out for yourself programs which involve matrix computations, there is a special set of statements for such computations. These statements must start with the word MAT. They are:

MAT READ A, B, C, ...	Read the matrices A, B, C, ..., their dimensions having been previously specified. Data is read in row-wise sequence.
MAT PRINT A, B; C, ...	Print the matrices A, B, C, ..., with A and C in the regular format, but B closely packed.
MAT B = A	Set the matrix B equal to the matrix A.
MAT C = A + B	Add the two matrices A and B and store the result in matrix C.
MAT C = A - B	Subtract the matrix B from the matrix A and store the result in matrix C.
MAT C = A * B	Multiply the matrix A by the matrix B and store the result in matrix C.
MAT C = INV (A)	Invert the matrix A and store resulting matrix in C.
MAT C = TRN (A)	Transpose the matrix A and store the resulting matrix in C.
MAT C = (K) * A	Multiply the matrix A by the value represented by K. K may be either a number or an expression, but in either case it must be enclosed in parentheses.
MAT C = CON	Each element of matrix C is set to one.
MAT C = ZER	Each element of matrix C is set to zero.
MAT C = IDN	The diagonal elements of matrix C are set to one's yielding an identity matrix.

**MAT INPUT V**

The input of a vector is called in.

Special rules apply to the dimensioning of matrices which occur in MAT instructions. DIM statements indicate what the maximum dimension of a matrix is to be. Thus if we write

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

then M may have up to 20 rows and up to 35 columns. If a matrix reference occurs without a DIM statement, a ten-by-ten matrix is established.

The actual dimension of a matrix may be determined either when it is first set up, or when it is computed. For example,

```
MAT READ M
```

reads a matrix M of the dimension previously declared in a DIM statement. On the other hand,

```
MAT READ M(17,3)
```

reads a 17-by-3 matrix providing sufficient space has been saved for it.

Four of the MAT statements explicitly accomplish redimensioning:

```
MAT READ C (M, N)
MAT C = ZER (M, N)
MAT C = CØN (M, N)
MAT C = IDN (N)
```

The first three statements would specify matrix C as consisting of M rows and N columns. The fourth statement would specify matrix C as a square matrix of N rows and N columns. These same instructions may be used to redimension a matrix during running. A matrix may be redimensioned to either a larger or a smaller matrix provided the new dimensions do not require more space than was originally reserved by the DIM statement. To illustrate, assume the following statements exist:

```
10 DIM A (8,8), B(8,8), C (8,8)
   .
   .
   .
50 MAT READ A (2, 2), B (2, 2)
60 MAT C = ZER (2, 2)
   .
   .
   .
100 MAT A = IDN (8, 8)
110 MAT READ B (4, 4), C (4, 4)
```

From these statements observe that the DIM statement reserves sufficient storage to accommodate three matrices each consisting of sixty-four (64) elements. The initial MAT READ specifies the dimensions of both matrices A and B as two rows and two columns. The MAT READ also reads the number of values required by the dimensions into the storage which was reserved by the DIM statement. The MAT READ reads the values in row-wise sequence. In the initial MAT READ, the elements in the order read are A(1, 1), A(1, 2), A(2, 1), A(2, 2), B(1, 1) B(1, 2), B(2, 1) and B(2, 2). (Matrix statements use 1 to n subscripting, not 0 to n.)

Statement 60 illustrates the use of ZER to specify dimensions and to zero the elements of the matrix C. Statements 100 and 110 illustrate redimensioning: matrix A is redimensioned as an eight row, eight column identity matrix, and matrices B and C are redimensioned as four row, four column matrices into which data is to be read.

While the combination of ordinary BASIC instructions and MAT instructions makes the

language much more powerful, you must be very careful about dimensions. In addition to having both a DIM statement and a declaration of current dimension, you must be careful with the MAT statements. For example, a matrix product  $\text{MAT } C = A * B$  may be illegal for one of two reasons: A and B may have dimensions such that the product is not defined, or even if it is defined, C may not have reserved enough space for the answer. In either case, a DIMENSION ERROR message results.

Matrices consisting of a single row or single column of elements (i.e., vectors) are permissible in MAT instructions. As is true with all other matrices, the dimensions for such matrices should be explicitly stated before use in a MAT instruction. Thus:

```
10 DIM A(3, 1), B(3, 3), C(3, 1)
20 MAT C = ZER(3, 1)
30 MAT READ A(3, 1), B(3, 3)
```

```

      .
      .
70 MAT C = B * A
```

illustrates the requirements for multiplying a (3x3) matrix by a (3x1) matrix (vector). Column vectors should always be considered as (nx1) matrices and row vectors must always be considered as (1 xn) matrices.

The same matrix may occur on both sides of a MAT equation in case of replacement, addition, subtraction, or constant multiplication; but not in any of the other instructions. Legal forms are:

```
MAT A = B
MAT A = A + B
MAT A = (2.5) * A
MAT A = A - B
```

Illegal forms are:

```
MAT A = B * A
MAT A = INV(A)
MAT A = TRN(A)
MAT A = A + B - C
```

The last example is an attempt to use more than one matrix operator in a MAT statement. Two MAT statements must be used to do two matrix operations.

The determinant of a matrix can be obtained by first inverting the matrix, and then using DET. For example:

```
MAT B = INV(A)
LET D = DET
```

The determinant of A is stored in D. You may then decide whether the determinant was large enough for the inverse to be meaningful.

Attempting to invert a singular matrix does not cause the program to stop, but DET is set equal to zero.

Two programs follow which illustrate some of the capabilities of the MAT instructions. In the first example, the values for M and N are read. Using these values as indexes, statement 30 sets the dimensions for matrices A, B, D, and G respectively. Also the values for the elements of these matrices are read. In sequence then, the dimensions of matrix C are specified and the elements set to zero (statement 40). Matrix A is printed (statement 60); matrix B is printed (statement 80). The sum of matrices A and B is found and stored in C (statement 90). Matrix C is printed (statement 110). The dimensions for matrix F (a vector) are set and the elements set to zero (statement 120). The product of matrices C and D is

computed and stored in F (statement 130). The dimensions for matrix H (single value) are specified and the elements set to zero (statement 140). Finally the product of matrices G and F is found and stored in H and printed (statements 150, 170).

In the second example, a value N is read which determines the order of the Hilbert Matrix segment to be computed, stored, and printed. Next this matrix is inverted and printed. Finally the Hilbert Matrix is multiplied by its own inverse and the resulting product matrix is printed. Notice that in the example N=2 then N=3 is run, demonstrating the ability to re-dimension larger during running.

BASICT 10:45

```

10 DIM A(5,5),B(5,5),C(5,5),D(5,5),E(5,5),F(5,5),G(5,5),H(5,5)
20 READ M,N
30 MAT READ A(M,M),B(M,M),D(M,N),G(N,M)
40 MAT C=ZER(M,M)
50 PRINT "MATRIX A OF ORDER "M
60 MAT PRINT A;
70 PRINT "MATRIX B OF ORDER "M
80 MAT PRINT B;
90 MAT C= A+B
100 PRINT " C=A+B "
110 MAT PRINT C;
120 MAT F=ZER(M,N)
130 MAT F= C*D
140 MAT H=ZER(N,N)
150 MAT H=G*F
160 PRINT " H "
170 MAT PRINT H;
1800 DATA 3,1
1810 DATA 1,2,3,4,5,6,7,8,9,9,8,7,6,5,4,3,2,1,1,2,3,3,2,1
1999 END

```

READY

RUN

BASICT 10:46

MATRIX A OF ORDER 3

```

1 2 3
4 5 6
7 8 9

```

MATRIX B OF ORDER 3

```

9 8 7
6 5 4
3 2 1
C=A+B

```

```

10 10 10
10 10 10
10 10 10
H

```

360

HILTST 11:01

```
10 DIM A(20,20),B(20,20),C(20,20)
20 READ N
30 MAT A =CØN(N,N)
40 MAT B =CØN(N,N)
45 MAT C =CØN(N,N)
50 FØR I=1 TØ N
60 FØR J=1 TØ N
70 LET A(I,J)=1/(I+J-1)
80 NEXT J
90 NEXT I
93 PRINT " HILBERT MATRIX ØF ØRDER "N
95 MAT PRINT A
100 MAT B=INV(A)
105 PRINT "INVERSE ØF HILBERT MATRIX ØF ØRDER "N
110 MAT PRINT B
115 MAT C=A*B
117 PRINT "HILBERT MATRIX TIMES ITS ØWN INVERSE ØRDER "N
118 MAT PRINT C
120 GØ TØ 20
190 DATA 2,3
1999 END
```

READY

RUN

HILTST 11:02

```
HILBERT MATRIX ØF ØRDER 2
1          0.5
0.5        0.333333
INVERSE ØF HILBERT MATRIX ØF ØRDER 2
4          -6
-6         12
HILBERT MATRIX TIMES ITS ØWN INVERSE ØRDER 2
1          0
7.45058 E-9 1.
HILBERT MATRIX ØF ØRDER 3
1          0.5          0.333333
0.5        0.333333    0.25
0.333333   0.25       0.2
INVERSE ØF HILBERT MATRIX ØF ØRDER 3
9.         -36.        30.
-36.       192.       -180.
30.        -180.      180.
HILBERT MATRIX TIMES ITS ØWN INVERSE ØRDER 3
1.         5.41409 E-7   -1.57456 E-6
-2.33452 E-7 1.         -7.69893 E-7
-1.92473 E-7 1.78814 E-7   0.999999
ØUT ØF DATA IN 20
```

The statement

MAT INPUT V

will call for the input of a vector. The number of components in the vector need not be specified. Normally the input is limited by having to be typed on one line. However if you end the line of input with & (before carriage return) the machine will ask for more input on the next line. Note that, although the number of components need not be specified, if we wish to put in more than 10 numbers we must save sufficient space with a DIM statement. After the input NUM will equal the number of components and V(1), V(2), . . . , V(NUM) will be the numbers entered. This allows variable length input. For example

```

5 LET X = 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 put in, and uses this as a signal to stop. Thus, the user can stop by simply pushing "carriage return" on an input request.

## ALPHANUMERIC DATA AND STRING MANIPULATION

Alphanumeric data, names, and other identifying information can now be handled in the BASIC language using string variables. You can input, store, compare and output alphanumeric and certain special characters.

A STRING is any sequence of alphanumeric and certain special characters not used for control purposes in the system.

STRING SIZE is limited to 119 valid characters.

Any variable followed by a "\$" represents a string. For example: A\$, B\$. A subscripted string variable refers to a particular string in a list of strings. For instance, B\$(4) would refer to the fourth string in the B\$ list.

Let's consider the BASIC statements where strings can be used.

### DIM Statement

Strings can be set up as one-dimensional lists only. Requests for two-dimensional lists are not allowed.

#### Examples:

```

10 DIM A(5), C$(20), A$(12), D(10,5)
20 DIM R$(35)
30 DIM M$(15), B$(15)

```

In statement 10, only C\$ and A\$ are string variables. R\$, as dimensioned in statement 20, will set aside space in memory for 35, 119 character lists. Any or all of these strings may be less than 119 characters; then only the space in memory required for the strings will be allocated.

### LET Statement

Strings and string variables may appear in only two forms of the LET statement. The first is used to replace a string variable with the contents of another string variable:

Example:

```
56 LET G$ = H$
```

and the second is used to assign a string to a string variable:

Example:

```
60 LET J$ = "THIS STRING"
```

Arithmetic operations may not be performed on string variables. Requests for addition, subtraction, multiplication, or division involving string variables produce an error message.

## IF—THEN Statement

Only one string variable is allowed on each side of the IF-THEN relation. All of the six standard relations (=, <>, <, >, <=, >=) are valid. When strings of different lengths are compared, the shorter string is filled with blanks so that it is the same length as the other string, then the comparison is made. For the collating sequence, refer to the BASIC Code Table, page 43.

Examples:

```
100 IF N$ = "SMITH" THEN 105
200 IF A$ <> B$ THEN 205
300 IF "JUNE" <= M$ THEN 305
400 IF D$ >= "FRIDAY" THEN 600
```

## CHANGE Statement

The CHANGE statement is used to convert upto 60 string characters into numerical "code" characters or the reverse. (See BASIC Code Table.)

In the following example the instruction CHANGE A\$ TO A in line 30 has caused the vector A to have as its zero component the number of characters in the string A\$ and to have code numbers in the other components.

```
10 DIM A(65)
20 READ A$
30 CHANGE A$ TO A
40 FOR I=0 TO A(0)
50 PRINT A(I);
60 NEXT I
70 DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
80 END
```

READY

RUN

CHANGE 11:38

```
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
```

To reverse the process and convert code characters to string characters, you must specify a value for the zero component. The value should be equal to the number of stored code characters you want to convert.

The following example reverses the process of the previous program. Note that the zero component is given the value 26 by line 50.

# ALPHANUMERIC DATA AND STRING MANIPULATION

```

10 DIM A(26)
20 FOR I = 1 TO 26
30 LET A(I) = 64+I
40 NEXT I
50 LET A(0) = 26
60 CHANGE A TO AS
70 PRINT AS
99 END

```

The above program will produce the string

ABCDEFGHIJKLMNØPQRSTUVWXYZ

BASIC Code Table

Characters	BASIC Code No.	Characters	BASIC Code No.
(blank)	32	@	64
!	33	A	65
“	34	B	66
#	35	C	67
\$	36	D	68
%	37	E	69
&	38	F	70
'	39	G	71
(	40	H	72
)	41	I	73
*	42	J	74
+	43	K	75
,	44	L	76
-	45	M	77
.	46	N	78
/	47	O	79
0	48	P	80
1	49	Q	81
2	50	R	82
3	51	S	83
4	52	T	84
5	53	U	85
6	54	V	86
7	55	W	87
8	56	X	88
9	57	Y	89
:	58	Z	90
;	59	[	91
<	60	\	92
=	61	]	93
>	62	↑	94
?	63		

Additional symbols useful on output are:

- ← (backward arrow) 95
- EOT (end of transmission) 4
- BELL (rings bell in teletype) 7
- LF (line feed) 10
- CR (carriage return) 13
- RUB - OUT (tape use only) 127

- Notes: (1) This is not a complete list – there are 128 characters numbered 0 through 127. Some of these numbers duplicate the above (on some teletypes) and some are just spaces.
- (2) The EOT character will hang up the phone if it is sent to a Model 33 Teletype.

## DATA, INPUT, and MAT INPUT Statements

In the DATA statements numbers and strings may be intermixed. Numbers will be 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 a string does not start with a letter, or contains a comma, it must be enclosed in quotes. For example:

```
90 DATA 10, ABC, 5, "4FG", "SEPT. 22, 1967", 2
```

The only convention on INPUT is that a string containing a comma must be enclosed in quotes.

With a MAT INPUT a string containing a comma or an ampersand must be enclosed in quotes. For example:

```
"MR. & MRS. SMITH", MR. JONES
```

is the correct format for a response to a MAT INPUT A\$, B\$.

In any of the three ways of getting string information into a program – DATA, INPUT, or MAT INPUT – leading blanks are ignored unless the string, including the blanks, is enclosed in quotes.

If in doubt use quotes; they will not cause any errors.

## READ and PRINT Statements

Strings may be read and printed in the usual manner. For example,

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

will print the word "TIME-SHARING." Note that the effect of ';' in the print statement is consistent with that discussed in the section on PRINT; that is, with alphanumeric output the semi-colon causes close packing whether that output is in quotes or is a string variable. (In contrast, recall that with numeric output the semi-colon causes space to be left between the numbers printed.) Commas and TAB's may be used as in any other PRINT statement. The loop

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

will read a list of 12 strings. In place of the READ and PRINT, the corresponding MAT instructions may be used for lists. For example, MAT PRINT M\$; will cause the elements of the list to be printed without spaces between them.

As usual, lists are assumed to have no more than 10 elements; otherwise a DIM statement is required.

Note: Numeric and string data are kept in two separate blocks, which act independently of each other. The command RESTORE will restore both numeric and string data. RESTORE\* will restore only the numeric data. RESTORE\$ will restore only the string data.

## DATA FILES

Data files may be used by BASIC language programs. Files may be created from the teletypewriter, saved, listed, and edited. Under program control, files may be read or written

for further use by the same or other programs. The end of data condition can be tested, and file rewinding, appending, backspacing, and other capabilities are provided.

## Initial File Preparation

A data file must be prepared in advance and saved in your catalog before being used in a program.

If a file with initial values is to be read, you must prepare it before program execution and save it.

Example (user-supplied data is underline):

```
NEW
NEW FILE NAME-- RFILE
READY
10 1, 2, 3, 4, 5, 6, 7
20 8, 9, 10, 11, 12, 13, 14
SAVE
READY
```

Note that the word DATA is not needed in these files. The first number on each line is the line number. The comma is the standard data delimiter. You can specify a non-standard delimiter, as described on page 57.

If a file is to be written during program execution, you must have previously saved it in your catalog. At the minimum, the file must contain one line number followed by a space. In Mark II BASIC you do not have to preset the size of the file. File size is limited to 50400 characters of source program.

## File Reference

Before the first use of any files in a program, their names must be specified in a file reference statement. The form of the file reference statement is

```
FILES name 1; name 2; ... ; name n
```

or

```
FILES name 1, password; ... ; name n, password
```

You can reference up to eight files in any program. If you reference more than eight files, the error message TOO MANY FILES will be printed out.

Multiple FILES statements are permitted, as long as the total number of files referenced does not exceed eight.

File naming must conform to the conventions for naming programs except:

- File names must not contain semicolons. Semicolons are interpreted as file separators.
- Leading spaces are ignored.
- File names should not contain slashes, “/”, or commas, “,”.

An asterisk may be used in place of a file name in the FILES statement, so that the file may be designated at a later time (see FILE Statement, page 56). The file used to replace the \*-designated file must already be saved in the same catalog.

Example:

```
10 FILES A;*;C;DATA
20 READ #4,A,B,C
30 FILE #2,"B"
40 SCRATCH #2
50 WRITE #2,A;B;C
99 END
```

The \*-designated file must be replaced with a saved file before any operations can be done on it.

File names may not appear twice in the same file reference statement.

## File Designator

The file designator is a numeric argument used in all file input and output statements. It selects the file to be used for the current operation.

The file designator is preceded by a number sign, "#." It may be an integer, variable, subscripted variable, or arithmetic expression.

Example:

```
10 FILES A;B;C;D
   :
   :
20 READ #1,X
   :
   :
30 READ #F,Y
   :
   :
40 READ #H(I),Z
   :
   :
50 READ #M*N,T
```

In statements 30, 40, and 50 above, the value of the designator specifies the file to be used. For example, if the value of F in statement 30 were 2, a data item would be read from file B.

If the value of a variable, subscripted variable, or arithmetic expression used as a file designator is a non-integer, the truncated value is used as the designator.

## File Modes

All files to be processed by BASIC must be considered as being in either read mode or write mode. A file in the read mode cannot be written. A file in the write mode cannot be read. Initially, the FILES statement results in all files being set to read mode. Before you can write to a file, you must place it in the write mode by using a SCRATCH statement (see SCRATCH Statement, page 55). To change a file from write mode to read mode, you must use a RESTORE statement (see RESTORE Statement, page 55).

## File Read

The form of the read file statement is

```
READ # file designator, input list
```

or

```
READ # file designator: input list
```

where the file designator is as described previously.

The input list consists of the variables, separated by commas, into which the data in the file is to be read. The list may contain non-string and string variables, and any of them may be subscripted.

The following example shows the use of three files, RFILE, DATA, and STRING, by a program, READ.

Example (user-supplied data is underlined):

```
OLD
OLD FILE NAME--RFILE
READY
LIST
RFILE          18:56
10 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5
```

READY

```
OLD
OLD FILE NAME--DATA
READY
LIST
DATA           18:56
10 1, 2, 3, 4, 5, 6, 7
20 8, 9, 10, 11, 12, 13, 14
```

READY

```
OLD
OLD FILE NAME--STRING
READY
LIST
STRING        18:57
10 ABC, DEF, GHI, JKL, MN0, PQR, STU, VWX, YZ
```

READY

```
OLD
OLD FILE NAME--READ
READY
LIST
READ          19:01
100 FILES RFILE, DATA, STRING
110 FOR I = 1 TO 7
120 READ #1, A(I), B(I)
130 PRINT A(I), B(I)
140 NEXT I
150 PRINT
160 PRINT
170 READ #2, T, U, V, X, Y, Z
180 PRINT T; U; V; X; Y; Z
190 PRINT
200 PRINT
210 FOR X = 1 TO 9
220 READ #3, C$(X)
230 PRINT C$(X)
240 NEXT X
999 END
```

READY

RUN

READ 19:02

1	1.5
2	2.5
3	3.5
4	4.5
5	5.5
6	6.5
7	7.5

1 2 3 4 5 6

ABC  
DEF  
GHI  
JKL  
MNØ  
PQR  
STU  
VWX  
YZ

For each execution of the READ statement, one value is read into the variable specified in the input list. If the entire file has not been read, the file pointer will remain positioned following the last read data item until additional statements designating that file are executed. For instance, in the example, if in the program READ you added a statement:

```
175 READ #2, R, S, T
```

R, S, and T would have the respective values 7, 8, and 9 assigned to them from the file DATA. Then the file DATA would have the pointer positioned at 10, the next data item in the file. Notice that the line number is not part of the data read by a READ statement.

## Reading Internal Data

Zero will be accepted as a file designator in the READ statement. A READ statement with zero as a file designator refers to data contained inside the program in a DATA statement.

Example:

```
10 READ #0, A, B, C
20 DATA 1, 2, 3
30 PRINT A; B; C
99 END
```

RUN

ZERØ 19:05

1 2 3

There is an important difference between reading internally with a READ #0 statement to read DATA, and using a READ # statement with a file designator other than zero to read from an external data file. When reading internally, string and non-string variables in the input list need not have the same order as string and non-string items in the DATA statement.

Example:

```
10 READ #0, A, G$, B, H$, C
20 DATA 1, ABC, 2, DEF, 3
or
10 READ #0, A, B, C, G$, H$
20 DATA 1, ABC, 2, DEF, 3
```

In both cases in the above example, the data items will be read correctly, with A, B, and C having values of 1, 2, and 3, and G\$ and H\$ having values of ABC and DEF.

But when reading an external file there must be a one-to-one correspondence between string and non-string items in the file and in the input list. Otherwise an INCORRECT FORMAT error message will be printed out.

Example (user-supplied data is underlined):

```
MIX          08:21
10 1,ABC,2,DEF,3

READY

NEW
NEW FILE NAME-- READMIX
READY
10 FILES MIX
20 READ #1,A,G$,B,H$,C
30 PRINT A;G$;B;H$;C
99 END

RUN

READMIX     08:20
1 ABC 2 DEF 3
```

The file MIX is correctly read. But if you change line 20 to put all the non-string variables in the input list (A, B, and C) before the string variables (G\$ and H\$), the file cannot be read.

Example:

```
10 FILES MIX
20 READ #1,A,B,C,G$,H$
30 PRINT A;G$;B;H$;C
99 END

RUN

READMX     08:22
INCORRECT FORMAT IN FILE MIX          IN 20
```

## File Write

The form of the write file statement is

WRITE # file designator, output list

or

WRITE # file designator: output list

where the file designator is as described previously.

The output list consists of variables, arithmetic expressions, or strings, separated by commas or semicolons, that indicate the data from which the file is to be generated. The variables may be either numeric or string, and any of them may be subscripted.

Example:

```
25 WRITE #1, B1, 4.5*(C/D), "STRING1", GS(6)
```

The WRITE # statement generates one line of output unless the teletypewriter line limit is exceeded or the last output list item is followed by a comma or a semicolon. When the line limit is exceeded, writing will continue on the next line with the next item of data. When the last item in the output list is followed by a comma or a semicolon, subsequent writing occurs on the same line if space is available. This arrangement permits listing the file on the teletypewriter.

The WRITE # statement generates a file beginning with line number 100, and increments by 10 for each additional line. The standard field delimiter, the comma, is used.

The format conventions of the regular PRINT statement apply to the WRITE # statement. The comma and semicolon, used to separate data items in the output list, cause the data to be written in regular or close-packed format. The TAB function can be used. But in counting for tabbing the line number is not included.

Example (user-supplied data is underlined):

Note--Files DATA and STRING are as shown on page 47.

```
10 FILES DATA, STRING, A
20 READ #1, A, B, C, D, E, F, G
30 READ #2, T$, US, VS, WS, XS, Y$, Z$
40 SCRATCH #3
50 WRITE #3, A, B, C, D, E, F, G
60 WRITE #3, A; B; C; D; E; F; G
70 WRITE #3, T$, US, VS, WS; XS; Y$; VS
99 END
```

RUN

XAMPLE 08:44

USED .54 UNITS

OLD

OLD FILE NAME--A

READY

LIST

A 08:46

```
100 1 ,          2 ,          3 ,          4 ,          5 ,
110 6 ,          7 ,
120 1 , 2 , 3 , 4 , 5 , 6 , 7 ,
130 ABC,          DEF,          GHI,          JKL, MN0, PQR, GHI,
```

Example using TAB:

```
10 FILES Z
20 READ #1, A, B, C
30 PRINT TAB(2); A; TAB(15); B; TAB(30); C
99 END
```

RUN

Z READ 15:11

```
1          2          3
```

A zero file designator used with a WRITE# statement will be accepted and cause the file to be written to the teletypewriter. In this case no SCRATCH statement is required, and no line numbers are supplied.

Example (user-supplied data is underlined):

```
10 FILES DATA;A
20 READ #1,A,B,C,D,E,F,G
30 WRITE #0,A;B;C;D;E;F;G
99 END
```

RUN

```
ZERODES      15:15      12/06/68
1 , 2 , 3 , 4 , 5 , 6 , 7 ,
```

## Reading with INPUT Statement

An alternative method for reading data from a file is provided by the INPUT statement. The form of the INPUT statement is

```
INPUT # file designator, input list
```

or

```
INPUT # file designator: input list
```

where the file designator and input list are as previously described.

The INPUT statement, unlike the READ statement, does not ignore the line numbers of a file. It treats line numbers as data items.

Example:

If file B contains

```
10 1,2,3,4,5
```

then the program

```
10 FILES B
20 INPUT #1,A,B,C,D,E
30 PRINT A;B;C;D;E
99 END
```

will give a printout

```
10,1,2,3,4,5
```

This happens because the line number is treated as part of the first data item, 1. If B contained

```
10 1 2 3 4
```

then the printout would be

```
10 1 2 3 4
```

with the line number being assigned to the variable name A. Thus when using the INPUT statement, you must include a delimiter immediately following each line number in the file. Otherwise the line number will be taken as part of the first data item on the line, with any imbedded spaces ignored. To avoid the inclusion of the line number in the first data item, INPUT statements should be used primarily with files that were written by the PRINT statement.

When using the INPUT statement to read data from a file with no line numbers or delimiters (see Writing with PRINT Statement, below), you must specify a blank delimiter. Otherwise the file will not be read correctly.

Example:

If file B contains

```
1 2 3 4 5
```

then the program

```
10 FILES B
15 DELIMIT #1,( )
20 INPUT #1,A,B,C,D,E
30 PRINT A;B;C;D;E
99 END
```

will produce

```
1 2 3 4 5
```

## Writing with PRINT Statement

A file may be written without line numbers or delimiters by using the PRINT statement. The form of the PRINT statement is

```
PRINT # file designator, output list
```

or

```
PRINT # file designator: output list
```

where the file designator and output list are as previously described.

The PRINT statement has the same result as the WRITE statement, except that no line numbers or delimiters are written.

Example (user-supplied data is underlined):

```
10 FILES DATA;B
20 READ #1,A,B,C,D,E,F,G
30 SCRATCH #2
40 PRINT #2,A;B;C;D;E;F;G
99 END
```

RUN

```
PFILE          15:35
```

```
USED           .36 UNITS
```

OLD

```
OLD FILE NAME--B
```

READY

LIST

```
B              15:36
```

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

In general, you cannot WRITE to a file to which you have been printing. But you can PRINT to a file to which you have been writing.

## MAT READ Statement

Data in matrix form may be read from a file with the MAT READ statement. The form of the statement is

```
MAT READ # file designator, input list
```

or

```
MAT READ # file designator: input list
```

where the file designator is as previously described and the input list contains matrix names.

The MAT READ statement will read from the designated file the matrices specified in the list. The file must have line numbers. Matrices in the list should have their dimensions specified, either in a DIM statement or in the MAT READ statement itself. When no dimensions are specified, a 10 x 10 matrix is assumed.

Data is read from the file in row-wise sequence. A zero file designator will cause the data to be read from an internal DATA statement.

Example:

```
10 FILES MATA;MATB
20 DIM C(3,3),D(5,7)
30 MAT READ #1,C,D
99 END
```

In this example, the matrices C and D, dimensioned 3 x 3 and 5 x 7, will be read from the file MATA. Since data is read in row-wise sequence, if MATA contains

1, 2, 3, ..., 44

then matrices C and D will contain

1	2	3	10	11	12	13	14	15	16
4	5	6	17	18	19	20	21	22	23
7	8	9	24	25	26	27	28	29	30
			31	32	33	34	35	36	37
			38	39	40	41	42	43	44

Here is an example of reading from an internal DATA statement.

Example:

```
10 DIM X(4,6),Y(3,5)
20 MAT READ #0,X,Y
25 MAT PRINT X;Y;
30 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22
40 DATA 23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41
99 END
```

In this example, matrix X will contain

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24

Here is an example where no dimensions are specified for the matrix.

Example:

```
10 FILES MATA
20 MAT READ #1,X,Y
30 MAT PRINT X;
40 PRINT
50 PRINT
60 MAT PRINT Y;
99 END
```

Since no dimensions are specified for X and Y, each is assumed to be a 10 x 10 matrix. If there are not enough data items in file MATA to complete a 10 x 10 matrix, matrix X will be filled out with zeroes. The second matrix specified, Y, will then also be filled with zeroes. If MATA contains

1,2,3,4,5,6,7,8,9

then the above program will read X as

```
 1  2  3  4  5  6  7  8  9  0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

and will read Y as

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

The dimensions can be specified in the MAT READ statement rather than in a DIM statement.

Example:

```
10 FILES DATA
20 MAT READ #1,X(3,3)
30 MAT PRINT X;
99 END
```

Here, X will be read as a 3 x 3 matrix.

## MAT WRITE Statement

Data in matrix form may be written to a file with the MAT WRITE statement. The form of the statement is

MAT WRITE # file designator, output list

or

MAT WRITE # file designator: output list

where the file designator is as previously described and the output list contains matrix names.

The MAT WRITE statement will write the matrices specified in the output list to the designated file. You cannot specify the dimensions of the matrices in the MAT WRITE statement.

Data is written to the file in row-wise sequence, and may be either packed or unpacked. A zero file designator will be accepted and will refer to the teletypewriter.

In the following example, the matrices X, Y, and Z will be read from file A, and then written in matrix form to files B and C. Matrices Y and Z will be tightly packed.

Example:

```
10 FILES A,B,C
20 DIM X(3,3),Y(4,6),Z(5,7)
30 MAT READ #1,X,Y,Z
40 SCRATCH #2
50 SCRATCH #3
60 MAT WRITE #2,X,Y;
70 MAT WRITE #3,Z;
99 END
```

## RESTORE Statement

The RESTORE statement causes the data pointer for the designated file to be repositioned at the beginning of the file. The form of the statement is

RESTORE # file designator

where the file designator is as previously described.

In addition to repositioning the data pointer, the RESTORE statement resets the file to the read mode. This makes it possible to read a file that has been previously written by the same program, or to read the same data more than once during a program run.

In the following example, line 40 will read the same values from the file DATA that were read by line 20. Line 70 sets file B to the read mode and the data pointer at the beginning of the file.

Example:

```
10 FILES DATA,B,C,D
20 READ #1,X,Y,Z
30 RESTORE #1
40 READ #1,M,N,O
50 WRITE #0,M,N,O,X,Y,Z
99 END
```

RUN

RES 16:06

```
1 , 2 , 3 , 1 , 2 ,
3 ,
```

## SCRATCH Statement

Files specified in a program are initially opened in the read mode. Before you can write to a file, it must be placed in the write mode. This can be done with the SCRATCH statement. The form of the statement is

SCRATCH # file designator

where file designator is as previously described. When a file is scratched, all data that was contained in the file before is lost.

In the following example, line 20 places file B in the write mode and positions the data pointer at the beginning of the file.

Example:

```
10 FILES A;B;C
   :
   :
20 SCRATCH #2
   :
   :
99 END
```

## FILE Statement

The identification of a file by a particular file designator may be changed within a program by use of the FILE statement. The form of the statement is

FILE # file designator, "file name"

or

FILE # file designator: "file name"

or, if the named file has a password,

FILE # file designator, "file name, password"

or

FILE # file designator: "file name, password"

The following example illustrates use of the FILE statement.

Example:

```
10 FILES DATA;A;B
20 READ #1, X, Y, Z
30 READ #3, Q, R, S
40 FILE #3, "STRING"
   :
   :
60 READ #3, T, U, V
99 END
```

In the example above, the file specified by the file designator 3 is changed from file B to the file STRING, using the FILE statement in line 40. Line 30 has already indicated that file B was to be read. Line 40 specifies that the file designator 3 will now indicate file STRING. Thus, in line 60, T, U, and V are assigned values from file STRING.

The file name used in a FILE statement may be a regular file name enclosed in quotation marks, as above; an asterisk enclosed in quotation marks; or a string variable, which may be subscripted.

Example:

```
10 FILES A;B
20 FILE #2, "*"
```

In this example the use of the "\*" closes file A, which means that file A cannot be used unless it is named in another file reference statement. The asterisk, when used as the file name, invalidates the associated file designator--2 in the example--until it is validated by a subsequent file statement.

The following example illustrates the use of a string variable as a file name in a FILE statement.

Example:

```
10 FILES A;B;C
20 LET X$ = "DATA"
30 FILE #2,X$
```

In this example X\$ is assigned the value of a file name, DATA. In line 30 file designator 2 is changed to specify the file DATA instead of the file B. The file DATA must already have been saved in the user's catalog. Note that in line 30 X\$ is not enclosed in quotation marks.

The file named in a FILE statement cannot appear in a file reference statement in the same program, unless the file has been released before its use in the FILE statement.

Example:

```
10 FILES DATA;A;B
20 FILE #2,"DATA"
```

This example will cause an error message to be printed out, because DATA appears in the file reference statement and is not released before its use in line 20. A file cannot be opened twice in succession without being released in the interval.

In the following example, line 20 replaces DATA with C, and therefore DATA can be used in the later FILE statement, line 30.

Example:

```
10 FILES DATA;A;B
20 FILE #1,"C"
30 FILE #2,"DATA"
```

## DELIMIT Statement

The standard delimiter used to separate items when reading from or writing to a file is the comma. Sometimes it may be useful to have a non-standard file delimiter. Non-standard delimiters are specified by the DELIMIT statement. The form of the statement is

DELIMIT # file designator, (character)

or

DELIMIT # file designator: (character)

or

DELIMIT # file designator, (abbreviation)

or

DELIMIT # file designator: (abbreviation)

where the file designator is as previously described, and the character is the non-standard delimiter to be used. The abbreviation is used for a non-printing character such as a line feed. Abbreviations for non-printing characters are those used in the USA Standard Code

for Information Interchange, and in common use: LF for line feed, CR for carriage return, etc.

Example (user-supplied data is underlined):

```
NEW
NEW FILE NAME-- DEL
READY
10 FILES DATA;B
20 READ #1,A,B,C,D,E,F
30 DELIMIT #2,(LF)
40 SCRATCH #2
50 WRITE #2,A;B;C;D;E;F
99 END
RUN
```

DEL 16:13

USED .40 UNITS

```
OLD
OLD FILE NAME--B
READY
LIST
```

B 16:14

```
100 1
      2
      3
      4
      5
      6
```

Whenever a file with a non-standard delimiter is to be read or written, the non-standard delimiter must be specified in a DELIMIT statement before the READ or WRITE statement. The PRINT statement will write the specified file with no delimiters or line numbers, regardless of whether non-standard delimiters or the comma are used.

Example:

DEL 16:17

10 1&2&3&4&5&6&7

READY

```
OLD
OLD FILE NAME--DELTEST
READY
LIST
```

DELTEST 16:18

```
10 FILES DEL;DATA
20 DELIMIT #1,(&)
30 READ #1,A,B,C,D,E,F
40 PRINT A;B;C;D;E;F
99 END
```

READY

RUN

DELTEST 16:18

```
1 2 3 4 5 6
```

In the above example, line 20 is required to specify the non-standard delimiter used in file DEL. If it were not present, an INCORRECT FORMAT message would be printed out upon the attempt to execute the READ statement in line 30.

A zero file designator will be accepted and interpreted to refer to the teletypewriter.

Example (user-supplied data is underlined):

```

NEW
NEW FILE NAME-- DEL2
READY
10 FILES DATA
20 READ #1,A,B,C,D,E,F,G
30 DELIMIT #0,( & )
40 WRITE #0,A;B;C;D;E;F;G
99 END
RUN

DEL2          16:20

  1 & 2 & 3 & 4 & 5 & 6 & 7 &

```

## APPEND Statement

Data may be added to a file with the APPEND statement. The form of the statement is

APPEND # file designator

where the file designator is as previously described.

The APPEND statement causes the data pointer for the designated file to be located after the last item of data in the file, and sets the file to the write mode. This allows you to release, recall, and write to files without losing previously stored data.

In the following example, X, Y, and Z\$ are written to file B following any data already in the file.

Example:

```

10 FILES A;B;C
20 READ #1,X,Y,Z$
30 APPEND #2
40 WRITE #2,X,Y,Z$
99 END

```

## MARGIN Statement

The MARGIN statement enables you to specify the rightmost character position for a designated file. The form of the statement is

MARGIN # file designator, expression

or

MARGIN # file designator: expression

where the file designator is as previously described, and the expression is evaluated to determine the value at which the right margin is to be set.

The following program will write file A with the right margin set at character position 25.

Example:

```
10 FILES A
20 SCRATCH #1
30 MARGIN #1,25
40 FOR X = 1 TO 60
50 WRITE #1,X;
60 NEXT X
99 END
```

If line 30 in the above example were

```
30 MARGIN #1,C*D
```

then the value of the expression C\*D would determine the right margin for file A. The integer part of the value is taken. If in the example the value of C\*D is 28.365, the margin in file A will be set to 28.

Zero will be accepted as a file designator, and will indicate the teletypewriter. For files other than the teletypewriter, the maximum margin size cannot exceed 118. A margin of 118 will be set if a greater value is used. A margin size of 0 will generate a margin of 118 for files other than the teletypewriter. For the teletypewriter, in this case the margin is assumed to be infinite.

## IF END Statement

The IF END statement enables you to test, when reading a file, for the end of data, or, when writing a file, for the end of file space. The form of the statement is

```
IF END # file designator THEN line number
```

or

```
IF END # file designator: THEN line number
```

or

```
IF END # file designator, THEN line number
```

When reading a file, the IF END statement tests the designated file to determine whether any valid data items remain. If not, the indicated path is taken.

Example:

If file B contains

```
10 1,2,3,4,5,6,7
```

then the program

```
10 FILES B
20 READ #1,A
30 PRINT A;
40 IF END #1 THEN 60
50 GO TO 20
60 PRINT "OUT OF DATA IN NUMBER ONE"
99 END
```

will produce the result

```
1 2 3 4 5 6 7 0 OUT OF DATA IN NUMBER ONE
```

In the above example, after the eighth time the READ statement in line 20 is executed, the IF END statement finds that no valid data remains in file B, and the indicated path to line 60 is taken.

Example:

If file B contains

```
10 1,2,3,4,5
```

then the program

```
10 FILES B
20 READ #1,A,B,C
30 PRINT A;B;C;
40 IF END #1 THEN 60
50 GO TO 20
60 PRINT "OUT OF DATA IN ONE"
99 END
```

will produce the result

```
1 2 3 4 5 0 OUT OF DATA IN ONE
```

In this example, the first READ in line 20 assigns variables A, B, and C the values of 1, 2, and 3. The IF END statement then finds that there is more data in the file, and line 50 is executed. The second READ in line 20 assigns the last two items in the file, 4 and 5, to the variables A and B, and assigns the value 0 to C. The IF END test then finds that there is no more data in the file, and the indicated path to line 60 is taken.

When writing to a file, you can use the IF END statement to test for the end of file space. If the end of file space is detected, the indicated path is taken.

Example:

The program

```
10 FILES D
20 SCRATCH #1
30 LET X = 0
40 LET X = X+1
50 WRITE #1,X;
60 IF END #1 THEN 80
70 GO TO 40
80 PRINT "IF END TEST INDICATES END OF FILE"
99 END
```

will repeatedly write the value of X to file D until the end of file space is detected by the IF END statement in line 60. Then the path to line 80 will be taken, and IF END TEST INDICATES END OF FILE will be printed out.

## IF MORE Statement

The IF MORE statement enables you to test, when reading a file, for more data, or, when writing a file, for more space. The form of the statement is

```
IF MORE # file designator THEN line number
```

or

```
IF MORE # file designator, THEN line number
```

or

```
IF MORE # file designator: THEN line number
```

When reading a file, the IF MORE statement can be used to test whether there is any more data in the file and to act upon the result of the test.

Example:

If file Y contains

```
1 0 1,2,3,4,5,6,7
```

then the program

```
10 FILES Y,B
20 READ #1,A,B,C
30 PRINT A;B;C
40 IF MORE #1 THEN 60
50 GO TO 80
60 PRINT "MORE DATA IN #1"
70 GO TO 20
80 PRINT "RAN OUT"
99 END
```

will give the results

```
1 2 3
MORE DATA IN #1
4 5 6
MORE DATA IN #1
7 0 0
RAN OUT
```

In the above example, the last data item in the file, 7, was read on the third execution of line 20. Then the variables B and C were assigned values of zero, because there was no more data after 7. The IF MORE statement then found no more data in the file and caused RAN OUT to be printed out. If lines 20 and 30 in the above example were:

```
20 READ #1,A
30 PRINT A
```

then the program would have produced the results:

```
1
MORE DATA IN #1
2
MORE DATA IN #1
3
MORE DATA IN #1
4
MORE DATA IN #1
5
MORE DATA IN #1
6
MORE DATA IN #1
7
RAN OUT
```

When writing to a file, the IF MORE statement can be used to test whether there is room to write any more to the file, and to act upon the result of the test.

Example:

The program

```
10 FILES B
20 SCRATCH #1
30 LET X = 1
40 LET X = X+1
```

```

50 WRITE #1,X
60 IF MORE #1 THEN 40
70 PRINT "NO MORE ROOM"
99 END

```

will print, after file B is completely filled

```
NO MORE ROOM
```

The program writes X to file B as long as the IF MORE statement in line 60 finds that more space remains in the file. When the file is full, the IF MORE detects the end of space and causes NO MORE ROOM to be printed out.

## BACKSPACE Statement

The BACKSPACE statement, when used while in the read mode, causes the data pointer to be stepped backward over one delimiter (and line number if present) to the previous data item. When used while in the input mode, the BACKSPACE statement causes the data pointer to be stepped backward to the beginning of the current line. Files being printed or written cannot be backspaced and then written to, because backspacing places the file in the read mode. The form of the statement is

```
BACKSPACE # file designator
```

You can use this statement to backspace to a particular data item or to the beginning of a file.

Example:

If the file DATA contains

```

10 1,2,3,4,5,6,7,8
20 9,10,11,12,13,14,15,16

```

then in the program

```

10 FILES DATA,B
20 READ #1,A,B,C,D
30 BACKSPACE #1
40 READ #1,E,F,G,H
99 END

```

line 20 will read the values 1, 2, 3, and 4 into variables A, B, C, and D, and the data pointer will be advanced to the next item, 5. Line 30 will then backspace the data pointer to the previous item, the number 4. Line 40 will then assign values of 4, 5, 6, and 7 to variables E, F, G, and H.

It is possible to backspace past the beginning of a file. When this happens, the first line in the file is used again and again.

Example:

```

10 FILES DATA
20 READ #1,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
30 FOR X = 1 TO 16
40 BACKSPACE #1
50 NEXT X
60 READ #1,R,S,T
99 END

```

In the above example, after line 20 the data pointer is positioned to indicate 16, the next item in the file. The loop in lines 30-50 backspaces 16 times, moving the data pointer past the beginning of the file. This causes the data pointer to begin backspacing from the end of the first line of data. In this case, the pointer will indicate 8, and line 60 assigns the values 8, 9, and

10 to R, S, and T. If the backspace loop had been greater, the first line of data in the file would have been repeatedly backspaced over and the data pointer would have been positioned at some item in the first line.

The following example illustrates backspacing of a file in the input mode.

Example:

```
10 FILES DATA
20 INPUT #1,A,B,C,D,E,F,G,H
30 BACKSPACE #1
40 INPUT #1,I,J,K,L,M,N,O,P
50 PRINT "FIRST SET OF VALUES"
60 PRINT A;B;C;D;E;F;G;H
70 PRINT
80 PRINT "SECOND SET OF VALUES"
90 PRINT I;J;K;L;M;N;O;P
99 END
```

RUN

```
X           16:59

FIRST SET OF VALUES
 101 2 3 4 5 6 7 8

SECOND SET OF VALUES
 101 2 3 4 5 6 7 8
```

Note that in the above example the line number is included in the first data item.

# Appendix A—Error Messages

---

Because most programs contain errors, a series of diagnostic messages is included in BASIC. Some of these messages occur during compilation and others during execution of a program. Many of the messages not only identify the type of error, but indicate the line number where the error occurred. We expect that as the development of the BASIC language continues these error messages will be revised.

During execution, certain messages occur which do not stop execution, but inform you of irregular conditions existing in identified lines of your program. Other messages, however, point out serious errors which stop execution.

## COMPILATION ERRORS

<u>MESSAGE</u>	<u>MEANING</u>
CUT PRØGRAM ØR DIMS	Either the program is too long, or the amount of space reserved by the DIM statements is too large, or a combination of these exists. This message can be eliminated by cutting the length of the program, reducing the size of the lists and tables, reducing the length of the lists and tables, reducing the length of printed labels, or reducing the number of simple variables.
DIMENSION TØØ LARGE AT (LINE #)	The size of a list or table is too large for the available storage at the line indicated.
END IS NØT LAST	Self-explanatory; it also occurs if there are two or more END statements in the program.
EXPRESSION TØØ CØMPlicated IN (LINE #)	Too many operations have been attempted in a single expression. Probably too many parentheses have been used. Use two or more simpler expressions instead.
FOR'S NESTED TØØ DEEPLY AT (LINE #)	Corresponding NEXT statement for preceding FOR statement must occur before another FOR statement can be used.
FØR WITHØUT NEXT IN (LINE #)	A NEXT statement is missing.
ILLEGAL CHARACTER IN (LINE #)	Use a valid character in place of an illegal character.
ILLEGAL CØNSTANT IN (LINE #)	More than nine digits or incorrect form in a constant number, or a number out of bounds (>1.70141E38).

<u>MESSAGE</u>	<u>MEANING</u>
ILLEGAL FØRMULA IN (LINE #)	This may indicate missing parentheses, illegal variable names, missing multiply signs, illegal numbers, or many other errors.
ILLEGAL INSTRUCTION IN (LINE #)	Other than one of the legal BASIC instructions has been used in the line indicated.
ILLEGAL LINE NUMBER AFTER (LINE #)	Line number is of incorrect form, or contains more than five digits.
ILLEGAL LINE REFERENCE IN (LINE #)	There is some character other than a number in a transfer statement (such as a GØ TØ) where the line number should be.
ILLEGAL MAT FUNCTION IN (LINE #)	A matrix function which is not possible has been attempted.
ILLEGAL MAT MULTIPLY IN (LINE #)	A matrix has not been multiplied correctly. MAT A = A*B is illegal.
ILLEGAL MAT TRANSPØSE IN (LINE #)	A matrix has not been transposed correctly. MAT A = TRN(A) is illegal.
ILLEGAL VARIABLE IN (LINE #)	An illegal variable name has been used.
INCØRRECT FØRMAT IN (LINE #)	The format of an instruction is wrong.
INCØRRECT NUMBER ØF ARGUMENTS IN (LINE #)	The number of arguments when defined must equal the number of arguments when referenced.
INCØRRECT NUMBER ØF SUBSCRIPTS IN (LINE #)	Indicates a matrix with one subscript or a vector with two.
MISMATCHED STRING ØPERATION IN (LINE #)	You have attempted to combine two strings algebraically, to compare a string and a number, or to assign a number to a string variable or vice versa.
NESTED DEF IN (LINE #)	Multiple-line DEF's cannot be nested.
NEXT WITHOUT FØR IN (LINE #)	A NEXT statement has been used without an accompanying FØR statement.
NØ END INSTRUCTION	The program has no END statement.
SYSTEM ERRØR IN (LINE #)	An error in BASIC; please report to your IPC.
TØØ MANY CØNSTANTS AT (LINE #)	There are too many constants. Put some in as DATA
TØØ MANY FILES AT (LINE #)	More than 8 files are specified in a FILES statement.
*UNDEFINED LINE NUMBER (LINE #) IN (LINE #)	The line number appearing in a GØTØ or IF-THEN statement does not appear as a line number in the program.
*UNDEFINED FUNCTION FN (LETTER) IN (LINE #)	A function such as FNF ( ) has been used without appearing in a DEF statement. Check for typographical errors.

\* These errors are not detected until run-time initialization.

<u>MESSAGE</u>	<u>MEANING</u>
UNFINISHED DEF	A multiple-line DEF has not been ended with FNEND.

## EXECUTION ERRORS

ABSOLUTE VALUE RAISED TO POWER IN (LINE #)	A computation of the form $(-3) \uparrow 2.7$ has been attempted. The system supplies $(ABS(-3)) \uparrow 2.7$ and continues. Note: $(-3) \uparrow 3$ is correctly computed to give -27.
CAN'T OPEN FILE (FILE NAME) IN (LINE #)	You have tried to access a file that doesn't exist.
CAN'T WRITE FILE (FILE NAME) IN (LINE #)	Out of space.
CHANGE ERROR IN (LINE #)	In converting numerical code characters into string characters, using the CHANGE statement, you have probably made an error in the character count. Check the zero element of the string.
DIMENSION ERROR IN (LINE #)	A dimension inconsistency has occurred in connection with a MAT statement in the indicated line. Execution stops.
DIVISION BY ZERO IN (LINE #)	A division by zero has been attempted. The system assumes the answer is $+\infty$ (about 1.70141E38) and continues running the program.
EXP TOO LARGE IN (LINE #)	The argument of an exponential function is $\geq 88.029$ . $+\infty$ (1.70141E38) is supplied for the value of the exponential and the running is continued.
ILLEGAL FILE COMMAND FOR (FILE NAME) IN (LINE #)	You have tried to write to a file that has not been set to the write mode, or read from a file that has not been set to the read mode.
ILLEGAL FILE NAME OR PASSWORD IN (LINE #)	File name or password violates rules for naming.
ILLEGAL 'VAL' ARGUMENT IN (LINE #)	You have used something other than a numeric argument in a VAL function.
INCORRECT FORMAT IN FILE (FILE NAME) (LINE #)	You have tried to read a string with a numeric variable.
INCORRECT FORMAT--RETYPE IT	Correct the input data.
INVALID FILE NUMBER IN (LINE #)	File number is less than 0 or greater than 8, or otherwise outside the range of acceptable file numbers, or the referenced file is not open.

<u>MESSAGE</u>	<u>MEANING</u>
LOG OF NEGATIVE NUMBER IN (LINE #)	The program has attempted to calculate the logarithm of a negative number. The system supplies the logarithm of the absolute value and continues.
LOG OF ZERO IN (LINE #)	The program has attempted to calculate the logarithm of 0. The system supplies $-\infty$ (about $-1.70141E38$ ) and continues running the program.
NOT ENOUGH INPUT--ADD MORE	Self-explanatory.
ON EVALUATED OUT OF RANGE IN (LINE #)	The integer part of the variable in the ON-GOTO statement is less than 1 or greater than the number of line numbers supplied by the statement.
OUT OF DATA IN (LINE #)	A READ statement for which there is no DATA has been encountered. This may mean a normal end of your program. Otherwise, it means you haven't supplied enough DATA. Execution stops.
OUT OF ROOM AT (LINE #)	The space reserved was not large enough. Try a dummy DIM statement, such as DIM A\$(1000).
OVERFLOW IN (LINE #)	A number larger than about $1.70141E38$ has been generated. The system supplies + (or -) $\infty$ (about $\pm 1.70141E38$ ) and continues running the program.
PROGRAM HALTED	S or STOP was typed when a numeric input was requested.
RETURN BEFORE GOSUB IN (LINE #)	This occurs if a RETURN is encountered before a GOSUB. (The GOSUB does not require a lower statement number, but must be performed before a RETURN.) Execution stops.
SQUARE ROOT OF NEGATIVE NUMBER IN (LINE #)	The program has attempted to extract the square root of a negative number. The system supplies the square root of the absolute value and continues running the program.
SUBSCRIPT ERROR IN (LINE #)	A subscript has been called for that lies outside the range specified in the DIM statement, or if no DIM statement applies, outside the range 0 through 10. Execution stops.
SYSTEM ERROR IN (LINE #)	An error in BASIC; please report to your IPC.
TOO MUCH INPUT--EXCESS IGNORED	Self-explanatory.
UNDERFLOW IN (LINE #)	A number in absolute size smaller than about $1.46937E-39$ has been generated. The system supplies 0 and continues running the program. In many circumstances underflow is permissible and may be ignored.

<u>MESSAGE</u>	<u>MEANING</u>
USELESS LOOP IN (LINE #)	Execution stops. Check the line indicated.
ZERO TO NEGATIVE POWER IN (LINE #)	A computation of the form $0 \uparrow (-1)$ has been attempted. The system supplies $+\infty$ (about $1.70141\text{E}38$ ) and continues running the program.

# Appendix B—Limitations on Basic

---

There are no pre-assigned limitations imposed on BASIC, but limitations do exist related to memory size. Certain combinations of BASIC language elements can generate a program which exceeds memory size limits. The list below indicates the extent of some of these limitations.

<u>ELEMENT</u>	<u>LIMITATION</u>
Length of program	Difficult to relate to the BASIC program, but in general limited to about 6 to 8 feet of teletypewriter paper filled with BASIC statements.
Constants	No definite limit.
Data	No definite limit.
GØ TØ and IF-THEN statements	No definite limit.
Lists and tables	No definite limit.
GØSUB and DEF statements	There may be no more than 100 nestings of GØSUB or DEF statements, nor may any combinations of GØSUB and DEF statements exceed 100 nestings. A combination of these statements could exist where a GØSUB statement called for a subroutine which contained additional GØSUB or DEF statements.
Length of file	50400 characters.





Computer Centers and offices of the Information Service Department are located in principal cities throughout the United States.

Check your local telephone directory for the address and telephone number of the office nearest you. Or write . . .

General Electric Company  
Information Service Department  
7735 Old Georgetown Road  
Bethesda, Maryland 20014

**GENERAL**  **ELECTRIC**  
**INFORMATION SERVICE DEPARTMENT**