KEN K

202026B

REFERENCE MANUAL

202026B

Information Systems

World Leader In Time-Sharing Service

MARK I

BASIC Language

GENERAL ELECTRIC

INFORMATION SERVICE DEPARTMENT

202026B

REFERENCE MANUAL

BASIC Language

Reprinted 2-70

GENERAL ELECTRIC

INFORMATION SERVICE DEPT.

Preface

This manual, which combines and supersedes two other manuals--BASIC Language Reference Manual (202026A) and BASIC Language Extensions Reference Manual (802207--describes the version of the BASIC language used with the General Electric Mark I Time-Sharing Service.

Another manual, Mark I Time-Sharing Service Command System Reference Manual (229116) explains all of the system commands that are a part of the Mark I Time-Sharing Service. It should be consulted for system information.

The original 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.

General Electric has continued to expand the capabilities of BASIC, adding such versatile features as string manipulation, data files, formatted line output, and others.

Contents

CONTENTS (CONT'D)

Introduction

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 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 form that the computer understands. 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 a combination of mathematical notation and English. If the computer is a French-speaking person, the program will be presented in French rather than English. And if the computer is a high-speed digital computer, the program must be presented in a programming language that the digital computer understands.

The second requirement for any program is that it 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 talking about programs that provide numerical answers to numerical problems. It would be easy to present a program in the English language, but such a program would pose insurmountable difficulties for the computer. English is rich with ambiguities and redundancies, the qualities that make poetry possible but computing impossible. Instead of using English, you present your program in a programming language that resembles ordinary mathematical notation, that has a simple vocabulary, and that permits a complete and precise specification of your program. The programming language you will use is BASIC, Beginner's All-purpose Symbolic Instruction Code. BASIC is precise, simple, and easy to understand.

An introduction to writing a BASIC program is given in Chapter 1, which includes all that you need to know to write a variety of useful and interesting programs. Chapter 2 deals with more advanced techniques. The Appendixes contain a variety of reference materials.

1. 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:

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

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

If ae − bd is not equal to zero, you should be able to solve this system to find that

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

If ae − bd is equal to zero, either there is 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 simply want you to understand the BASIC program for solving this system.

Study the following program carefully--the purpose of most lines in the program is self-evident--and then read the commentary and explanation.

```
100 READ A,B,D,E
110 LET G=A*E-B*D
120 IF G=0 THEN 180
130 READ C,F
140 LET X=(C*E-B*F)/G
150 LET Y=(A*F-C*D)/G
160 PRINT X,Y
170 GØ TØ 130
180 PRINT "NØ UNIQUE SØLUTIØN"
190 DATA 1,2,4
200 DATA 2,-7,5
210 DATA 1,3,4,-7
999 END
```

You can see, first, that this sample program uses only capital letters, because the teletypewriter has only capital letters.

Second, you can see that each line of the program begins with a line number. These numbers are called *line numbers;* they identify the lines, each of which is called a *statement*. A program is made up of statements, most of which are instructions to the computer. The 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 also facilitates correcting and changing programs, as we shall explain later on.

Third, note that each statement starts, after its line number, with an English word. The 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 we are now considering.

Note also that, although it is not obvious from the program, spaces have no significance in BASIC statements, except in messages that are to be printed out, as in line number 180. Spaces may be used or not, at will, to make a program more readable. Statement 100 could have been typed as 100READA, B, D, E and statement 110 as 110LETG=A*E-B*D.

With this preface, let's go through the program, step by step. The first statement, 100, 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 word READ to be given values according to the next available numbers in the DATA statements. In our sample program, we read A in statement 100 and assign the value 1 to it from statement 190, and similarly with B and 2, and with D and 4. At this point, we have exhausted the available data in statement 190, but there is more in statement 200. We pick up there the number 2 to be assigned to E.

Next we go to statement 110, a LET statement, where we first encounter a formula to be evaluated. (The asterisk, *, is used to denote multiplication.) In this statement we direct the computer to find 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 formula on the righthand side of the equal sign.

We know that if G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 120, whether G is equal to zero. If the computer finds a Yes answer to the question, it is directed to go to line 180. Line 180 tells it to print out NØ UNIQUE SØLUTIØN. From this point, it would go to the next statement. But lines 190, 200, and 210 give it no instructions, since DATA statements are not executed, and it then goes to line 999, which directs it to END the program.

If the answer to the question "Is G equal to zero?" is No, as it is in our sample program, the computer simply goes to the next statement, in this case statement 130. (An IF--THEN statement tells the computer where to go if the IF condition exists, but to go on to the next statement if it does not exist.) Statement 130 directs the computer to read the next two entries from the DATA statements--in this case -7 and 5, both in statement 200--and to assign them to C and F respectively. The computer is now ready to solve the system:

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

In statements 140 and 150, we direct the computer to find the values of x and y according to the formulas provided. Note that we must use parentheses to show that CE – BF is divided by G. Without parentheses, only BF would be divided by G, and the computer would find

$$X = CE - \frac{BF}{G}$$

The computer is told in line 160 to print the two values computed, those of X and Y. Having done so, it moves on to line 170, where it is directed back to line 130. If there are additional numbers in the DATA statements, as there are here in 210, the computer is told in line 130 to take the next number and assign it to C, and the one after that to F. The computer is now ready to solve the system:

$$x + 2y = 1 \qquad\qquad 4x + 2y = 3$$

As before, it finds the solution in 140 and 150, prints out the values in 160, and then is directed in 170 to go back to 130.

In line 130 the computer reads two more values, 4 and -7, which it finds in line 210. It then solves the system:

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

and prints out the solutions. It is directed back again to 130, but there are no more pairs of numbers available for C and F in the DATA statements. The computer therefore informs you that it is out of data, printing on the paper in the teletypewriter ØUT ØF DATA IN 130, and stops.

Let's look at the importance of the various statements. For example, what would have happened if we had omitted line number 160? The answer is simple. The computer would have solved the equations three times and then told us it was out of data. However, since it was not told to show us (PRINT) the answers, it would not do so, and the solutions would be the computer's secret.

What would have happened if we had left out line 120? In the problem just solved, nothing. But if G were equal to zero, we would have set the computer the impossible task of dividing by zero in 140 and 150, and it would tell us so by printing out DIVISIØN BY ZERØ IN 140 and DIVISIØN BY ZERØ IN 150. Suppose we had left out statement 170? The computer would have solved the first system, printed out the values of X and Y, and then gone on to line 180. As directed, it would print out NØ UNIQUE SØLUTIØN, and then stop.

A natural question that may arise is, why this selection of line numbers? The answer is that the particular choice of line numbers is arbitrary. The only requirement is that statements be numbered in the order that we want the computer to follow in executing the program. We could have numbered the statements 1, 2, 3, 4, ..., 13; but we do not recommend this numbering. We would normally number the statements 100, 110, 120, ..., 999. We put the numbers such a distance apart so that we can later insert additional statements easily. For example, if we find that we have left out two statements belonging between those numbered 140 and 150, we can give them any two numbers between 140 and 150, say 144 and 146. In the editing and sorting process, the computer will put them in the correct place.

Another question that may arise has to do with the placing of the elements of data in the DATA statements: Why place them as they are in the sample program? The choice is arbitrary. We need only arrange 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, and so on. In place of the three statements numbered 190, 200, 210, we could have put

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

or we could have written, perhaps more logically

```
        190 DATA 1,2,4,2
        200 DATA -7,5
        210 DATA 1,3
        220 DATA 4,-7
```

putting the coefficients in the first DATA statement and the three pairs of righthand constants in the following DATA statements.

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

The sample program and the resulting printout are shown now as they appear on the teletypewriter.

```
100 READ A,B,D,E
110 LET G=A*E-B*D
120 IF G=0 THEN 180
130 READ C,F
140 LET X=(C*E-B*F)/G
150 LET Y=(A*F-C*D)/G
160 PRINT X,Y
170 GØ TØ 130
```

```
180 PRINT "NØ UNIQUE SØLUTIØN"
190 DATA 1,2,4
200 DATA 2,-7,5
210 DATA 1,3,4,-7
999 END
RUN

LINEAR     14:37

 4               -5.5
 .666667          .166667
-3.66667         3.83333

ØUT ØF DATA  IN 130
```

FORMULAS

The computer can carry out a great many operations. It can add, subtract, multiply, divide, extract square roots, raise a number to a power, find the sine of a number on an angle measured in radians, and so on. We shall now learn how to tell the computer to carry out these various operations in the order that we want them done.

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

Symbol	Example	Meaning
+	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 ↑ 2	Raise to the power. Find X^2.

We must be careful with parentheses to make sure that we group together the things 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, then multiply this result by B, and then add A to the resulting product. This is the usual convention for $A + BC^D$. If this is not the intended order, we must use parentheses to indicate a different order. Suppose it is the product of B and C that we want raised to the power D. We must write A+(B*C)↑D. Or, if we want to multiply A + B by C to the power D, we write (A+B)*C↑D. We could add A to B, multiply their sum by C, and raise the product to the power D by writing ((A+B)*C)↑D.

The order of priorities for computing is according to the following rules.

1. The formula inside parentheses is computed before the enclosed quantity is used in further calculations.

2. In the absence of parentheses in a formula that includes addition, multiplication, and the raising of a number to a power, the computer first raises the number to the power, then does the multiplication, and does the addition last. Division has the same priority as multiplication, and subtraction the same as addition.

3. In the absence of parentheses in a formula that includes only multiplication and division (or only addition and subtraction), the computer works from left to right.

The rules are illustrated in the sample program already considered. The rules also tell us that the computer, given A-B-C, will subtract B from A and then C from their difference. Given A/B/C, it will divide A by B and then that quotient by C. Given A↑B↑C, the computer will raise the number A to the power B and then raise the resulting number to the power C. If there is ever any question in your mind about the priority, put in more parentheses to avoid possible ambiguities.

In addition to the five arithmetic operations, the computer can evaluate several mathematical functions. The functions are given special three-letter names, as shown in the following table.

Function	Meaning	
SIN(X)	Find the sine of X.	X interpreted as a number, or as an angle measured in radians.
CØS(X)	Find the cosine of X.	
TAN(X)	Find the tangent of X.	
ATN(X)	Find the arctangent of X.	
EXP(X)	Find e^x.	
LØG(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 ($\sqrt{X}$).	

Three other mathematical functions are available in BASIC: INT, RND, and SGN. These are reserved for explanation in Chapter 2.

In place of X, we may substitute any formula or number in parentheses following any of the functions listed above. For example, we may tell the computer to find $\sqrt{4 + X^3}$ by writing SQR(4+X↑3), or the arctangent of $3X - 2e^X + 8$ by writing ATN(3*X-2*EXP(X)+8).

Since we have mentioned numbers and variables, we should be sure 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 as many as 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   483.4156

The following are not numbers in BASIC:

14/3 $\sqrt{7}$ .00123456789

The first two are formulas, but not numbers. The last one has more than nine digits. We may tell the computer to find the decimal expansion 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 flexibility by use of the letter E, which stands for "times ten to the power." Using E, we can write .00123456789 in several forms acceptable to the computer: .123456789E-2 or 123456789E-11 or 1234.56789E-6. We can write ten million as 1E7 and 1969 as 1.969E3. We do not write a number as E7, but must write 1E7 to indicate that it is 1 that is multiplied by 10^7.

Variables

A variable in BASIC is denoted by any letter, or by any letter followed by a single digit. 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 is written. Variables are given or assigned values by LET, READ, and INPUT statements. All variables have the initial value of zero, so that if you want the starting value of a variable to be zero you need not assign it that value. (Another kind of variable, the string variable, is discussed later on in Chapter 2.)

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.

Three other mathematical symbols, symbols of relation, are available in BASIC to indicate any of six standard relations. These are used in IF--THEN statements, where values must be compared. The six possible relations are shown in the following table.

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

The use of loops can be illustrated 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:

```
100 PRINT 1,SQR(1)
105 PRINT 2,SQR(2)
110 PRINT 3,SQR(3)
        •
        •
        •
590 PRINT 99,SQR(99)
595 PRINT 100,SQR(100)
600 END
```

With the following program, using one type of loop, we can get the same table with only 5 lines of instruction instead of 101.

```
100 LET X=1
110 PRINT X,SQR(X)
120 LET X=X+1
130 IF X<=100 THEN 110
999 END
```

Statement 100 gives the value of 1 to X, which initializes the loop. Line 110 causes the printing of both 1 and its square root. Line 120 increases the value of X by 1, to 2. Line 130 asks whether X is less than or equal to 100--a Yes answer directs the computer back to line 110. Here it prints 2 and $\sqrt{2}$, and goes to 120. Again X is increased by 1, this time to 3, and at 130 it goes back to 110. This process is repeated--line 110 (print 3 and $\sqrt{3}$), line 120 (X = 4), line 130 (since 4 is less than or equal to 100 go back to line 110), and so on--until the loop has been traversed 100 times. Then X becomes 101. The computer now finds a No answer to the question in line 130 (X is greater than 100, not less than or equal to 100). It therefore does not return to 110 but moves on to line 999, and ends the program.

All loops contain four elements: initialization (line 100 in our program), the body (line 110), modification (line 120), and an exit test (line 130). Because loops are so important, BASIC provides a pair of statements that specify a loop even more simply than the previous program. They are the FØR and NEXT statements. Their use is illustrated in this program:

```
100 FØR X=1 TØ 100
110 PRINT X,SQR(X)
120 NEXT X
999 END
```

which does exactly the same thing as the two previous programs. In line 100, X is set equal to 1, and a test is set up, like that of line 130 above. Line 120 causes X to be increased by 1, and also carries out the test to decide whether to go back to line 110 or to go on. Thus lines 100 and 120 take place of lines 100, 120, and 130 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, we could specify it by writing, for example:

```
100 FØR X=1 TØ 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 twentieth time. Another step of 5 would take X beyond 100, so the program would go on to the end after printing 96 and its square root. The step value may be either positive or negative. We could obtain the first table printed in reverse order by writing line 100 as

```
100 FØR X=100 TØ 1 STEP -1
```

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

More complicated FØR 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

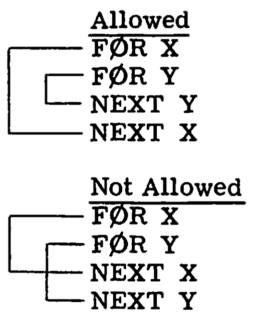
```
250 FØR X=N+7*Z TØ (Z-N)/3 STEP (N-4*Z)/10
```

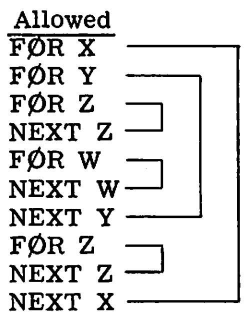
For a positive step size, the loop continues as long as the control variable is *less* than or equal to the final value. For a negative step size, the loop continues as long as the control variable is *greater* than or equal to the final value.

If the initial value is greater than the final value (or less than for a negative step size), then the body of the loop will not be done even once. 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.

```
100 READ N
110 FØR K=1 TØ N
120 LET S=S+K
130 NEXT K
140 PRINT S
150 GØ TØ 100
160 DATA 3,10,0
999 END
```

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





LISTS AND TABLES

In addition to the ordinary numeric variables used in BASIC, there are variables that we can use to designate the elements of a list or table. We use these where we would ordinarily use a subscript or a double subscript, as for the coefficients of a polynomial (a_0, a_1, a_2, ...) or the elements of a matrix ($b_{i,j}$). The variables that we use in BASIC consist of a single letter, which we call the name of the list or table, followed by the subscripts in parentheses. For the coefficients of the polynomial we would write A(0), A(1), A(2), and so on; for the elements of the matrix we would write B(1,1), B(1,2), and so on.

We can enter the list A(0), A(1), ... A(10) into a program very simply with four lines:

```
100 FØR I=0 TØ 10
110 READ A(I)
120 NEXT I
130 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. If we want larger subscripts, we must use a dimension (DIM) statement, to tell the computer to save 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:

```
100 DIM A(25)
110 READ N
120 FØR I=1 TØ N
130 READ A(I)
140 NEXT I
150 DATA 15
160 DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

Statements 110 and 150 could have been eliminated by writing 120 as FØR I=1 TØ 15, but the form we used allows us to lengthen the list by changing only statement 150, so long as the number of elements in the list does not exceed 25.

We would enter a 3 x 5 table into a program by writing:

```
100 FØR I=1 TØ 3
110 FØR J=1 TØ 5
120 READ B(I,J)
130 NEXT J
140 NEXT I
150 DATA 2,3,-5,-9,2
160 DATA 4,-7,3,4,-2
170 DATA 3,-3,5,7,8
```

We may enter a table with no dimension statement, and the computer will handle all the entries from B(0,0) to B(10,10). But 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 can be easily corrected by entering, for example, the line:

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

which will reserve space for a 20 by 30 table.

The single letter denoting a list or table name may also be used to denote a simple variable without confusion. But 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. You might have the list item B(I+K) or the table items B(I,K) or Q(A(3,7),B-C).

Let's look now at a sample program that uses both a list and a table. The program computes the total sales of each of five salesmen, each of whom sells the same three products.

```
SALES1      14:38

100 FØR I=1 TØ 3
110 READ P(I)
120 NEXT I
130 FØR I=1 TØ 3
140 FØR J=1 TØ 5
150 READ S(I,J)
160 NEXT J
170 NEXT I
180 FØR J=1 TØ 5
190 LET S=0
200 FØR I=1 TØ 3
210 LET S=S+P(I)*S(I,J)
220 NEXT I
230 PRINT "TØTAL SALES FØR SALESMAN ";J,"$";S
240 NEXT J
250 DATA 1.25,4.30,2.50
260 DATA 40,20,37,29,42
270 DATA 10,16,3,21,8
280 DATA 35,47,29,16,33
999 END

RUN

SALES1      14:39

TØTAL SALES FØR SALESMAN  1    $ 180.5
TØTAL SALES FØR SALESMAN  2    $ 211.3
TØTAL SALES FØR SALESMAN  3    $ 131.65
TØTAL SALES FØR SALESMAN  4    $ 166.55
TØTAL SALES FØR SALESMAN  5    $ 169.4
```

The list P gives the price per item of each of the three products. The table S tells how many items of each product each man sold. As you can see from the program, product number 1 sells for $1.25 per item, number 2 for $4.30 per item, and number 3 for $2.50 per item. You can see 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 100, 110, and 120, using data in line 250. It reads in the sales table in lines 130-170, using data in lines 260-280. The same program could be used again, modifying only line 250 if the prices change, and only lines 260-280 to enter the sales in another month.

The sample program did not need a dimension statement, since the computer automatically saves enough space to allow 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, though, to place DIM statements near the beginning of the program.

ERRORS AND DEBUGGING

Occasionally the first run of a new problem will be free of errors and give the correct answers. Usually, though, errors will have to be corrected before the program runs right. Errors are of two types: errors of form that prevent the running of the program, and logical errors in the program that cause the computer to produce either wrong answers or no answers at all.

Errors of form will cause error messages to be printed. The various error messages are listed and explained in Appendix A. Logical errors are often much harder to find, particularly when the program gives answers that are nearly correct. In either case, after you find the errors, you can correct them by changing lines, inserting new lines, or deleting lines from the program. 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. 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. For this reason, most programmers start out using line numbers that are multiples of five or ten, but that is a matter of choice.

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

As with most problems in computing, we can best illustrate the process of finding errors (or bugs) in a program, and correcting (or debugging) it, by an example. Let's consider the problem of finding the value of X between 0 and 3 for which the sine of X is a maximum, and printing 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 of X, but we shall use the computer to test successive values of X from 0 to 3. First we shall use intervals of .1, then of .01, and finally of .001.

Thus, we shall tell the computer to find the sine of 0, of .1, of .2, of .3, . . . , of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. It will do so 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 it will check 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 the computer finished the series, M will have the value of 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. Finally, 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 the three searches, we want the computer to print three numbers: the value X0 that has the largest sine, the sine of that number, and the interval of search.

Before going to the terminal, we write a program. Let's assume it is the following:

```
100 READ D
110 LET X0=0
120 FØR X=0 TO 3 STEP D
130 IF SIN(X) =M THEN 190
140 LET X0=X
150 LET M=SIN(X0)
160 PRINT X0,X,D
170 NEXT X0
180 GØ TØ 110
190 DATA .1,.01,.001
999 END
```

We shall illustrate the entire sequence on the teletypewriter, and make explanatory comments on the righthand side.

```
NEW
NEW FILE NAME--MAXSIN
READY.

100 READ D
110 LWR XO=0
120 FØR X=0 TØ 3 STEP D
130 IF SIN(X)<=M THEN 190
140 LET XO=X
150 LET M=SIN(X)
160 PRINT XØ,X,D
170 NEXT XO
180 GØ TØ 110
110 LET XO=0
190 DATA .1,.01,.001
999 END
RUN
WAIT.
```

After typing line 180, we notice that LET was mistyped in line 110, so we retype it, this time correctly.

```
MAXSIN    14:42


INCØRRECT  FØRMAT      IN  160
NEXT WITHØUT FØR       IN  170
FØR WITHØUT NEXT
UNDEFINED NUMBER     190
```

When we receive the first error message, we inspect line 160 and find that we used XØ instead of X0 for a variable. The next two error messages refer to lines 120 and 170, where we see that we mixed variables. We correct this by changing line 170. The fourth error message points out that line 130 directed the computer to a DATA statement and not to line 170 where it should go. We correct this by retyping line 130.

```
USED       0.83 UNITS.
160 PRINT XO,X,D
170 NEXT X
130 IF SIN(X)<=M THEN 170
RUN
```

```
MAXSIN    14:44

 .1               .1               .1
 .2               .2               .1
 .3               .3
STØP.
READY.

110 LET M=-1
RUN
WAIT.
```

This is obviously incorrect. Every value of X is being printed. We stop the printing by pressing the BREAK key. Then we ponder the program for a while, trying to figure out what's wrong with it.

We notice that SIN(0) is compared with M on the first time through the loop, but we had assigned no value to M. So we wonder if giving a value less than the maximum value of the sine will do it. We give it the value of -1, by changing line 110, where we had incorrectly initialized X0 instead of M.

```
MAXSIN    14:45

 0                0                .1
 .1               .1               .1
 .2               .2               .1
 .3               .3
STØP.
READY.

160
175 PRINT XO,M,D
RUN
WAIT.
```

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

We fix this by moving the PRINT statement outside the loop. We delete line 160, and line 175 is outside of the loop. We also realize that we want M printed and not X.

```
MAXSIN     14:46

 1.6              .999574          .1
 1.6              .999574          .1
 1.6              .999574          .1
 1.6
STØP.
READY.

180 GØ TØ 100
95 PRINT "X VALUE", "SINE", RESØLUTIØN"
RUN
WAIT.
```

We see that we are doing the same thing over and over again, the case for D=.1. So we stop the printing and inspect the program again.

Of course. Line 180 sent us back to to line 110 to repeat the operation rather than back to line 100 to pick up a new value for D. We also decide to put in headings for our columns by a PRINT statement.

```
MAXSIN     14:47    W1  WED 03/19/69


INCØRRECT  FØRMAT     IN   95
```

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

```
USED        1.00 UNITS.
95 PRINT "X VALUE", "SINE", "RESØLUTIØN"
RUN
```

We retype line 95 with all of the required quotation marks.

```
MAXSIN     14:48

X VALUE          SINE              RESØLUTIØN
 1.6              .999574           .1
 1.57             1.                .01
 1.571            1.                .001

ØUT ØF DATA   IN 100
```

Exactly the desired results. Of the 31 numbers 0, .1, .2, .3, ..., 2.8, 2.9, 3, it is 1.6 that has the largest sine, namely .999574. Similarly for the finer subdivisions.

```
USED       18.17 UNITS.
LIST


MAXSIN     14:49

95 PRINT "X VALUE", "SINE", "RESØLUTIØN"
100 READ D
110 LET M=-1
120 FØR X=0 TØ 3 STEP D
130 IF SIN(X)<=M THEN 170
140 LET XO=X
150 LET M=SIN(X)
170 NEXT X
175 PRINT XO,M,D
180 GØ TØ 100
190 DATA .1,.01,.001
999 END


SAVE


READY.
```

The whole process used only 18.33 computer resource units.

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

We save the program for later use. This should not be done unless we do expect to use the program later.

In solving this problem, there are two common devices we did not use. One is the insertion of a PRINT statement when we wonder whether the machine is computing what we think we asked it to compute. For example, if we wondered about M, we could have inserted 155 PRINT M, and we would have seen the values. The other device is used after several corrections have been made and you are not quite sure what the program now looks like. Simply type LIST or LISTNH, 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 concise description of each of the types of BASIC statements you will find most useful in writing the simpler kinds of BASIC programs. For each form, we shall assume a line number and use underlining to denote a general type. Thus, variable means any variable, which is a single letter, possibly followed by a single digit.

LET

The LET statement is not a statement of algebraic equality. It is an instruction to the computer to do certain computations and to assign the answer to a certain variable. Each LET statement is of the form

LET variable = formula

Examples:

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

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 computer puts all of the DATA statements, in the order in which they appear, into a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, with a READ statement still asking for more, the program is assumed to be done.

Since we have to read in data before we can work with it, READ statements normally are placed near the beginning of a program. The location of DATA statements is unimportant, so long as they are in the correct order. A common practice is to put all DATA statements together just before the END statement.

Each READ statement is of the form

READ sequence of variables

and each DATA statement is 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

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

Remember that only numbers are put in a DATA statement, and that 15/7 and $\sqrt{3}$ are formulas, not numbers.

INPUT

At times it is desirable to have data entered during running of a program. This is particularly true when one person writes the program and saves it in the computer's memory, and other persons are to supply the data. This may be done by using an INPUT statement, which is very much like a READ statement, but does not draw numbers from a DATA statement. Each INPUT statement is of the form

```
    INPUT sequence of variables
```

If, for example you want the user to supply values for X and Y in a program, you include the statement

```
    140 INPUT X,Y
```

before the first statement that is to use either of the two values. When it encounters the INPUT statement, the computer types a question mark on the printout and waits for input. The user types two numbers, separated by a comma, and presses the return key, and the computer goes on with the rest of the program.

Frequently an INPUT statement is accompanied by a PRINT statement, to make sure that the user knows what the question mark is asking for. If you include in your program

```
    120 PRINT "YØUR VALUES ØF X, Y, AND Z ARE";
    130 INPUT X,Y,Z
```

the computer will type out

```
    YØUR VALUES ØF X, Y, AND Z ARE?
```

Without the semicolon at the end of line 120, the question mark would have been printed on the next line.

Data entered by INPUT statement is not saved with the program. Also, it may take a long time to enter a large amount of data using INPUT. INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the program run, as it is with game playing programs.

PRINT

The PRINT statement has a number of different uses. It is discussed in more detail in Chapter 2. 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 do a combination of A and B
D. To skip a line

We have seen examples of only A and B 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 the value of X and then, a few spaces to the right, its square root. The second will print five different numbers: X, Y, B^2-4AC, and e^{A-B}. The computer will compute the two formulas and print the values for you, if 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 "NØ UNIQUE SØLUTIØN"
430 PRINT "X VALUE", "SINE", "RESØLUTIØN"
```

You have seen both in the sample programs. The first prints the statement within the quotation marks. 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 the program MAXSIN.

Examples of type C:

```
150 PRINT "THE VALUE ØF X IS";X
315 PRINT "THE SQUARE RØØT ØF";X;"IS";SQR(X)
```

If the first has computed the value of X as 3, it will print out

```
THE VALUE ØF X IS 3
```

If the second has computed the value of X as 625, it will print out

```
THE SQUARE RØØT ØF 625 IS 25
```

In statements of type C, the semicolon is used to minimize space.

Example of type D:

```
250 PRINT
```

The computer will advance the paper one line when it encounters this statement.

GØ TØ

There are times in a program when you do not want all statements executed in the order in which they appear in the program. An example of this occurs in the MAXSIN program where the computer has computed X0, M, and D and printed them out in line 160. We did not want the program to go on to the END statement yet, but we wanted it to go through the same process for a different value of D. So we directed the computer to go back to line 100 with a GØ TØ statement. Each GØ TØ statement is of the form

GØ TØ <u>line number</u>

Example:

```
150 GØ TØ 75
```

ØN—GØ TØ

The simple GØ TØ statement provides a single branched switch. The ØN--GØ TØ statement provides a multi-branched switch. The form of the statement is

ØN <u>expression</u> GØ TØ <u>line number</u>, <u>line number</u>, . . . , <u>line number</u>

The expression is any valid BASIC expression, and the line numbers are those to which the statement will transfer depending on the value of the expression.

Example:

```
230 ØN X+Y GØ TØ 575, 490, 150
```

This statement will transfer to line 575, 490, or 150 depending on whether the value of the expression X+Y is 1, 2, or 3.

The expression value will be truncated to an integer if it is not already an integer. For example, if X+Y equals 2.5, the value will be truncated to 2, and the program will branch to line 490, the second line number in the list.

Branching to a line containing a DIM, REM, or DATA statement is not allowed. As many line numbers may be included in an ØN---GØ TØ statement as will fit on one line.

IF—THEN

At times we want to jump the normal sequence of statements 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 130 of MAXSIN. Each IF--THEN statement is of the form

IF <u>formula</u> <u>relation</u> <u>formula</u> THEN <u>line number</u>

Examples:

```
340 IF SIN(X)<=M THEN 630
120 IF G=0 THEN 165
```

The first statement asks whether the sine of X is less than or equal to M, and tells the computer to go to line 630 if it is. The second statement asks if G is equal to zero, and tells the computer to go to line 165 if it is. In each case, if the answer to the question is No, the computer will go on to the next line of the program.

FØR and NEXT

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

FØR <u>variable</u> = <u>formula</u> TØ <u>formula</u> STEP <u>formula</u>

Most commonly, the formulas will be integers and the STEP will be omitted, which means that a step size of plus one is assumed. The accompanying NEXT statement is simple in form, but the variable must be exactly the same one as that following FØR in the FØR statement. The form of the NEXT statement is

NEXT <u>variable</u>

Examples:

```
130 FØR X=0 TØ 3 STEP D
180 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
270 NEXT X

456 FØR J=-3 TØ 12 STEP 2
470 NEXT J
```

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, the formulas are evaluated once and for all 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 150 FØR Z=2 TØ -2 without a negative step size, the loop will not be executed, and the computer will go immediately to the statement following the corresponding NEXT statement.

DIM

Whenever we want to enter a list or table with a subscript greater than 10, we must use a DIM statement to tell the computer to save enough room for the list or table.

Examples:

```
120 DIM H(35)
135 DIM Q(5,25)
```

The first statement would enable us to enter a list of 35 items--36 if we use H(0)-- and the second a table 5 x 25--or 6 x 26 if we use row 0 and column 0.

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:

```
99 END
```

2. Advanced BASIC

In Chapter 1, you learned how to write programs in BASIC. In this chapter we will discuss some capabilities of BASIC that were not discussed yet. These include:

- Alphanumeric data and string manipulation
- Files
- Matrices

We will also consider some advanced capabilities in printing output, several functions that we have not yet mentioned, and several statements either in more detail or for the first time. And, finally, we will consider two sample BASIC programs that make use of many of the advanced capabilities of BASIC.

ALPHANUMERIC DATA AND STRING MANIPULATION

Alphanumeric data, names, and other identifying information can be handled in BASIC using string variables. You can enter, store, compare, and print out alphanumeric and certain special characters in the Mark I character set.

A *string* is any sequence of alphanumeric and certain special characters in the Mark I character set not used for control purposes in the Mark I system. *String size* is limited to 15 valid characters.

A *string variable* is denoted by a letter followed by a dollar sign. For example, A$, B$, and X$ denote string variables.

DIM

Strings can be set up as one-dimensional arrays only. If you request a two-dimensional array you will receive the error message DIMENSIØN TØØ LARGE.

Examples:

```
100 DIM A(5),C$(20),A$(12),D(10,5)
200 DIM R$(35)
300 DIM M$(15),B$(15)
```

In line 100, only C$ and A$ are string variables. R$, as dimensioned in line 200, will save storage space for 35 fifteen character arrays. Any or all of the 35 strings may in fact be less than fifteen characters long.

LET

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:

```
156 LET G$=H$
```

The second is used to assign a string to a string variable.

Example:

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

Arithmetic operations may not be done on string variables. Requests for addition, subtraction, multiplication, or division involving string variables produce the error message ILLEGAL STRING ØPERATIØN AT XXX.

The LET statement permits multiple variable replacement.

Example:

```
262 LET X=Y=Z=21*N/2
435 LET A$=G$=J$="THIS STRING"
```

The first statement places the value of the expression 21*N/2 in variables X, Y, and Z. The second statement assigns the string THIS STRING to variables A$, G$, and J$. Any valid expression or string may be used.

READ and DATA

READ statements can contain string variables intermixed with ordinary variables. In the corresponding DATA statements, every item corresponding to a string variable in the READ statement must be a valid string. If the string contains any characters that have special meaning in BASIC--such as commas, semicolons, leading or trailing spaces, and so on--it must be enclosed in quotation marks. Unquoted strings must begin with an alphabetic character.

Example:

```
100 READ A$,B$,C$,D$,A,E$
200 DATA THE," ","PEØPLE,",YES--,500,ØF THEM.
300 PRINT A$;B$;C$;B$;D$;A;E$
999 END
```

This program will print out THE PEØPLE, YES-- 500 ØF THEM. The DATA statement has quotation marks around B$ because it is a blank space, and around C$ because it includes a comma.

INPUT

Like READ and DATA statements, INPUT statements can contain string variables intermixed with ordinary variables. Every item corresponding to a string variable in the INPUT statement must be a valid string variable. If the string contains characters that have special meaning in BASIC, it must be enclosed by quotation marks. If the string begins with other than an alphabetic character it must be enclosed in quotation marks.

Example:

```
110 INPUT L$(17),M$,N$(I)
```

PRINT

The PRINT statement also can contain string variables intermixed with ordinary variables. When a string variable is encountered that has not been assigned, the PRINT statement will produce for that variable a string of 15 blank spaces. A semicolon after a string variable in a PRINT statement causes the printout of the variable following that string to be directly connected to the string variable.

Examples:

```
135 PRINT A,16,B$,C$;N
140 PRINT 100+I, "DATA",L$;M$;N$
150 PRINT S$
```

IF—THEN

Only one string variable is allowed on each side of the IF-THEN relation sign. All of the six standard relations are valid (=,<>, <, >, <=, and >=). When strings of different lengths are compared, the shorter string and the corresponding part of the longer string will be used. If they compare, the shorter string is taken to be the lesser of the two.

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
```

You must use quotation marks around the string to be compared, as above, unless it is referenced in the IF--THEN statement by a string variable name.

Characters are compared in their BASIC code representations. The collating sequence used in comparing is listed in Appendix C.

MORE ABOUT PRINTING

Although the format of the printout is automatically supplied for the beginner, the PRINT statement, the TAB function, and image formatted output permit a greater flexibility for the advanced programmer in setting up different formats for his output.

PRINT

The teletypewriter line is divided into five zones of fifteen spaces each. Some control of the use of these zones 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 been used, to move to the first print zone on the next line.

Shorter zones can be made by use of the semicolon. The zones are three spaces long for 1-digit numbers, six spaces long for 2-digit, 3-digit, and 4-digit numbers, nine spaces long for 5-digit, 6-digit, and 7-digit numbers, twelve spaces long for 8-digit, 9-digit, and 10-digit numbers, and fifteen spaces long for 11-digit numbers. As with the comma, a semicolon is a signal to move to the next short print zone or, if the last such zone on the line has been used, to move to the first print zone of the next line.

The first space in any print zone is reserved for the sign, even though it is not printed out if it is plus.

If you typed the program

```
100 FØR I=1 TØ 15
110 PRINT I
120 NEXT I
999 END
```

the teletypewriter would print 1 at the beginning of the first line, 2 at the beginning of the next line, and so on, finally printing 15 at the beginning of the fifteenth line. But if you changed line 110 to read

```
110 PRINT I,
```

you would have the numbers printed in zones, reading

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

If you wanted the numbers printed in the same fashion, but more tightly packed, you could change line 110 to replace the comma with a semicolon

```
110 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 statement signals a new line, unless a comma or semicolon is the last symbol. The instruction

```
150 PRINT X,Y
```

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

```
150 PRINT X, Y,
```

will result in the printing of two numbers and no return. The next number to be printed will be printed in the third zone on the same line as the values of X and Y.

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

```
250 PRINT
```

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

```
100 FØR M=1 TØ N
110 FØR  J=0 TØ M
120 PRINT B(M,J);
130 NEXT J
140 PRINT
150 NEXT M
```

The program will print B(1,0) and next to it B(1,1). Without line 140, the teletypewriter would then go on printing B(2,0), B(2,1), and B(2,2) on the same line, and even B(3,0), B(3,1), and so on, if there were room. Line 140 directs the teletypewriter to start a new line after printing the B(1,1) value corresponding to M=1, and to do the same thing after printing the value of B(2,2) corresponding to M=2, and so on.

TAB

The print function TAB vermits tabbing of the teletypewriter. Whenever the TAB function is used in the PRINT statement, it will cause the print head to move over to the position indicated by the argument of TAB.

Example:

```
150 PRINT X; TAB(10); Y; TAB(2*N); Z
```

The argument of TAB refers to a print position on the teletypewriter line. The positions are assumed to run from 0 through 74. In the example, if the value of N is 10, the print head will move to the 10th print position after printing the value of X, and to the 20th print position after printing the value of Y.

When using the TAB function, you should use the semicolon in the PRINT statement in order to minimize field width.

If the argument of TAB is less than the current teletypewriter print head position, it is ignored.

All arguments of TAB are treated modulo 75.

Rules for Printing Numbers

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

- If a number is an integer, the decimal point is not printed. If the integer is larger than or equal to 2^{30} (i.e. 1,073,741,824), the teletypewriter will print the first digit, followed by (1) a decimal point, (2) the next five digits, and (3) an E followed by the appropriate exponent integer. For example, it will print 32,437,580,259 as 3.24376E+10.
- For any decimal number, no more than six significant digits are printed.
- 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. For example, .03456 means that the number is exactly .0345600000, but 3.45600E-2 means that the number has been rounded to .0345600.
- Trailing zeroes 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.

```
100 FØR I=1 TØ 45
110 PRINT 2↑I;
120 NEXT I
999 END
RUN


PRINT     14:53

 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    134217728
 268435456   536870912   1.07374E+09    2.14748E+09    4.29497E+09
 8.58993E+09    1.71799E+10    3.43597E+10    6.87195E+10    1.37439E+11
 2.74878E+11    5.49756E+11    1.09951E+12    2.19902E+12    4.39805E+12
 8.79609E+12    1.75922E+13    3.51844E+13
```

PRINT USING AND IMAGE STATEMENTS

You can set up formatted line output by use of the PRINT USING and image statements.

The form of the PRINT USING statement is

PRINT USING <u>line number</u>, <u>output list</u>

where the line number is that of the image statement to be used in formatting the output line, and the output list can consist of numbers, variables, string constants, string variables, and functions.

The form of the image statement is

<u>line number</u>:<u>line image</u>

where the line number is that of the image statement in the program, and the line image consists of format control characters and printable constants.

Format control characters are

'	(apostrophe) a one-character field that is filled with the first character in an alphanumeric string regardless of the string length.
" "	(quotation marks) the replacement field of an alphanumeric string of two or more characters; the field width includes the quotation marks as well as the characters (if any) contained within the marks.
#	(pound sign) the replacement field for a numeric character.
↑↑↑↑	(four up-arrows) indicates scientific notation for a numeric field.

All other characters are treated as printable constants.

The following simple example is part of a program, showing the use of the PRINT USING statement, the line image statement, and format control characters.

Example:

```
110 PRINT USING 120,A$,"$",A,324,X
120:"  "        '###.##      #####      ##.##↑↑↑↑
```

If the values of A$, A, and X are FIRST, 12.9, and 24687, output is

```
FIRST      $ 12.90       324       2.47E+04
```

An image statement must begin with a colon. It is composed of fields which form the print line. There are five types of fields:

- Integer fields
- Decimal fields
- Exponential fields
- Alphanumeric fields
- Literal fields

Integer Fields

The following rules apply to integer fields.

- An integer field is composed of pound signs (#).
- Numbers in an integer field are right justified and truncated if they are not integral.
- The field will be widened to the right if the number is too big.
- The field must reserve a place for the algebraic sign.
- If the number is greater than 1,073,741,823 in absolute value, an asterisk will be printed.

Example:

```
100: ####   #####   ###
110 READ A,B,C
120 PRINT USING 100,A,B,C
130 GØ TØ 110
140 DATA 123.45,-34.856,45.7,457.34,-17,89.999
999 END
RUN

PRINTU    15:00

   123      -34     45
   457      -17     89

ØUT ØF DATA   IN 110
```

Decimal Fields

The following rules apply to decimal fields.

- A decimal field is a string of pound signs (#) with an imbedded period. Note that .### is not a decimal field because the period is not imbedded.
- The number will be rounded to the number of places specified by the pound signs following the decimal.
- The number is right justified, placing the decimal as given in the field definition.
- The field will be widened to the right if the number is too large.
- The field must include a place for the algebraic sign.

Example:

```
100: ####.##    ####.####    #####.    #.###
110 READ A,B,C,D
120 PRINT USING 100,A,B,C,D
130 GØ TØ 110
140 DATA 123.456,-34.856,47.7,-.0177
150 DATA 1.999,876.55,-17,.893
999 END
RUN

PRINTU    15:03

   123.46      -34.8560      48.     -.018
     2.00      876.5500     -17.      .893

ØUT ØF DATA   IN 110
```

Exponential Fields

The following rules apply to exponential fields.

- An exponential field is a decimal field followed by four up-arrows(↑), which reserve a place for the exponent.
- The pound signs preceding the decimal represent the factor by which the exponent will be adjusted.
- The number will be rounded as with decimal fields.
- A place must be reserved for the sign.

Example:

```
100:  #.#####↑↑↑↑      ##.###↑↑↑↑      ###.↑↑↑↑      #.##↑↑↑↑
110 READ A,B,C,D
120 PRINT USING 100,A,B,C,D
130 GØ TØ 110
140 DATA 123.456,-34.856,47.7,-.0177
150 DATA 1.999,876.55,-17,.893
999 END
RUN

PRINTU     15:05

    .12346E+03    -3.486E+01     48.E+00     -.18E-01
    .19990E+01     8.766E+02    -17.E+00      .89E+00

ØUT ØF DATA  IN 110
```

Alphanumeric Fields

The following rules apply to alphanumeric fields.

- The apostrophe is used to print the first character from a string variable or quoted constant.
- A field bounded by quotation marks is used to print two or more characters.
- In an alphanumeric field of two or more characters, the string is left justified within the field and blank filled or truncated on the right.

Example:

```
100:  "23456"    "THE NAME GØES HERE"    '    ''
110 READ A$,B$,C$,D$,E$
120 PRINT USING 100,A$,B$,C$,D$,E$
130 DATA ABCDEFGHI
140 DATA ABCDEF
150 DATA ABC
160 DATA ABC
170 DATA ABC
999 END
RUN

PRINTU     15:08

  ABCDEFG    ABCDEF                     A     AA
```

Literal Fields

A literal field is composed of characters or character strings that are not control characters. It will appear on the print line exactly as it appears in the image.

Example:

```
100:  THE VALUE FØR A IS    '####.##
110 LET A=100.54
120 LET A$="$"
130 PRINT USING 100,A$,A
999 END
RUN

PRINTU     15:09

  THE VALUE FØR A IS    $ 100.54
```

General Rules

The following rules apply in general to formatted line output.

- A program may contain up to 100 images.
- The list elements in the PRINT USING statement may be expressions, variables, numeric constants, and quoted literals.
- Numeric list elements must replace numeric fields, and alphanumeric elements must replace alphanumeric fields, or you will receive the error message BAD IMAGE.
- If the output list contains more elements than there are replaceable fields in the image statement, a carriage return is supplied after the last field in the image, and the image is reused. The extra elements will be printed on a second line only if they match the image fields that are to be used.

Example:

```
100 PRINT USING 120
110 PRINT
120:           I          I↑2        I↑3
130:     #######     #######     #######
140 FØR I=1 TØ 6
150 LET A(I)=I
160 LET B(I)=I↑2
170 LET C(I)=I↑3
175 NEXT I
180 FØR I=1 TØ 6 STEP 2
190 PRINT USING 130,A(I),B(I),C(I),A(I+1),B(I+1),C(I+1)
200 NEXT I
999 END
RUN

PRINTU     15:14

           I        I↑2        I↑3

           1          1          1
           2          4          8
           3          9         27
           4         16         64
           5         25        125
           6         36        216
```

The following program demonstrates one kind of application in which the formatted output line is useful.

Example:

```
100 PRINT USING 170
110 PRINT
120 FØR I=1 TØ 4
130 READ A$,A,B
140 LET T=A*B
150 PRINT USING 180,A$,A,B,"$",T
160 NEXT I
170:NAME           HRS WØRKED         RATE         PAY
180:"      "         ####.##       ##.###/HR   '####.##
190 DATA ANDREWS,47.5,3.987,KELLY,40,2.865,MANLEY,46,3.020
200 DATA ZUMPANØ,42.34,4.255
999 END
RUN
WAIT.


FØRMAT    15:19

NAME           HRS WØRKED         RATE         PAY

ANDREWS             47.50       3.987/HR   $ 189.38
KELLY               40.00       2.865/HR   $ 114.60
MANLEY              46.00       3.020/HR   $ 138.92
ZUMPANØ             42.34       4.255/HR   $ 180.16
```

FUNCTIONS

There are two functions that were listed in Chapter 1 but not described: INT and RND.

Three other functions that you will sometimes find useful are SGN, CLK, and TIM. And you can write your own functions by use of the DEF statement.

INT

The INT function is the one that frequently appears in algebraic computation as [x], and it gives the greatest integer not greater than x. Thus INT(2.35) equals 2, INT(-2.35) equals -3, and INT(12) equals 12.

One use of the INT function is to round numbers. We can use it to round to the nearest integer by asking for INT(X+.5). This will round 2.9, for example, to 3, by finding

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

You should convince yourself that INT(X+.5) will do the rounding guaranteed for it, that 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, INT(10*X+.5)/10 will round X correct to one decimal place; INT(100*X+.5)/100 will round X correct to two decimal places; and INT(X*10↑D+.5)/10↑D will round X correct to D decimal places.

RND

The function RND is a pseudo random number generator. It requires a single argument, which has the following meanings:

- If the argument is positive, the argument is used to initiate the random number sequence.
- If the argument is negative, a random number is used to initiate the random number sequence.
- If the argument is zero, RND will supply a random number. The first use of RND(0) in a program will always yield the same random number.

A positive or negative argument would probably be used to initiate a sequence of random numbers, after which a zero argument would be used repeatedly.

If the initial value used for the argument is any power of 2, the same initial random number results as when 2 is used.

If we want the first twenty random numbers, we can use the following program to get twenty six-digit decimals.

Example:

```
100 LET X=RND(1)
110 FØR L=1 TØ 20
120 PRINT RND(0),
130 NEXT L
999 END
RUN
```

```
RNDTST      15:23

 .473599          .442519          .498805          .373168          .921321
 .123978          8.68505E-02      4.06526E-02      .341097          .468896
 2.97623E-02      .75441           .498551          .242898          9.31652E-02
 .280064          .159309          .211611          .042684          .383241
```

If, on the other hand, we want twenty random one-digit integers, we can change line 120 to read

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

and we then obtain

```
RNDTST      15:24

 4  4  4  3  9  1  0  0  3  4  0  7  4  2  0  2  1  2  0  3
```

We can vary the kind of random numbers we get. For example, if we want twenty random numbers ranging from 1 to 9 inclusive, we can change line 120 as shown below

```
120 PRINT INT(9*RND(0)+1);
RUN
```

```
RNDTST      15:25

 5  4  5  4  9  2  1  1  4  5  1  7  5  3  1  3  2  2  1  4
```

Or we can obtain random numbers which are integers from 5 to 24 inclusive by changing line 120 as follows

```
120 PRINT INT(20*RND(0)+5);
RUN


RNDTST    15:26

 14    13    14    12    23    7  6  5  11    14    5  20    14    9  6
 10    8  9  5  12
```

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

INT(A*RND(0)+B)

after first having initiated the random number sequence with a positive or negative argument, as in line 100 of our sample program.

If you were to run the first version of our sample program again, you would get the same twenty numbers in the same order. But we can get a different set by throwing away some of the random numbers. In the following program we find the first ten random numbers and do nothing with them. We then find the next twenty and print them. You can see, by comparing this with the earlier program, that the first ten of these random numbers are the same as the second ten of the first program.

Example:

```
100 LET Z=RND(1)
110 FØR I=1 TØ 10
120 LET Y=RND(0)
130 NEXT I
140 FØR I=1 TØ 20
150 PRINT RND(0),
160 NEXT I
999 END
RUN


RNDTST    15:28

 2.97623E-02     .75441          .498551         .242898         9.31652E-02
 .280064         .159309         .211611         .042684         .383241
 8.89948E-02     .140658         .643944         .829565         5.32346E-02
 .795665         .66257          .384661         .817653         .942257
```

SGN

The function SGN allows you to test for the sign of any value. The form is SGN(argument) and it yields +1, -1, or 0 depending on the value of the argument. The options are

Argument Value	Yields
Zero	0
Positive, not zero	+1
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

CLK

The function CLK(X), X being a dummy argument, yields the time of day in military hours.

Examples:

```
100 PRINT CLK(X)
295 IF CLK(X)>15.00 THEN 1000
130 IF A-CLK(X)<.5 THEN 1000
```

TIM

The function TIM(X), X being a dummy argument, yields the program elapsed time in seconds.

Examples:

```
100 PRINT TIM(X)
688 IF TIM(X)>10 THEN 1000
```

DEF

In addition to making use of the standard functions, you can define any other function that you expect to use several times in your program. You use a DEF statement to define such a function. The name of the defined function must be three letters, the first two of which are FN. Hence you can define up to 26 functions in one program: FNA, FNB, FNC, and so on.

The usefulness of DEF you can see in a program, for example, where you often need the function e^{-x^2}. You introduce the function by the line

```
130 DEF FNE(X)=EXP(-X↑2)
```

and later on call for various values of the function by FNE(.1), FNE(3.45), FNE(A+2), and so on. DEF can be a great time-saver when you want values of some function for a number of different values of the variable.

The DEF statement may be put anywhere in the program, and the expression to the right of the equal sign may be any formula that can be fitted on one line. It may include any combination of other functions, including ones defined by other DEF statements, and it can involve other variables besides the one denoting the argument of the function. For example, if FNR is defined by

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

and 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 you want to.

DEF is generally limited to cases where the value of the function can be computed within a single BASIC statement. Often much more complicated functions, or even pieces of a program, must be calculated at several different points within a program. For these functions, the GØSUB statement will frequently be useful. It is described in the next section.

ADDITIONAL BASIC STATEMENTS

Several kinds of BASIC statements were not covered in Chapter 1. These are discussed in this section. They are:

- GØSUB and RETURN
- CALL
- STØP
- REM
- RESTØRE
- CHAIN

GØSUB and RETURN

When a particular part of a program is to be used more than one time, or possibly at several different places in the overall program, it is most efficiently programmed as a subroutine. The subroutine is entered with a GØSUB statement.

Example:

```
190 GØSUB 310
```

The line number, 310, is the line number of the first statement in the subroutine.

The last line of the subroutine should be a RETURN statement, directing the computer to return to the earlier part of the program.

Example:

```
450 RETURN
```

This statement, if it is the last line in the subroutine entered in the previous example, tells the computer to go back to the first line numbered greater than 190 and continue the program there.

You may use a GØSUB statement inside a subroutine to execute yet another subroutine. This is called nested GØSUBs. It is absolutely necessary that a subroutine be left only with a RETURN statement. Using a GØ TØ or an IF--THEN to get out of a subroutine will not work correctly. You may have several RETURNs in the subroutine so long as only one of them will be used.

You should be very careful not to write a program in which a GØSUB appears inside a subroutine that refers to one of the subroutines already entered. Recursion is not allowed.

The following example, a program for determining the greatest common divisor of three integers using the Euclidean Algorithm, illustrates the use of a subroutine.

Example:

```
100 PRINT "A","B","C","GCD"
110 READ A,B,C
120 LET X=A
130 LET Y=B
140 GØSUB 230
150 LET X=G
160 LET Y=C
170 GØSUB 230
180 PRINT A,B,C,G
190 GØ TØ 110
200 DATA 60,90,120
210 DATA 38456,64872,98765
220 DATA 32,384,72
230 LET Q=INT(X/Y)
240 LET R=X-Q*Y
250 IF R=0 THEN 290
260 LET X=Y
270 LET Y=R
```

```
280 GØ TØ 230
290 LET G=Y
300 RETURN
999 END
RUN


GCD3NØ     15:30

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

ØUT ØF DATA   IN 110
```

The first two numbers are selected in lines 120 and 130, and their GCD is determined in the subroutine, lines 230 - 300. The GCD just found is called X in line 150, the third number is called Y in line 160, and the subroutine is entered from line 170 to find the GCD of these two numbers. This number, the GCD of the three given numbers, is printed out with the three numbers in line 180.

CALL

The CALL statement is used to call an external program for use as a subroutine within the main program just as the GØSUB statement calls a subroutine inside the main program. The statement form is CALL saved program name.

Examples:

```
100 CALL HISDWN
200 CALL EQCLS*
```

You can call either previously saved programs of your own, as in line 100; common programs in your catalog library, as in line 200; or system library programs, either regular or run-only.

The standard program naming rules apply.

Examples:

```
140 CALL A B
150 CALL AB
```

Statements 140 and 150 both call a program named AB, since BASIC ignores all leading, trailing, and imbedded blanks. No arguments are permitted after the program name in the CALL statement. Subroutines may call other routines, but no program may call the main program or itself.

The return from a subroutine to the calling program is by a RETURN statement. Multiple returns are permissible. The return is always to the statement immediately following the statement in which the program was called.

All variables and defined functions are common to the main program and the called subroutines. They need not be defined separately in each program.

Example:

```
NEW
NEW FILE NAME--DEFPRT
READY.

100 LET Y=4
110 LET X=7
220 LET A=FNP(3)
230 RETURN
999 END
SAVE
WAIT.

READY.

NEW
NEW FILE NAME--MAIN
READY.

100 DEF FNP(Z)=SQR(X↑2+Y↑2+Z↑2)
110 CALL DEFPRT
120 PRINT A
999 END
RUN
WAIT.

MAIN       15:35

 8.60233
```

Statement 110 calls DEFPRT, which stores 4 in Y and 7 in X, and calculates a value for A by use of a function defined in line 100 of MAIN. The function uses the value 3 for Z as defined in statement 220 of DEFPRT. DEFPRT then returns to the statement immediately following the CALL statement, and the calculated value for A, 8.60233, is printed.

An END or STØP statement terminates all execution, whether it is executed in a subroutine or in the main program.

The line numbers in the different programs are completely independent. GØ TØ and IF--THEN statements reference line numbers in their own program only.

Data is compiled from the main program first, and then from each of the called programs in the order in which the CALL statements are encountered.

Consider the following example.

Example:

```
ØLD
ØLD FILE NAME--SUBR


READY.

LIST


SUBR       15:38

100 READ X,Y,Z
110 DATA 6,7,8
120 RETURN
999 END
```

```
ØLD
ØLD FILE NAME--MAIN

READY.

LIST

MAIN       15:39

100 READ A,B
110 CALL SUBR
120 READ C,D,E
130 DATA 1,2
140 DATA 3,4,5
150 PRINT A,B
160 PRINT X,Y,Z
170 PRINT C,D,E
999 END

RUN

MAIN       15:40

 1              2
 3              4              5
 6              7              8
```

Statement 100 reads the numbers 1 and 2 into A and B. Statement 110 transfers control to SUBR, which reads 3, 4, and 5 (not 6, 7, and 8) into variables X, Y, and Z. After the return to statement 120, 6, 7, and 8 are read into C, D, and E.

The CALL statement allows more effective use of program space available. A program referred to by a CALL statement will be compiled only once no matter how many times it is called in each of the routines. Object code generated by the called program counts toward object code limitation, but the characters in the called program do not count toward the BASIC character limitation.

As many as 10 different programs may be called.

STØP

The STØP statement is equivalent to GØ TØ 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 parts of programs are exactly equivalent.

Example:

```
250 GØ TØ 999                 250 STØP
     .                             .
     .                             .
     .                             .
340 GØ TØ 999                 340 STØP
     .                             .
     .                             .
     .                             .
999 END                       999 END
```

REM

The REM statement allows you to insert explanatory remarks in a program. The form is REM any comment. The computer completely ignores the part of the line following REM, allowing you to include directions for using the program, identifications of the parts of a long program, or anything else. Although what follows REM is ignored, you may use the line number of a REM statement in a GØSUB or IF--THEN statement.

Examples:

```
100 REM INSERT DATA IN LINES 900-998.  THE FIRST
110 REM NUMBER IS N, THE NUMBER ØF PØINTS.  THEN
120 REM THE DATA PØINTS THEMSELVES ARE ENTERED, BY


200 REM THIS IS A SUBRØUTINE FØR SØLVING EQUATIØNS.
     .
     .
     .
300 RETURN
     .
     .
     .
520 GØSUB 200
```

RESTØRE

Sometimes it is necessary to use data in a program more than once. The RESTØRE statement permits reading the data as many additional times as necessary. Whenever RESTØRE is encountered in a program, the computer restores the data block pointer to the first item of data. A subsequent READ statement will then start reading the data all over again.

One word of warning: if the data items you wish to use again are preceded by code numbers or parameters, superfluous READ statements should be used to pass over the numbers.

As an example, the following part of a program 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.

Example:

```
100 READ N
110 FØR I=1 TØ N
120 READ X
     .
     .
     .
200 NEXT I
     .
     .
     .
560 RESTØRE
570 READ X
580 FØR I=1 TØ N
590 READ X
```

CHAIN

The CHAIN statement allows you to stop the execution of the current program and begin compilation and execution of another program without direct intervention. The CHAIN statement is equivalent to giving the commands STØP, ØLD, a program name, and RUN.

The statement form is

CHAIN saved program name

or

CHAIN saved program name, line number

Only one program name may appear in a statement. The name must conform to the rules used in naming BASIC programs.

Examples:

```
100 CHAIN NEXT
100 CHAIN NEXT,100
100 CHAIN PLØTER***
100 CHAIN TUT01$***
100 CHAIN PAYRØL,555
```

Notice that, as shown in the examples, BASIC library programs may be chained.

When a number appears after the saved program name, as in the second and fifth lines of the examples, the number indicates the line number of the named program at which execution is to begin. When no number appears, execution begins with the first executable statement of the named program.

Once a CHAIN statement is executed, the current program is stopped and the named program brought in. Because there is no logic path to any statements following the CHAIN statement, all needed current program statements must be executed before the CHAIN statement.

DATA FILES

An external data file is a saved program in which you can record information for later use. There are two types of data files: BCD and binary. The data in a file must all be of one type.

BCD data files may be created with a BASIC program or they may be typed in as a saved program. Thus all of the editing commands, such as LIST, EDIT DELETE, and so on, may be used in accessing and modifying BCD files.

Binary data files can be created only with a BASIC program. They cannot be created from a terminal or altered in the same manner as BCD files.

If a BCD or binary file is to be written during program execution, an area on the disc large enough to contain the entire file to be generated must be reserved in your library before program execution. This is done by renaming and saving special files available in the system library (see "Dummy Catalog Files"). These files are empty. Their purpose is to preset the size of the files to be generated.

Example:

```
ØLD
ØLD FILE NAME--CH0768***
READY
RENAME
NEW FILE NAME--WFILE
READY
SAVE
READY
```

In the example, the data file WFILE has been created; it has a storage capacity of 768 characters.

File Reference

The form of the file reference statement is

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

where name 1, name 2, and so on are the names of files to be read or written by the program. *The file reference statements must precede all executable statements in the program.* The number of files that can be used in a program depends on the size of the program, but you can always reference at least 8 files in any program.

The named files must be saved in your catalog before running the program. Files referenced but not saved produce the error message FILE NØT SAVED when you run the program.

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

- File names must not contain semicolons. They will be interpreted as file name separators.
- Leading and imbedded blanks are ignored.
- The file name is left-justified.
- File names should not contain slashes or commas.

Examples:

```
100 FILES A;B;C

       or

100 FILES A
110 FILES B;C
```

The following file names are identical:

```
100 FILES XYZ
100 FILES XY Z
100 FILES X Y Z
```

File Designator

The file designator is used in all statements referencing files. It singles out a file named in the file reference statement. It may be an integer, an expression, a variable, or a subscripted variable.

Example:

```
100 FILES P;Q
     .
     .
     .
170 READ #1,A(I),B(I)
     .
     .
     .
195 READ #A*B,A(I),B(I)
```

Statement 170 refers to file P. Statement 195 refers to a file depending on the value of the expression A*B, which must be an integer. If the value of A*B is 1, file P is selected; if the value of A*B is 2, file Q is selected.

If the value of the file designator is less than one, non-integral, or greater than the number of files referenced, the error message ILLEGAL FILE DESIGNATØR will be printed.

Dummy Catalog Files

Since it is necessary to save a file of a size large enough to contain the data to be written, the system library contains the following six files which you can rename and save in your catalog.

Library Name	Characters	Library Name	Characters
CH0192***	192	CH1536***	1536
CH0384***	384	CH3072***	3072
CH0768***	768	CH6144***	6144

The number of data points that can be written into a file is a function of the number of characters in each data point. Line numbers and spaces must also be considered in the count.

BCD DATA FILES

BCD files are sequential access files. For each execution of the READ or WRITE statement, data is transmitted serially. BCD files contain line numbers and are listable.

If a BCD file with initial values is to be read from the disc, you must prepare it before program execution and save it in your catalog.

Example:

```
NEW
NEW FILE NAME--RFILE
READY
100 1,1.5,2,2.5,3
110 3.5,4,4.5,5,5.5
SAVE
READY
```

When preparing a BCD file, you may if you wish use a blank in place of a comma as a data separator. This option allows files prepared in FØRTRAN to be processed in BASIC and conversely. RFILE in the previous example can be prepared as:

```
100 1 1.5 2 2.5 3
110 3.5 4 4.5 5 5.5
```

Note that the word DATA is not needed in these files. The first number on each line is the line number.

All files are in either read mode or write mode. Initially, the FILES statement results in all files being set to the read mode. This protects you from accidentally destroying valuable files. You can later change the mode of any file by issuing a SCRATCH or RESTØRE statement. The SCRATCH statement establishes the designated file as write mode. The RESTØRE statement establishes the designated file as read mode.

File Reading

The form of the BCD read file statement is

READ #file designator, input list

where the file designator is as previously described. The pound sign denotes a BCD file.

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

Example:

A file containing 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5 is to be read into A(I) and B(I).

```
100 FILES RFILE
110 FØR I=1 TØ 5
120 READ #1,A(I),B(I)
130 NEXT I
```

For each execution of the READ # statement one value is read into an A(I) and B(I) so that, at the termination of the loop:

```
A(1) = 1            B(1) = 1.5
A(2) = 2            B(2) = 2.5
A(3) = 3            B(3) = 3.5
A(4) = 4            B(4) = 4.5
A(5) = 5            B(5) = 5.5
```

The file pointer will remain positioned following the last read data item (5.5 in the example) until further file statements designating the file (RFILE in the example) are executed.

File Writing

The form of the BCD write file statement is

WRITE #<u>file designator</u>, <u>output list</u>

where the file designator is as previously described.

The output list consists of the variables, separated by semicolons, from which the file is generated. The list may contain non-string variables, string variables, strings, and expressions. Subscripting is permissible in the output list.

WRITE # always generates a file beginning with line number 1000, incrementing by 10 for each new line. The line number sequencing can be modified, if you wish, by EDIT RESEQUENCE.

Each WRITE # statement generates one line of output unless the teletypewriter line limit is exceeded or the last list item is followed by a semicolon. When the teletypewriter line limit is exceeded, writing continues on the next line with the next data item. When the output list is followed by a semicolon, subsequent writing occurs on the same line in a closely packed format.

Example:

```
100 FILES WFILE
110 SCRATCH #1
120 FØR I=1 TØ 25
130 WRITE #1,I;I*I
140 NEXT I
```

When listed WFILE would contain

```
1000   1    1
1010   2    4
1020   3    9
  .    .    .
  .    .    .
  .    .    .
1240   25   625
```

Following is an example showing how strings and string variables are used with files.

Example:

```
100 FILES STRING
110 SCRATCH #1
120 LET A$="STRING1"
130 WRITE #1,A$;"STRING2"
140 WRITE #1,"STRING2";A$
```

Then listing the file STRING gives

```
1000 STRING1  STRING2
1010 STRING2  STRING1
```

End-of-File and End-of-Space

The end-of-file (EOF) is a special mark written by BASIC itself that indicates the end of data in the file.

The end-of-space (EOS) is the physical end of the disc area reserved for a file. When a file is completely filled with data, EOF = EOS; otherwise EOF< EOS.

Whenever a word is generated with a WRITE # statement, an EOF mark is placed immediately following the last word written. Subsequent transmissions to the file move the EOF mark so that it always follows the last word written. When a file is generated from the teletypewriter, an EOF mark is placed immediately after the last data item in the file as soon as the file is saved.

End-of-File Test

The form of the statement that tests for an EOF mark or the EOS is

IF END #<u>file designator</u> THEN <u>line number</u>

where the file designator is as previously described.

The statement will test whether an EOF was detected by the last command reading the designated file. When writing a file, the statement will test whether an EOS was detected.

If the last READ # statement found an EOF mark, the program will go to the line number specified in the IF END # statement. Otherwise the next sequential statement is executed.

If the program continues reading or writing a file after the EOF or EOS has been detected, an error message (END ØF FILE or END ØF SPACE) will be printed and the program will continue. The error message will be printed out each time an attempt is made to exceed the EOF or EOS limit.

Example:

```
100 FILES F1;F2
110 SCRATCH #2
120 DIM X(100),Y(100),Z(100)
130 FØR I=1 TØ 100
140 IF END #1 THEN 180
150 READ #1,X(I),Y(I),Z(I)
160 WRITE #2,SQR(X(I)↑2+Y(I)↑2+Z(I)↑2)
170 NEXT I
180 STØP
```

If file F1 contained 300 data items, no EOF would be encountered; but if it contained only 150 data items, for example, the IF END # statement (line 140) would cause a transfer to line 180 following the 50th execution of the loop.

File Restoring

The form of the BCD restore file statement is

RESTØRE #<u>file designator</u>

where the file designator is as previously described.

The RESTØRE # statement causes the position of the designated file pointer to be moved so that the next transmission is from the beginning of the file.

If a file is already restored, the RESTØRE # statement merely sets the file to read mode.

BASIC automatically restores the referenced files before a program begins executing.

Example:

```
100 FILES UTIL
110 SCRATCH #1
120 FØR I=1 TØ 50
130 WRITE #1,I;SQR(I)
140 NEXT I
150 RESTØRE #1
     .
     .
     .
170 READ #1,X,X1
```

In the example the RESTØRE # statement (line 150) is required to restore the written file before reading it. Reading then begins at the first data item in the file.

A file being written cannot be read before it is restored. A file being read cannot be written before it is scratched. This means that if any modifications are to be made to an existing file via the program, it must be copied (written) to another file up to the modification point. The modification can then be written into the second file.

File Scratching

The form of the BCD scratch file statement is

SCRATCH #<u>file designator</u>

where the file designator is as previously described. This statement causes the designated file to be scratched, or made ready for writing. If the file is already reset or restored the statement merely sets the file to write mode. Following the SCRATCH # statement, transmission to the file commences at the beginning of the file.

Example:

```
100 FILES UTIL
110 SCRATCH #1
120 FØR I=1 TØ 50
130 WRITE #1,RND(0)
140 NEXT I
150 RESTØRE #1
160 READ #1,X,Y,Z
```

In the example, the SCRATCH # statement (line 110) must be executed before the write into the file. The RESTØRE # statement must be executed before reading from the written file.

File Backspacing

The form of the BCD backspace file statement is

BACKSPACE #file designator

where the file designator is as previously described. This statement is permitted only on files in the read mode.

The BACKSPACE # statement causes the position of the data pointer for the file being read to be moved backward one data item. If the data pointer is already at the beginning of the file, the backspace statement is ignored.

Some applications require a file to be processed forward and then backward. The following example illustrates how this can be done.

Example:

```
100 FILES F1
110 IF END #1 THEN 210
120 READ #1,A
130 REM CØUNT NUMBER ØF PØINTS IN FILE
140 LET N=N+1
       .
       .
       .
200 GØ TØ 110
210 FØR I=1 TØ N
220 BACKSPACE #1
230 READ #1,A
240 BACKSPACE #1
       .
       .
       .
300 NEXT I
```

BINARY DATA FILES

Binary data files are not listable. The file space normally used for line numbers in files can be used for data. They are random access files; that is, data can be written into or read from a binary file in any order. Since data conversion routines are eliminated in processing binary files, program efficiency is increased.

For binary files, the restrictions on file mode are eliminated. Binary files may be read or written without having been previously restored or scratched. The RESTØRE and SCRATCH statements serve only to move the data element pointer to the first element in the file.

File Writing

The form of the binary write file statement is

WRITE : file designator, output list

where the file designator is as previously described. The colon denotes a binary file.

The output list may contain numeric variables, string variables, or literals. The numeric variable or literal will cause a two-word entry into the file. The string variable or literal will cause a six-word entry into the file. Each two-word entry is defined as a data element. Consequently each numeric variable or numeric literal consists of one data element, and each string variable or string literal consists of three data elements.

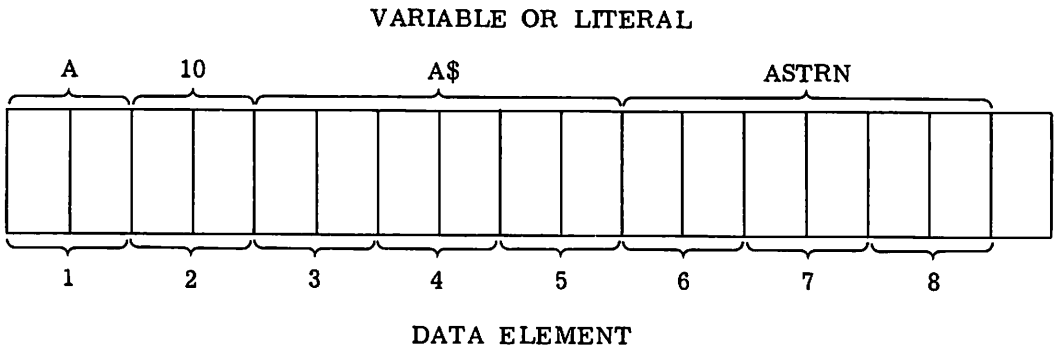
For example, a numeric variable or literal may appear in a WRITE : statement as:

```
100 WRITE:1,A           (numeric variable)
110 WRITE:1,10          (numeric literal)
```

A string variable or literal may appear in a WRITE : statement as:

```
120 WRITE:1,A$          (string variable)
130 WRITE:1,"ASTRN"     (string literal)
```

The following diagram illustrates how each of these will be entered into the file. In the diagram, each block represents one computer word.



Binary files contain no line numbers. Data may be written into the file sequentially or randomly.

The following is an example of a sequentially created file.

Example:

```
100 FILES WFILE
110 SCRATCH:1
120 FØR I=1 TØ 25
130 WRITE:1,I;I*I
140 NEXT I
```

File WFILE will contain 25 pairs of numbers, or 50 successive data elements.

In the following example, string information is written into a binary file.

Example:

```
100 FILES STRING
110 SCRATCH:1
120 LET M$="MSTRING"
130 WRITE:1,M$;"NSTRING"
```

File STRING will contain two strings or six data elements, three data elements for each string.

File Reading

The form of the binary read file statement is

READ : file designator, input list

where the file designator is as previously described.

The input list may contain numeric variables or string variables. The numeric variable will cause one data element (two words) to be read from the file. The string variable will cause three data elements (six words) to be read from the file.

Data may be read sequentially or randomly from a file.

In the following example data is read sequentially.

Example:

```
100 FILES RFILE
110 FØR I=1 TØ 5
120 READ:1,A(I),B(I)
130 NEXT I
```

The first ten data elements in file RFILE will be read alternately into arrays A and B.

Random Accessing

The form of the random access statement is

SET : file designator, variable

where the file designator is as previously described.

The SET : statement will position the pointer of the designated file to the element specified by the variable. The variable may be an integer or an expression. If the value of the variable is less than one, non-integral, or greater than the length of the file, the error message ILLEGAL PØINTER will be printed out.

Example:

```
100 FILES VFIL
110 SCRATCH:1
120 FØR I=1 TØ 25
130 WRITE:1,I*I
140 NEXT I
150 SET:1,7
160 READ:1,X,Y
```

File VFIL will contain 25 elements. In line 150, the pointer will be positioned to the 7th element. Reading then begins at the 7th element, and the values 49 and 64 will be read into X and Y.

If the file being processed contains string variables, the length of a string entry must be considered before the SET : statement is used.

Example:

```
100 FILES RNF
110 SCRATCH:1
120 WRITE:1,"ASTRN";"BSTRN"
130 SET:1,4
140 READ:1,B$
```

When the WRITE : statement is executed, ASTRN will occupy the first three entries of file RNF, and BSTRN will occupy the next three entries. To access the second string, the pointer must be set to the position where the second string begins, which is done by line 130. The string data, BSTRN, is then read into the string variable, B$.

The following uses of the SET : statement are acceptable.

```
180 SET:1,A*B
195 SET:3,A(I)
```

In each case the value of the variable must be an integer within the limits of the file.

Locating the Element Pointer

Since the element pointer can be moved randomly in a binary file, it is useful to have a method of finding out where the pointer is at any time during the execution of a program.

The LØC function will provide the location of the pointer in the file referenced.

For example:

LET X=LØC(file designator)

In this statement, X represents any numeric variable, and the file designator is as previously described.

Example:

```
100 FILES F1;F2
     .
     .
     .
210 LET A1=LØC(1)
     .
     .
     .
290 PRINT LØC(2)
```

The value of the element pointer into file F1 will be stored in A1 (line 210).

The value of the element pointer into file F2 will be printed (line 290).

File Scratching

The form of the binary scratch file statement is

SCRATCH : file designator

where the file designator is as previously described.

The SCRATCH statement cause the data element pointer to be repositioned so that transmission to the file starts at the beginning of the file.

Example:

```
100 FILES VFIL
115 SCRATCH:1
125 FØR I=1 TØ 20
140 WRITE:1,INT(10*RND(X))
155 NEXT I
```

The SCRATCH : statement may be replaced with a SET : statement to accomplish the same purpose:

```
115 SET:1,1
```

A file being read can be written without being scratched. This means that files can be modified during program execution.

Example:

```
100 FILES RFIL
110 LET Z1=0
120 LET S1=26
130 SET:1,S1
140 READ:1,X
150 IF X=0 THEN 180
160 SET:1,LØC(1)-1
170 WRITE:1,Z1
180 LET S1=S1+26
190 IF S1<EØF(1) THEN 130
200 END
```

This example reads every 26th data element and goes back and writes over that element with a zero, provided that the element is not already zero.

File Restoring

The form of the binary restore file statement is

RESTØRE : <u>file designator</u>

where the file designator is as previously described.

The RESTØRE : statement will move the element pointer to the beginning of the designated file.

Example:

```
100 FILES VFIL
110 SCRATCH:1
120 FØR J=1 TØ 50
130 WRITE:1,J;SQR(J)
140 NEXT J
150 RESTORE:1
      .
      .
      .
180 READ:1,X
```

This program writes 100 data elements into VFIL. Then the element pointer is moved to the beginning of the file (line 150), and the first data element is read into X.

A file being written can be read before it is restored. If line 150 in the last example is omitted, reading will begin where writing stopped, and the 101st data element will be read into X.

End-of-File Test

For binary files the end-of-file test is a test for the end of file space. The IF END statement will test whether an end-of-file-space condition was detected by the last READ: or WRITE: statement. When reading a partially filled binary file, the READ: statement should be executed only the number of times required to read the legitimate data in the file. The IF END test would allow such a file to be read until all the space in the file had been exhausted.

The form of the statement is

IF END : file designator THEN line number

If the last READ: statement or WRITE: statement encountered the end of file space, the program will go to the line number specified in the IF END: statement. Otherwise the next sequential statement is executed.

Example:

```
100 FILES F1
110 SCRATCH:1
120 FØR J=1 TØ 1000
130 IF END:1 THEN 160
140 WRITE:1,J/2
150 NEXT J
160 END
```

An intrinsic function is also provided to test for the end-of-file condition. The function LØF will retrieve the length of the referenced file. For example:

LET X = LØF (file designator)

In this statement, X represents any numeric variable, and the file designator is as previously described.

The functions LØC and LØF can replace the IF END : statement in the previous sample program as follows:

```
130 IF LØC(1)=LØF(1) THEN 160
```

When the element pointer into file F1 reaches the end of the file, writing stops.

If a program continues reading or writing a file after the end-of-file-space condition has been detected, the error message END ØF FILE SPACE will be printed, and the program will continue executing.

MATRICES

The matrix operation statements available in BASIC are among the most powerful and useful in the entire language.

Following is a list of matrix statements.

MAT READ A,B,C,	Read matrices A, B, and C, their dimensions having been previously specified. Data is read in row-wise sequence.
MAT PRINT A,B;C	Print matrices A, B, and C, with A and C in the regular format, but B closely packed.
MAT C=A+B	Add matrices A and B and store the result in matrix C.
MAT C=A-B	Subtract matrix B from matrix A and store the result in matrix C.
MAT C=A*B	Multiply matrix A by matrix B and store the result in matrix C.
MAT C=INV(A)	Invert matrix A and store the result in matrix C.

MAT C=TRN(A)	Transpose matrix A and store the result in matrix C.
MAT C=(K)*A	Multiply 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=CØN	Set each element of matrix C to one. CØN means constant.
MAT C=ZER	Set each element of matrix C to zero.
MAT C=IDN	Set the diagonal elements of matrix C to one's and the non-diagonal elements to zeroes, yielding an identity matrix.

MAT READ and MAT PRINT

Using the MAT READ and MAT PRINT statements, you can read data into or print data from a matrix without having to reference each element of the matrix individually.

Examples:

```
100 MAT READ A,F,H,G
150 MAT PRINT C
175 MAT READ Z
190 MAT PRINT A,L
```

Information is read into a matrix using the DATA statement. The elements in the DATA statement are taken in row order, that is,

$$A_{1,1}, A_{1,2}, \ldots, A_{1,m}, A_{2,1}, A_{2,2}, \ldots, A_{2,m}, \ldots, A_{n,m}.$$

Information is read from DATA statements until the matrix array is completely filled. Partial matrices cannot be read or printed.

Example:

```
110 DIM L(2,3),M(2,2)
150 MAT READ L,M
160 LET L(2,2)=-2*L(2,2)
200 MAT PRINT L,M
500 DATA 1,2,3,4,5,6,3,-12,0,7
```

Line 110 defines L as a 2 by 3 matrix and M as a 2 by 2 matrix. The MAT READ statement reads in row order from the DATA statement at line 500. The matrix element $L_{2,2}$ is recomputed at line 160. The two matrices are then printed to yield:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & -10 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 3 & -12 \\ 0 & 7 \end{bmatrix}$$

Matrix Addition, Subtraction, and Multiplication

You can add, subtract, and multiply matrices using the matrix arithmetic statements. The matrix dimensions must be conformable for each operation. If dimensions are not conformable, execution is stopped and you receive a dimension error message.

Matrix arithmetic statements may take the forms

```
MAT C=A+B          MAT C=A-B          MAT P=Q*R
```

Only one operation can be done in each statement.

Example:

Calculate [H] a,a = [E] a,a- [K] a,a + [A] a,3* [B] 3,a

```
612 MAT H=A*B
615 MAT H=H+E
618 MAT H=H-K
```

Scalar Multiplication

You can multiply a matrix by a scalar expression using a statement of the form

MAT X = (<u>expression</u>) *D

where X and D are matrices and the expression in parentheses is a scalar quantity. The parentheses are required to indicate scalar rather than matrix multiplication. Only one operation per statement is allowed.

Examples:

```
100 MAT F=(2)*G
150 MAT Q=(2.33+M)*Q
750 MAT B=(N)*A
```

Identity Matrix

An identity matrix is defined by a statement of the form

MAT B=IDN   or   MAT R=IDN (<u>expression</u>, <u>expression</u>)

In the first statement, matrix B is set up as an identity matrix. If B is not defined to be square, you will receive a dimension error message. In the second statement, the size of the identity matrix R is determined at execution time by the value of the expression enclosed in parentheses

Examples:

```
190 MAT A=IDN
100 MAT V=IDN(2*N+1,2*N+1)
120 MAT B=IDN(Q,Q)
130 MAT W=IDN
140 MAT C=IDN(1,1)
```

Matrix Transposition

Matrices are transposed using the form

MAT Y=TRN(Z)

where Y and Z are both matrices. The matrix Z transpose will replace matrix Y. Y and Z must conform. Matrix transposition in place (MAT A=TRN(A)) is not allowed.

Examples:

```
300 MAT G=TRN(H)
400 MAT U=TRN(V)
```

Matrix Inversion

Matrices are inverted using the form

```
MAT I=INV(J)
```

where I and J are both matrices. I will contain the matrix J inverse. I and J must conform. Matrix inversion in place (MAT A=INV(A)) is not allowed. If a matrix is singular, you will receive the message NEARLY SINGULAR MATRIX.

Examples:

```
500 MAT K=INV(L)
560 MAT A=INV(B)
```

Matrix ZER and CØN Functions

The ZER function is used to zero out all elements of a matrix. It may also be used to redefine the dimensions of a matrix during execution as described in "Dimensioning." As an example

```
MAT C=ZER
```

will zero out the elements of matrix C.

The CØN function is used to set all elements of a matrix to one's. As an example

```
MAT C=CØN
```

will set all elements of matrix C to one's.

Dimensioning

Every matrix variable used in a program must be given a single-letter name.

A matrix variable must be defined in a DIM statement, which sets aside the amount of storage required by the matrix variable during execution of the program. For example:

```
DIM P(3,4),Q(5,5)
```

The DIM statement defines two matrices, P and Q. P is defined as a 12 element matrix, and Q as a 25 element matrix. Note that the first element of P is P(1,1) and the last element P(3,4). The elements of Q run from Q(1,1) through Q(5,5). All matrix variables must be doubly dimensioned, as shown here.

Before any computation using the MAT statements, you must declare the precise dimensions of all matrices to be used in the computation. Four of the MAT statements are used for this purpose:

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

The first three statements specify matrix C as consisting of M rows and N columns. The fourth statement specifies matrix C as a square matrix of N rows and N columns.

These same statements 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 storage space than was originally reserved by the DIM statement. To illustrate, consider the following.

Example:

```
110 DIM  A(8,8),B(8,8),C(8,8)
150 MAT  READ A(2,2),B(2,2)
160 MAT  C=ZER(2,2)
          •
          •
          •
200 MAT  A=IDN(8,8)
210 MAT  READ B(4,4),C(4,4)
```

Note that the DIM statement reserves enough storage to accommodate three matrices, each consisting of 64 elements. The initial MAT READ specifies the dimensions of both matrices A and B as 2 rows and 2 columns.

The MAT READ also reads the number of values required by the dimensions into the storage that was reserved by the DIM statement. It reads them 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). Statement 160 uses the ZER to specify dimensions and to zero the elements of matrix C. Statements 200 and 210 illustrate redimensioning. Matrix A is redimensioned as an 8 row, 8 column identity matrix; and matrices B and C are redimensioned as 4 row, 4 column matrices into which data is to be read.

The combination of ordinary BASIC statements and MAT statements makes BASIC very powerful, but you must be careful about dimensions. In addition to having both a DIM statement and a declaration of current dimension, you should watch your use of the MAT statements. For example, a matrix product MAT C=A*B may be illegal for either of two reasons: A and B may have such dimensions that the product is not defined, or C may have the wrong dimensions for the answer. In either case you will receive the DIMENSIØN ERRØR message.

Examples

Two programs follow that illustrate some of the capabilities of the MAT statements. In the first program, the values for M and N are read. Using these two values as indices, statement 120 sets the dimensions for matrices A, B, D, and G. The values for the elements of these four matrices are read, Then, in sequence:

- The dimensions of matrix C are specified and the elements set to zero (line 130).
- Matrix A is printed (line 150).
- Matrix B is printed (line 170).
- The sum of matrices A and B is found and stored in C (line 180).
- Matrix C is printed (line 200).
- The dimensions for matrix F, a vector, are set and the elements set to zero (line 210).
- The product of matrices C and D is computed and stored in F (line 220).
- The dimensions for matrix H (single value) are specified and the elements set to zero (line 230).
- Finally, the product of matrices G and F is found and stored in H and printed (lines 240, 260).

In the second program, a value N is read that determines the order of the Hilbert matrix segment to be computed, stored, and printed. Next the matrix is inverted and printed. Finally the Hilbert matrix is multiplied by its own inverse, and the resulting product matrix is printed. Notice that line 290 specifies N as equal to 2 to produce the first three matrices of order 2, and later returns to read in the data "3," redimensions to a larger array--larger than 2, but smaller than the original 20--and produces more output.

MATRIX PROGRAM EXAMPLE 1:

```
100 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)
110 READ M,N
120 MAT READ A(M,M),B(M,M),D(M,N),G(N,M)
130 MAT C=ZER(M,M)
140 PRINT "MATRIX A ØF ØRDER";M
150 MAT PRINT A;
160 PRINT "MATRIX B ØF ØRDER";M
170 MAT PRINT B;
180 MAT C=A+B
190 PRINT "       C=A+B"
200 MAT PRINT C;
210 MAT F=ZER(M,N)
220 MAT F=C*D
230 MAT H=ZER(N,N)
240 MAT H=G*F
250 PRINT "   H"
260 MAT PRINT H;
270 DATA 3,1
280 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
999 END
RUN
```

```
MAT-1      15:44

MATRIX A ØF ØRDER 3
 1  2  3

 4  5  6

 7  8  9

MATRIX B ØF ØRDER 3
 9  8  7

 6  5  4

 3  2  1

        C=A+B
 10      10      10

 10      10      10

 10      10      10


   H
 360
```

MATRIX PROGRAM EXAMPLE 2:

```
100 DIM A(20,20),B(20,20),C(20,20)
110 READ N
120 MAT A=CØN(N,N)
130 MAT B=CØN(N,N)
140 MAT C=ZER(N,N)
150 FØR I=1 TØ N
160 FØR J=1 TØ N
170 LET A(I,J)=1/(I+J-1)
180 NEXT J
190 NEXT I
200 PRINT "HILBERT MATRIX ØF ØRDER";N
210 MAT PRINT A;
220 MAT B=INV(A)
230 PRINT "INVERSE ØF HILBERT MATRIX ØF ØRDER";N
240 MAT PRINT B;
250 MAT C=A*B
260 PRINT "HILBERT MATRIX TIMES ITS ØWN INVERSE ØRDER";N
270 MAT PRINT C;
280 GØ TØ 110
290 DATA 2,3
999 END
RUN

MAT-2     15:46

HILBERT MATRIX ØF ØRDER 2
 1  .5

 .5    .333333


INVERSE ØF HILBERT MATRIX ØF ØRDER 2
 4.   -6.

-6.    12.


HILBERT MATRIX TIMES ITS ØWN INVERSE ØRDER 2
 1.    0

-3.72529E-09    1.


HILBERT MATRIX ØF ØRDER 3
 1  .5    .333333

 .5    .333333  .25

 .333333  .25   .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.   -1.78814E-07    0

-2.23517E-08    1.   -5.96046E-08

-1.49012E-08   -1.78814E-07    1


ØUT ØF DATA  IN 110
```

EXAMPLES OF ADVANCED BASIC PROGRAMS

Following are two sample programs illustrating the use of many of the advanced capabilities of BASIC. The first program is developed in an inventory case problem, and makes use of a BCD file. The second program uses a binary file to store personnel information.

Inventory Problem

Mr. Swift, a storekeeper, would like to know how any five items in his store are selling in any given month. He would like a permanent file of the items that were sold each week over a four week period. He wants to update his file at the end of each week, and he may or may not want to get a complete written record of his sales. He may want to get a written report at any time during the month.

The record should consist of an easy to read table listing the items and the number sold in each week. The table should also show the total number of items for each week, the total number of each item sold to date, and the total number of all items sold to date.

The five items that Mr. Swift would like to check are salt, pepper, sugar, nutmeg, and coffee. The month is March.

The following program results from Mr. Swift's requirements. The program is explained by remarks included in it.

```
100 FILES STØCK
110 DIM A(4,3)
120 FØR I=0 TØ 5
130 READ A$(I)
140 NEXT I
150
160 REM  W=INITIAL PASS FLAG, P=PRINTØUT FLAG
170 REM  W=0 INITIAL PASS,    P=0 PRINTØUT DESIRED
180 READ W,P
190 IF W<0 THEN 300
200
210 REM FØR THE INITIAL PASS, WRITE THREE ZERØES.  THERE IS NØ
220 REM DATA INITIALLY, AND SØMETHING MUST BE WRITTEN BEFØRE
230 REM IT CAN BE READ.
240 SCRATCH#1
250 WRITE#1,0;0;0
260 RESTØRE#1
270
280 REM READ IN DATA WRITTEN INTØ THE FILE FRØM PREVIØUS WEEKS.
290 REM X...ITEM, Y...WEEK, A(X,Y)...NUMBERS ØF ITEMS SØLD TØ DATE.
300 READ#1,X,Y,A(X,Y)
310 IF END#1 THEN 350
320 GØ TØ 300
330
340 REM READ DATA FØR THIS WEEK ØR DATA MISSED FRØM PREVIØUS WEEKS.
350 READ X,Y,Z
360 IF X<0 THEN 430
370
380 REM WHEN X IS NEGATIVE, THE DATA READ IS FINISHED.
390 LET A(X,Y)=A(X,Y)+Z
400 GØ TØ 350
410
420 REM WRITE THE UPDATED INFØ TØ THE PERMANENT DATA FILE.
430 SCRATCH#1
440 FØR X=0 TØ 4
450 FØR Y=0 TØ 3
460 WRITE#1,X;Y;A(X,Y);
470 NEXT Y
480 NEXT X
490
500 REM IS A PRINTØUT WANTED?  0...YES   -1...NØ
510 IF P<0 THEN 870
```

```
520
530 REM PRINT THE MØNTH.
540 PRINT A$(5)
550 PRINT
560 PRINT TAB(10);
570 REM PRINT THE CØLUMN HEADER FØR EACH WEEK.
580 FØR I=0 TØ 3
590 PRINT USING 610,I+1,
600 NEXT I
610:   #####
620 PRINT USING 630
630:   TØTALS
640 PRINT
650
660 REM BEGIN TØ GENERATE THE TABLE ØF VALUES.
670 FØR I=0 TØ 4
680 PRINT A$(I);TAB(10);
690 FØR J=0 TØ 3
700 REM SUM ØF EACH ITEM FØR THE ELAPSED WEEKS.
710 LET T(I)=T(I)+A(I,J)
720 REM SUMS FØR EACH WEEK
730 LET S(J)=S(J)+A(I,J)
740 PRINT USING 610,A(I,J),
750 NEXT J
760 PRINT USING 610,T(I)
770 NEXT I
780 PRINT
790 PRINT TAB(10);
800
810 REM PRINT SUBTØTALS FØR EACH WEEK AND THE TØTAL FØR THE PERIØD.
820 FØR K=0 TØ 3
830 PRINT USING 610,S(K),
840 LET S=S+S(K)
850 NEXT K
860 PRINT USING 610,S
870 END
880 REM 0...SALT,  1...PEPPER,  2..SUGAR,  3...NUTMEG,  4...CØFFEE
890 DATA SALT,PEPPER,SUGAR,NUTMEG,CØFFEE,MARCH
1000 DATA -1,0
1010 DATA -1,0,0
```

Suppose that two weeks have passed and Mr. Swift wants a record of his sales to date. He runs the program with data in line 1000 as shown above. The -1 specifies not the first week, and the 0 specifies a printout of the file data. Also he enters data in line 1010 as shown above. The -1 indicates the termination of data, and the two final zeroes are dummy data put in to satisfy the READ. He gets the following results.

```
RUN



SALT         8:39

MARCH

                  1       2       3       4    TØTALS

SALT              3       4       0       0        7
PEPPER            7       5       0       0       12
SUGAR             4       3       0       0        7
NUTMEG            8       7       0       0       15
CØFFEE            2       9       0       0       11

                 24      28       0       0       52
```

At the end of the third week, Mr. Swift wants to make an update. He wants a printout. All he must do is replace line number 1010, as shown below. The BCD data file is updated, and the printout shows the entries for the third week and the resulting changes in the totals.

```
1010 DATA 0,2,7, 1,2,5, 2,2,1, 3,2,5, 4,2,12, -1,0,0
RUN



SALT        14:00

MARCH

                      1        2        3        4     TOTALS

SALT                  3        4        7        0       14
PEPPER                7        5        5        0       17
SUGAR                 4        3        1        0        8
NUTMEG                8        7        5        0       20
COFFEE                2        9       12        0       23

                     24       28       30        0       82
```

Personnel Information

Following is a typical example of the use of binary files. There are two programs. WØRKER establishes the data base in the file INFØ. WØRK1 shows how values can be altered at will.

```
100 FILES INFØ
110 SCRATCH :1
120 READ A$,S,A1,D,S1,V
130 WRITE:1,A$;S;A1;D;S1;V
140 GØ TØ 120
150 DATAGØRDØN,9345,27,0,4,0
160 DATAPLUMMER,10200,30,4,0,5
170 DATAGARANTINØ,8600,22,0,2,7
180 DATATHØMAS,11550,29,3,0,2
190 DATACHENEY,8800,25,0,4,6

RUN

WØRKER     14:06

ØUT ØF DATA  IN 120


100 FILES INFØ
110 READ A,B,C
120 GØSUB 200
130 SET:1,A*8+B
140 READ:1,X
150 LET X=X+C
160 SET:1,LØC(1)-1
170 WRITE:1,X
180 GØ SUB 240
190 STØP
200 PRINTUSING 220
210 PRINT
220:NAME           SALARY  AGE   DEPENDENTS   SICK DAYS   VACATIØN DAYS
230:"            "S######  ###         ###        ####             ####
240 SET:1,A*8+1
250 READ:1,A$,S,A1,D,S1,V
260 PRINTUSING 230,A$,S,A1,D,S1,V
270 RETURN
280
```

```
290 REM DATA FØRMAT --- A-NAME, B-CØDE VALUE,  C-AMØUNT TØ BE ADDED
300 REM  NAMES---- 0-GØRDØN, 1-PLUMMER, 2-GARANTINØ, 3-THØMAS
310 REM            4-CHENEY
320
330 REM CØDE-----  1-NAME, 4-SALARY, 5-AGE, 6-DEPENDENTS, 7-SICK DAYS
340 REM            8-VACATIØN DAYS
350
360 REM  FØR DATA --- 2,4,820 --- ALTER MR. GARANTINØ'S SALARY BY $820
370
380 DATA 2,4,820

RUN


WØRK1      14:11

NAME          SALARY  AGE   DEPENDENTS    SICK DAYS    VACATIØN DAYS

GARANTINØ  $  8600    22        0            2              7
GARANTINØ  $  9420    22        0            2              7
```

The second line of the printout shows the $820 increase in Mr. Garantino's salary.

Appendix A Error Messages

Because most programs under development contain errors, a series of error messages is included in BASIC. Some of the messages are received 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. In the following table, XXX means a line number.

During execution, some messages occur that do not stop execution, but inform you of irregular conditions existing in identified lines of your program. Other messages, however, point out more serious errors that cause execution to stop.

Compilation Errors

MESSAGE	MEANING
CUT PRØGRAM ØR DIMS	Either the program is too long, or the amount of space reserved by the DIM statements is too much, or both. Cut the length of the program, reduce the size of the lists and tables, reduce the length of printed labels, or reduce the number of simple variables.
DIMENSIØN TØØ LARGE IN XXX	The size of a list or table is too large. Make it smaller. Maximum dimension is A(1022).
FILE NØT DEFINED IN XXX	The file reference statement is missing.
FILES NØT FIRST IN XXX	The file reference statement is preceded by an executable statement.
FØR WITHØUT NEXT	A NEXT statement is missing. This message can occur in conjunction with NEXT WITHØUT FØR.
ILLEGAL CØNSTANT IN XXX	A number is out of bounds (> 5.78960E76).
ILLEGAL FØRMULA IN XXX	Perhaps the most common error message. May indicate missing parentheses, illegal variable names, missing multiply signs, illegal numbers, or other errors. Check the statement thoroughly.
ILLEGAL INSTRUCTIØN IN XXX	Other than one of the 26 legal BASIC instructions has been used following the line number.
ILLEGAL NUMBER IN XXX	The line number is of incorrect form or contains more than 5 digits.
ILLEGAL PRØGRAM NAME IN XXX	A program name in a CALL or CHAIN statement is a name with more than 6 characters and is other than a library program name. A program cannot call itself.

MESSAGE	MEANING
ILLEGAL RELATIØN IN XXX	Something is wrong with the relational expression in an IF--THEN statement. Check to see if you have used one of the 6 permissible relational symbols.
ILLEGAL VARIABLE IN XXX	An illegal variable name has been used.
IMAGE TABLE ØVERFLØW	More than 100 image statements have been included in the program.
INCØRRECT FØRMAT IN XXX	The format of the statement is wrong.
NEXT WITHØUT FØR IN XXX	There is an incorrect NEXT statement, perhaps with a wrong variable given. Check also for incorrectly nested FØR statements.
NØ DATA	There is at least one READ statement in the program, but there are no DATA statements.
ØVER 10 SUBRØUTINES	The program tried to call more than 10 different programs.
PRØGRAM NØT SAVED (Program Name)	A program named in a CALL statement is not saved.
PRØGRAM TØØ LØNG IN XXX	The program is too long for the available storage. Cut the size of the program or dimensions.
PRØGRAM WØN'T FIT (Program Name)	The program called is too large to fit in the remaining available space.
REDIMENSIØNED ARRAY IN XXX	An array has previously been dimensioned.
STRING TØØ LØNG IN XXX	A string has more than 15 characters.
TØØ MANY CØNSTANTS IN XXX	A statement contains constants that result in more than 75 different constants in the program. Example: LET A(4) = 1.24; 4 and 1.24 are constants.
TØØ MANY FILES	The number of files referenced caused the program size limit to be exceeded. (At least 8 files will always be allowed.)
TØØ MANY GØTØ'S IN XXX	More than 79 different line number references were made in GØTØ, IF--THEN, GØSUB, or ØN--GØTØ statements.
TØØ MANY LØØPS	There are more than 26 FØR--NEXT combinations in the program.
TØØ MUCH DATA	The program contains more than 1280 numbers, or too many numbers and strings, or too many strings as data.
UNDEFINED FUNCTIØN	A function such as FNF( ) has been used without appearing in a DEF statement. Check for typographical errors.

MESSAGE	MEANING
UNDEFINED IMAGE XXX	A PRINT USING statement references line XXX, but line XXX either does not exist or is not an image statement.
UNDEFINED NUMBER IN XXX	The statement number appearing in a GØTØ, IF--THEN, GØSUB, or ØN--GØTØ statement does not appear as line number in the program.

Execution Errors--Execution Continued

MESSAGE	MEANING
ABSØLUTE VALUE RAISED TØ PØWER IN XXX	A computation of the form (-3)↑2.7 has been attempted. The computer supplies (ABS(-3))↑2.7 and continues. Note: (-3)↑3 is correctly computed to give -27.
ATTEMPT TØ READ A NUMBER AS A STRING VARIABLE	Self explanatory.
DIVISIØN BY ZERØ IN XXX	A division by zero has been attempted. The computer supplies +∞ (about 5.78960E76) and continues running the program.
END ØF FILE IN XXX	An attempt has been made to read data from a BCD file after all data has been read. The file pointer is located at the end of the file. No data is transmitted.
END ØF FILE SPACE IN XXX	After all physical space in a file has been used, an attempt has been made to write into a BCD file, or an attempt has been made to read from or write into a binary file. No data is transmitted.
INPUT DATA NØT IN CØRRECT FØRMAT, RETYPE IT	Self explanatory.
LØG ØF NEGATIVE NUMBER IN XXX	The program has attempted to calculate the logarithm of a negative number. The computer supplies the logarithm of the absolute value and continues.
LØG ØF ZERØ IN XXX	The program has attempted to calculate the logarithm of zero. The computer supplies -5.78960E76 and continues.
ØVERFLØW IN XXX	A number larger than about 5.78960E76 has been generated. The computer supplies ±5.78960E76 and continues running the program.
SQUARE RØØT ØF A NEGATIVE NUMBER IN XXX	The program has attempted to extract the square root of a negative number. The computer supplies the square root of the absolute value and continues.
UNDERFLØW IN XXX	A number smaller in absolute size than about 4.31809E-78 has been generated. The computer supplies zero and continues. In many circumstances, underflow is permissible and may be ignored.

MESSAGE	MEANING
ZERØ TØ A NEGATIVE POWER IN XXX	A computation of the form 0↑(-1) has been attempted. The computer supplies +∞ (about 5.78960E76) and continues.

Execution Errors--Execution Terminated

MESSAGE	MEANING
BAD IMAGE IN XXX	There are syntax errors in the image statement referenced by line number XXX, or an attempt has been made to put numeric data in an alphanumeric field, or alphanumeric data in a numeric field.
CALLS ØR GØSUB NESTED TØØ DEEPLY IN XXX	Too many CALLs or GØSUBs without a RETURN. It may be that subroutines are being left by GØTØ or IF--THEN statements rather than by RETURNs. The program stops.
DATA FILE (LINE XXX) FØRMAT ERRØR	At line XXX of the data file being read, data is not in the required format.
DIMENSION ERROR IN XXX	A dimension inconsistency has occurred in connection with one of the MAT statements. The program stops.
EXPRESSIØN ØUT ØF RANGE	The range of an ØN--GØTØ statement is incorrect. Example: ØN X GØTØ 10,20,30. When the integer value of X is either minus, zero, or greater than 3, the expression is out of range.
FILE NØT BCD	An attempt has been made to do a BCD file read or write on a binary file.
FILE NØT BINARY	An attempt has been made to do a binary file read or write on a BCD file.
FILE(S) NØT SAVED: (File Name)	The files indicated have been referenced but are not saved in your library.
ILLEGAL FILE DESIGNATØR IN XXX	The file designator is less than unity, non-integral, or greater than the number of referenced files.
ILLEGAL PØINTER	The element pointer is less than unity, non-integral, or greater than the length of the referenced file.
NEARLY SINGULAR MATRIX IN XXX	The INV operation in MAT has encountered a matrix with zero or nearly zero pivotal elements. The matrix being inverted is singular or nearly so. Note, however, that this check is not 100 percent reliable. For instance, this message need not occur even if the inverse is meaningless, as with high order Hilbert matrices. If this error occurs, the program stops.
ØUT ØF DATA IN XXX	A READ statement for which there is no DATA has been encountered. If this means a normal end of your program, ignore the message. Otherwise, it means that you haven't supplied enough DATA. In either case, the program stops.

MESSAGE	MEANING
READING BCD IN XXX	An attempt has been made to write into a BCD read mode file. Indicates a logic error or no SCRATCH statement encountered before read mode activity.
RETURN BEFØRE GØSUB ØR CALL IN XXX	A RETURN has been encountered before the first GØSUB or CALL in the program. Note: BASIC does not require the GØSUB to have an earlier statement number--only to execute a GØSUB before executing a RETURN. The program stops.
SUBSCRIPT ERRØR IN XXX	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. The program stops.
WRITING BCD IN XXX	An attempt has been made to read or backspace a BCD write mode file. Indicates a logic error or no RESTØRE statement encountered before read mode activity.

Appendix B Limitations on BASIC

There are some limitations imposed on BASIC by the limited amount of computer storage. Listed below are some of these limitations, in particular, those that are related to the error messages in Appendix A. The reader should realize that although the BASIC language itself is fixed, in time some of these limitations may be relaxed slightly.

ITEM	LIMITATION
Source program size	The source program may not consist of more than 256 lines. It may not contain more than 6144 characters.
Constants	The total number of different constants must not exceed 75.
Data	There can be no more than 1280 data numbers.
FØR statements	There can be no more than 26 FØR statements in a program.
GØ TØ, IF--THEN, GØSUB, and ØN--GØ TØ statements	The total number of different references in these statements cannot exceed 79.
Compiled program size	Cannot exceed 4148 words.
Image statements	Maximum of 100.
Dimension of array	A singly dimensioned array cannot exceed 1022. The limitations on a doubly dimensioned array with dimensions X, Y, are: (1) X cannot exceed 1022, (2) Y cannot exceed 510, and (3) the product of X+1 and Y+1 cannot exceed 2074.
Naming of variables	The variable A is distinct from the element A(0).
Subscripting	Numeric variable names consisting of two characters may not be subscripted.

Appendix C Comparison Order for BASIC Characters

BASIC characters are compared in their BASIC code representations. The following table gives the BASIC code number for each character. Codes of nonprinting characters are enclosed in parentheses.

Code	Character	Code	Character	Code	Character
00	0	24	D	53	$
01	1	25	E	54	*
				(55)	End of Message
02	2	26	F	56	>
03	3	27	G	57	↑
04	4	30	H	60	(space)
05	5	31	I	61	/
		(32)	Bell		
06	6	33	. (period)	62	S
07	7	34	" (quote)	63	T
10	8	35	?	64	U
11	9	36	<	65	V
		(37)	Carriage Ret.		
12	’ (apostrophe)	40	- (minus)	66	W
13	:	41	J	67	X
14	(	42	K	70	Y
15	;	43	L	71	Z
				(72)	Line Feed
16	=	44	M	73	, (comma)
17	\	45	N	74	)
20	+	46	O	75	[
21	A	47	P	76	]
22	B	50	Q	(77)	Fill
23	C	51	R		
		(52)	Tab		

Appendix D Using the Time-Sharing System

The Mark I Time-Sharing System consists of a GE-235 computer with a number of input-output stations, currently Model 33 and Model 35 Teletypes. Those using the input-output stations are able to share the use of the computer with each other so as to suggest that each one has sole use of the computer. The teletypewriters are the devices through which the user communicates with the computer. This appendix contains elementary instructions for using the Time-Sharing System. For complete information, see the Mark I Time-Sharing Service Command System Reference Manual (229116).

The Keyboard

The teletypewriter keyboard is a standard typewriter keyboard for the most part. There are three special keys the user must be familiar with.

RETURN	The RETURN key is located at the right-hand end of the third row of keys, and does more than act as an ordinary carriage return. The computer ignores the line being typed until this key is pushed.
CTRL	The CTRL (control) key is located at the left-hand end of the third row of keys. When it is pressed along with the X key, the computer deletes the entire line being typed. This also acts as a carriage return.
←	The backwards arrow key is the shift of Ø. It is used to delete the character or space immediately preceding the ←. If this key is pressed N times, the N preceding characters or spaces will be deleted.

Examples:

ABCWT←←DE appears as ABCDE when RETURN is pushed.

AB C←←CDE appears as ABCDE when RETURN is pushed.

Some languages available on the Time-Sharing System use the three characters \ , [, and] . They are located on the keys L, K, and M, respectively, when either SHIFT key is pushed.

Teletypewriter Operation

Besides the keyboard itself there are four control buttons necessary to operate the machine.

Button	Location	Function
ØRIG	Leftmost of six small buttons on the right.	Turns on the teletypewriter and connects it to the phone line.
CLR	Next to ØRIG.	Turns off the teletypewriter and disconnects the phone circuit.
LØC LF	Left of the space bar on Model 35 Teletypes only.	Feeds paper to permit tearing it off.

BUZ-RLS	Rightmost of six small buttons on the right.	Turns off the buzzer that signals a low paper supply.

If the teletypewriter is on a direct line to the computer, pushing the ØRIG button is all that is necessary to connect up with the computer. To disconnect from the computer, type GØØDBYE or BYE. If that fails, push CLR.

In order to connect with the computer from a teletypewriter not on a direct line:

- Push the ØRIG button and wait for the dial tone.
- Dial one of the numbers at the Time-Sharing Center.

In order to disconnect, type GØØDBYE or BYE, and if that fails, push CLR.

Control Commands

There are a number of commands that may be given to the computer by typing the command at the start of a new line, with no line number, and following the command with a carriage return. The following table lists some of the most frequently used of these commands.

Command	Meaning
CATALØG	The computer types a list of the names of all the programs currently saved under that user number.
EDIT	Gives a brief explanation of the format used in the EDIT commands.
LENGTH	Gives you an idea of the length of the program, to the nearest 200 characters. The maximum length of one program is 6400 characters.
LIST	Causes an up-to-date listing of the program to be typed out.
LIST--XXXXX	Causes an up-to-date listing of the program to be typed out beginning at line number XXXXX and continuing to the end.
NEW	Erases from working storage the program currently being worked on, and asks for a NEW FILE NAME.
ØLD	Erases from working storage the program currently being worked on, and asks for an ØLD FILE NAME.
RENAME	Permits you to change the name of the program you are currently working on, but does not destroy the program.
RUN	Begins the computation of a program.
RUN (typed during computation)	Gives an indication that a program is running and how much machine time has elapsed since the run began.
SAVE	Saves the program intact for later use. To retrieve a saved program, type ØLD.
SCRATCH	Destroys the program currently being worked on, but leaves the user number and program name intact. It gives you a clean sheet to work on.
STATUS	Gives an indication of the status of the teletypewriter you are using (running, idle, or disconnected).
STØP	Stops the computation at once. It can be typed only when the teletypewriter is not printing.

Command	Meaning
SYSTEM	Permits you to change systems (BASIC, ALGØL, etc.) without going through the sign-on sequence again.
TTY	Supplies the following information: teletypewriter number, user number, language being used, program being used, and status of teletypewriter.
UNSAVE	Erases a saved program from memory. Since the memory of the computer is finite, this command should be used to free space in storage for other users' programs.

INDEX

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

202026B (20M) 2-70