RCA
Information
Systems

SPECTRA 70

TIME SHARING OPERATING SYSTEM (TSOS)

# Interactive BASIC System Information Manual

70-00-518
February 1969

The information contained herein is subject to change without notice. Revisions may be provided to advise of such additions and/or changes.

February 1969

The development of the BASIC Language and the original version of this manual were supported in part by the National Science Foundation under terms of Grant NSF GE 3864 to Dartmouth College. Under this grant Dartmouth Professors John G. Kemeny and Thomas E. Kurtz developed the BASIC Language and a compiler for operation under a time-sharing system in use at Dartmouth.

The printing of this manual by RCA does not necessarily constitute endorsement of RCA products by Dartmouth College.

# CONTENTS

# CONTENTS
(Cont'd)

# 1. WHAT IS A PROGRAM

♦ A program is a set of directions, or a recipe, that is used to tell a computer how to provide an answer to some problem. It usually starts with the given data as the ingredients, contains a set of instructions to be performed or carried out in a certain order, and ends up with a set of answers as the cake. And, as with ordinary cakes, if you make a mistake in your program, you will end up with something else – perhaps hash!

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

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

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

# 2. A BASIC PRIMER

## AN EXAMPLE

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

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

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

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

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

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

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

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

We immediately observe several things about this sample program. First, we see that the program uses only capital letters, since the teletypewriter has only capital letters.

**AN EXAMPLE**
**(Cont'd)**

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

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

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

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

We next go to statement 15, which is a LET statement, and first encounter a formula to be evaluated. (The asterisk "*" is used to denote multiplication.) In this statement we direct the computer to compute the value of AE-BD, and to call the result G. In general, a LET statement directs the computer to set a variable equal to the formula on the right side of the equals sign. We know that if G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 20, if G is equal to zero. If the computer discovers a yes answer to the question, it is directed to go to line 65, where it prints "NO UNIQUE SOLUTION". From this point, it would go to the next statement. But lines 70, 80, and 85 give it no instructions, since DATA statements are not *executed,* and it then goes to line 90 which tells it to "END" the program.

If the answer to the question "Is G equal to zero?" is "no", as it is in this example, the computer goes on to the next statement, in this case 30. (Thus, an IF-THEN tells the computer where to go if the "IF" condition is met, but to go on to the next statement if it is not met.) The computer is

**AN EXAMPLE (Cont'd)**

now directed to read the next two entries from the DATA statements, -7 and 5, (both are in statement 80) and to assign them to C and F respectively. The computer is now ready to solve the system

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

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

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

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

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

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

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

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

and to print out the solutions. It is directed back again to 30, but there are no more pairs of numbers available for C and F in the DATA statements. The computer then informs you that it is out of data, printing on the paper in your teletypewriter "OUT OF DATA" and stops.

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

**AN EXAMPLE (Cont'd)**

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

Another question arises from the seemingly arbitrary placing of the elements of data in the DATA statements: why place them as they have been in the sample program? Here again, the choice is arbitrary and we need only put the numbers in the order that we want them read (the first for A, the second for B, the third for D, the fourth for E, the fifth for C, the sixth for F, the seventh for the next C, etc.). In place of the three statements numbered 70, 80, and 85, we could have put

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

or we could have written, perhaps more naturally,

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

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

The program and the resulting run is shown below exactly as it appears on the teletypewriter:

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

 4            -5.5
 .666667       .166667
-3.66667      3.83333
30 READ A,B,D,E
> OUT OF DATA
```

**AN EXAMPLE (Cont'd)**

After typing the program, we type RUN followed by depressing ETX. Up to this point the computer checks the program but does not execute it. It is this command which directs the computer to execute your program.

**FORMULAS**

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

The computer performs its primary function (that of computation) by evaluating formulas which are supplied in a program. These formulas are very similar to those used in standard mathematical calculation, with the exception that all BASIC formulas must be written on a single line. Five arithmetic operations can be used to write a formula. These are:

| 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 $X^2$). |
| ∧ | X ∧ 2 | Alternative exponentiation. |
| ** | X**2 | Alternative exponentiation. |

We must be careful with parentheses to make sure that we group together those things which we want together. We must also understand the order in which the computer does its work. For example, if we type A + B * C ↑ D, the computer will first raise C to the power D, multiply this result by B, and then add A to the resulting product. This is the same convention as is usual for $A + B*C^D$. If this is not the order intended, then we must use parentheses to indicate a different order. For example, if it is the product of B and C that we want raised to the power D, we must write A + (B * C) ↑ D; or, if we want to multiply A + B by C to the power D, we write (A + B) * C↑D. We could even add A to B, multiply their sum by C, and raise the product to the power D by writing ((A+B) *C)↑D. The order of priorities is summarized in the following rules:

1. The formula inside parentheses is computed before the parenthesized quantity is used in further computations.

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

3. In the absence of parentheses in a formula involving only multiplication and division, the operations are performed from left to right, even as they are read. So also does the computer perform addition and subtraction from left to right.

**FORMULAS (Cont'd)**

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

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

| Functions | Interpretation |
|---|---|
| SIN (X) | Find the sine of X |
| COS (X) | Find the cosine of X |
| TAN (X) | Find the tangent of X |
| ATN (X) | Find the arctangent of X |
| | X interpreted as a number, or as an angle measured in radians |
| EXP (X) | Find $e^X$ |
| LOG (X) | Find the natural logarithm of X (ln X) |
| ABS (X) | Find the absolute value of X (\|X\|) |
| SQR (X) | Find the square root of X ($\sqrt{X}$) |

Two other mathematical functions are also available in BASIC: INT and RND; these are explained in Section 3. In place of X, we may substitute any formula or any number in parentheses following any of these formulas. For example, we may ask the computer to find $\sqrt{4 + X^3}$ by writing SQR (4 +X↑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 that we understand how to write numbers for the computer and what variables are allowed.

**Numbers**

♦ A *number* may be positive or negative and it may contain up to six digits, but it must be expressed in decimal form. For example, all of the following are numbers in BASIC: 2, -3.675, 123456, - .654321, and 83.4156. The following are *not* numbers in BASIC: 14/3, $\sqrt{7}$, and .00123456789. The first two are *formulas,* but not numbers, and the last one has more than six digits. We may ask 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 further flexibility by use of the letter E, which stands for *times ten to the power.* Thus, we may write .00123456 in a form acceptable to the computer in any of several forms: .123456E-2 or 123456E-11 or 1234.56E-6. We may write ten million as 1E7 and 1965 as 1.965E3. We do not write E-7 as a number, but must write 1E7 to indicate that it is 1 that is multiplied by $10^7$.

Actually, numbers up to 15 digits long are acceptable to BASIC; the number 12.3456789 is acceptable, but is effectively rounded to 12.34568.

**Variables**

◆ A *variable* in BASIC is denoted by any letter, or by any letter followed by a single digit. Thus, the computer will interpret E7 as a variable, along with A, X, N5, I0, and O1. A variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program was written. Variables are given or assigned values by LET, READ, or INPUT statements. The value so assigned will not change until the next time a LET, READ, or INPUT statement is encountered with a value for that variable.

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. Any run-time error, however, results in program termination.

Six other mathematical symbols are provided for in BASIC, symbols of relation, and these are used in IF-THEN statements where it is necessary to compare values. An example of the use of these relation symbols was given in the sample program in Section 1. Any of the following six standard relations may be used:

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

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

```
10  PRINT 1, SQR (1)
20  PRINT 2, SQR (2)
30  PRINT 3, SQR (3)
    . . . . . .

99  PRINT 99, SQR (99)
100  PRINT 100, SQR (100)
101  END
```

**LOOPS (Cont'd)**

With the following program, using one type of loop, we can obtain the same table with far fewer lines of instruction, 5 instead of 101:

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

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

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

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

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

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

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

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

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

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

**LOOPS (Cont'd)**

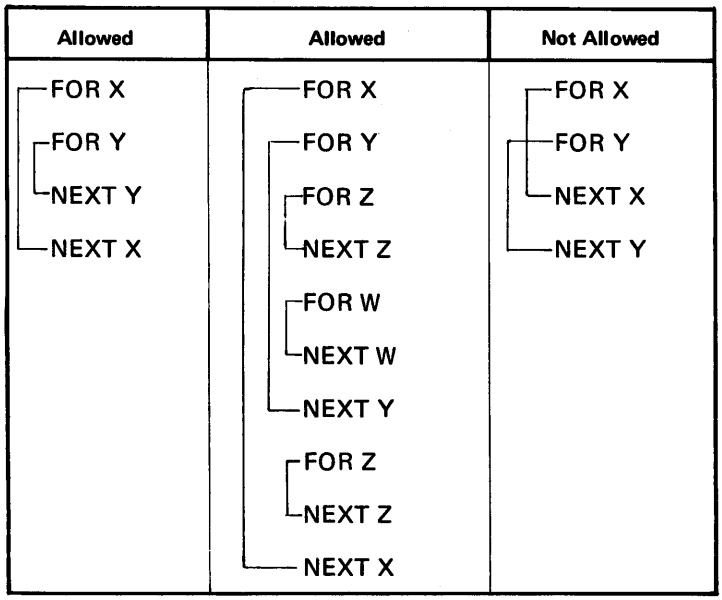
More complicated FOR statements are allowed. The initial value, the final value, and the step size may all be formulas of any complexity. For example, if N and Z have been specified earlier in the program, we could write

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

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

If the initial value of the step-size is greater than the final value specified in the FOR statement, the compiler terminates the program with a diagnostic statement. The same is true when the initial value is less than a final step-size which is negative.

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



## LISTS AND TABLES

♦ In addition to the ordinary variables used by BASIC, there are variables which can be used to designate the elements of a list or of a table. These are used where we might ordinarily use a subscript or a double subscript, for example the coefficients of a polynomial ($a_0$, $a_1$, $a_2$, . . . ) or the elements of a matrix ($b_{i,j}$). The variables which we use in BASIC consist of a single letter, which we call the name of the list, followed by the subscripts in parentheses. Thus, we might write A(0), A(1), A(2), etc. for the coefficients of the polynomial and B(1, 1), B(1,2), etc. for the elements of the matrix.

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

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

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

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

Statements 20 and 60 could have been eliminated by writing 30 as FOR I = 1 to 15, but the form as typed would allow for the lengthening of the list by changing only statement 60, so long as it did not exceed 25.

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

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

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

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

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

**LISTS AND TABLES (Cont'd)**

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

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

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

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

```
NEW PROGRAM NAME--SALES1
READY

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

**LISTS AND TABLES (Cont'd)**

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

**USE OF THE TIME-SHARING SYSTEM**

♦ Now that we know something about writing a program in BASIC, how do we set about using a teletypewriter to type in our program and then to have the computer solve our problem?

Sitting down at the teletypewriter, you first push the button labeled ORIG. This turns on the teletypewriter. You wait for the dial tone and then dial the computer number. The computer answers with a "BEEP" tone. The computer will then type %PLEASE LOGON and a / on the next two lines. You are to type in LOGON and your IDENTIFICATION. Press the ETX (Control C) key. (You must push the ETX (Control C) key after typing any line – only then does your line enter the computer.)

The computer will type a slash (/) – and you should type EXECUTE BASIC before hitting the ETX (Control C) key next.

The computer then types NEW OR OLD and an asterisk, and in response you type the appropriate adjective: NEW if you are about to type a new problem and OLD if you want to recover a problem on which you have been working earlier and have stored in the computer's memory.

The computer then asks NEW PROGRAM NAME – (or OLD PROGRAM NAME, as the case may be) and you type any combination of letters and digits you like that start with a letter, but no more than eight. In the sample problem preceding you will remember that we named it SALES1. If you are recalling an old problem from the computer's memory, you must use exactly the same name as that which you gave the problem before you asked the computer to save it.

The computer then types READY and an *, then you should begin to type your program. Make sure that each line begins with a line number which contains no more than five digits and contains no non-digit characters. Also be sure to number each line and to press the ETX key at the completion of each line.

If, in the process of typing a statement, you make a typing error and notice it immediately, you can correct it by pressing the backward arrow (shift key above the letter O). This will delete that which is in the preceding space, and you can then type in the correct character. Pressing this key a number of times will erase from this line the characters in that number of preceding spaces. The carriage return, RETURN key will delete the entire line being typed.

**USE OF THE TIME-SHARING SYSTEM (Cont'd)**

After typing your complete program, you type RUN and press the ETX key. The computer will then analyze your program. If the program is one which the computer can run, it will then run it and type out any results for which you have asked in your PRINT statements. This does not mean that your program is correct, but that it has no errors of the type known as *execution* or *run-time* errors as opposed to *grammatical* errors. If it has errors of this type, the computer will type an error message (or several error messages).

If you are given an error message, informing you of an error in line 60, for example, you can correct this by typing a new line 60 with the correct statement. If you want to eliminate the statement on line 110 from your program, you can do this by typing 110 and then the ETX key. If you want to insert a statement between those on lines 60 and 70, you can do this by giving it a line number between 60 and 70.

After you have all of the information you want, and are ready to leave the teletypewriter, you should type BYE. The computer then returns control to the TSOS Executive and responds with a slash. You may then LOGOFF to receive the amount of time you used and free the terminal for use by others.

A sample use of the time-sharing system is shown below.

```
%B001 PLEASE LOGON.
/LOGON FCZ
%B002 LOGON ACCEPTED AT 1046 ON 11/19/68, TSN 0093
ASSIGNED.
/EXEC BASIC
%L001 PROGRAM LOADING
 NEW OR OLD
 *NEW
 NEW PROGRAM NAME--CTEST
 READY
 *10 FOR N=1,7
 ?10 FOR N=1TO 7
 *20 PRINT N,SQR(N)
 *30 NEXT N
 *40 PRINT "DONE"
 *50 END
 *RUN
 1             1
 2             1.41421
 3             1.73205
 4             2
 5             2.23607
 6             2.44949
 7             2.64575
DONE
```

## ERRORS AND DEBUGGING

♦ It may occasionally happen that the first run of a new problem will be free of errors and give the correct answers. But it is much more common that errors will be present and will have to be corrected. Errors are of three types: errors of form (or grammatical errors) and run-time errors, both of which prevent the running of the program, and logical errors in the program, which cause the computer to produce wrong answers or no answers at all.

Grammatical errors cause the system to respond with a question mark (?) followed by a copy of the incorrect statement up to, but not including, the first character in error. In the previous example, line 10 was typed as:

```
*10 FOR N=1,7
```

Because the comma is not permitted in a FOR statement, the system responded:

```
?10 FOR N=1
```

The error must be corrected. In this case the remainder of the correct source statement is typed and the line

```
?10 FOR N=1 TO 7
```

is successfully processed by BASIC.

Logical errors are often harder to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines, or by deleting lines from the program. As indicated in the last section, a line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those of two existing lines; and a line is deleted by typing its line number and pressing the ETX key. Notice that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers will start out using line numbers that are multiples of five or ten, but that is a matter of choice.

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

As with most problems in computing, we can best illustrate the process of finding the errors (or *bugs*) in a program, and correcting (or *debugging*) it, by an example. Let us consider the problem of finding that value of X between 0 and 3 for which the sine of X is a maximum, and ask the machine to print out this value of X and the value of its sine. If you have studied trigonometry, you know that $\pi/2$ is the correct value; but we shall use the computer to test successive values of X from 0 to 3, first using intervals of .1, then of .01, and finally of .001. Thus, we shall ask the computer to find the sine of 0, of .1, of .2, of .3 . . . ., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. It will do it by testing SIN (0) and SIN (.1) to see which is larger, and calling the larger of these two

**ERRORS AND DEBUGGING (Cont'd)**

numbers M. Then it will pick the larger of M and SIN (.2) and call *it* M. This number will be checked against SIN (.3), and so on down the line. Each time a larger value of M is found, the value of X is *remembered* in X0. When it finishes, M will have been assigned to the largest of the 31 sines, and X0 will be the argument that produced that largest value. It will then repeat the search, this time checking the 301 numbers 0, .01, .02, .03, . . . , 2.98, 2.99, and 3, finding the sine of each and checking to see which sine is the largest. Lastly, it will check the 3001 numbers 0, .001, .002, .003, . . . ., 2.998, 2.999, and 3, to find which has the largest sine. At the end of each of these three searches, we want the computer to print three numbers: the value X0 which has the largest sine, the sine of that number, and the interval of search.

Before going to the teletypewriter, we write a program and let us assume that it is the following:

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

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

```
NEW OR OLD
*NEW
NEW PROGRAM NAME--MAXSIN
READY
*10 READ D
*20 LWR X0=0
?20 L

      ET X0=0
```

After typing line 20, BASIC notices that LET was mistyped in line 20; so we retype it, this time correctly, retyping only the incorrect characters.

## ERRORS AND DEBUGGING (Cont'd)

```
*FOR X=0 TO 3 STEP D
?30 FOR X=0 TO 3 STEP D
*40 IF SIN(X)<=M THEN 100
*50 LET XO=X
*60 LET M=SIN(X)
*70 PRINT XO,X,D
*80 NEXT XO
*90 GO TO 20
*100 DATA .1, .01,.001
*110 W_END
*RUN
    80 NEXT XO
  >ILLEGAL FOR-NEXT NESTING
 >FOR BUT NO NEXT,LINES     30
*40 IF SIN(X) < = M THEN 80
*80 NEXT X
```

Notice the use of the underline to erase a character in line 110, which should have started with E.

The next 3 error messages relate to lines 30 and 80, where we see that we mixed variables. This is corrected by changing line 80.

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

We try to RUN again.

```
*RUN
.1              .1              .1
.2              .2              .1
.3              .3              .1
.4              .4              .1
.5              .5              .1
.6              .6              .1
.7              .7              .1
.8              .8              .1
.9              .9              .1
1               1               .1
1.1             1.1             .1
```

This is obviously incorrect. Because we are having every value of X printed, we direct the machine to stop printing by hitting the BREAK key while it is running. We ponder the program for a while, trying to figure out what is wrong with it. We notice that SIN (0) is compared with M on the first time through the loop, but we had assigned no value to M. So we wonder if giving a value less than the maximum value of the sine will do it, say -1

Because we hit the BREAK to stop the PRINT loop, control was returned to the TSOS Executive which responded with a slash. If we had not been in a PRINT loop, the ESCAPE key would also have returned control to the Executive.

## ERRORS AND DEBUGGING (Cont'd)

```
/INTR
 MAXSIN...
 STOP AT 00040,
 END OR CONTINUE--END
 *20 LET M=-1
 *RUN
 0              0              .1
 .1             .1             .1
 .2             .2             .1
 .3             .3             .1
 .4             .4             .1
 .5             .5             .1
 .6             .6             .1
 .7             .7             .1
 .8             .8             .1
 .9             .9             .1


 BREAK (hit BREAK key)
 /R

*70
*85  PRINT XO, M, D
*RUN

1.6             .999574        .1
1.6             .999574        .1
1.6             .999574        .1
1.6


BREAK (hit BREAK key)
/R

*90  GO TO 10
*5   PRINT "X VALUE", "SIN",
"RESOLUTION"
*RUN
```

The Executive Command INTR will return control to BASIC, reporting the status of our program before behaving like the RESUME command, whereas the Executive command RESUME will return control to BASIC and respond only with an asterisk, permitting modifications to our program.

We first return control to BASIC using the INTR command, which reports the status of our program, and in turn, offers us the choice of continuing or ending the execution of MAXSIN. We respond END and BASIC responds with an *.

Had we responded CONT, BASIC would have continued printing our results.

We see that we initialized X0 instead of M in line 20, so we change line 20 to give an initial value to M.

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 that it goes through the loop.

We fix this by moving the PRINT statement outside the loop. Typing 70 ETX deletes that line, and line 85 is outside of loop. We also realize that we want M printed and not X.

We see that we are performing the same operation (the case for D=.1) over and over again.

So we stop it and inspect the program again. Of course, line 90 sent us back to line 20 to repeat the operation and not back to line 10 to pick up a new value for D. We also decide to put in headings for our columns by a PRINT statement.

**ERRORS AND DEBUGGING (Cont'd)**

```
 X VALUE        SINE        RESOLUTION
  1.6           .999574      .1
  1.57          1.           .01
  1.571         1.           .001
  10 READ D
>OUT OF DATA
```

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

```
*LIST

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

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

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

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

## SUMMARY OF ELEMENTARY BASIC STATEMENTS

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

### LET

◆ This statement is not a statement of algebraic equality, but is rather a command to the computer to perform certain computations and to assign the answer to a certain variable. Each LET statement is of the form: LET[variable] = [formula]. LET is optional.

*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 takes all of the DATA statements in the order in which they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, 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 occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they occur in the correct order. A common practice is to collect all DATA statements and place them just before the END statement.

Each READ statement is of the form: READ [sequence of variables] and each DATA statement of the form: DATA [sequence of numbers].

*Examples:*

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

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

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

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

**PRINT**

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

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

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

Examples of type A:

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

**PRINT** *(Cont'd)*

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

Examples of type b:

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

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

Examples of type c:

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

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

Example of type d:

```
250 PRINT
```

The computer will advance the paper in a teletypewriter one line when it encounters this command.

**GO TO**

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

*Example:*

```
150    GO TO 75
```

**IF – THEN**

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

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

**IF – THEN** *(Cont'd)*

*Examples:*

```
 40 IF SIN (X) < = M THEN 80
 20 IF G = 0 THEN 65
112 IF A$="YES" THEN 175
224 IF C$=D$ THEN 400
```

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

**FOR and NEXT**

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

FOR [variable] = [formula TO formula STEP formula]

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

*Examples:*

```
 30  FOR X = 0 TO 3 STEP D
 80  NEXT X

120  FOR X4 = (17 + COS (Z))/3 TO 3*SQR (10) STEP 1/4
235  NEXT X4

240  FOR X = 8 TO 3 STEP -1

456  FOR J = -3 TO 12 STEP 2
```

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

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

If you write 50 for Z = 2 TO -2, without a negative step size, the body of the loop will not be performed and the computer will terminate program preparation.

**DIM**

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

**DIM**
*(Cont'd)*

*Examples:*

```
20 DIM H (35)
35 DIM Q (5,25)
```

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

**END or STOP**

◆ The STOP and END statements result in termination of program execution. A TSOS BASIC program need not include a terminal END statement, since one will automatically be supplied by the compiler; however, several END statements may be included, each of which can terminate the program.

The STOP statement results in the following being printed to the terminal.

STOP AT line-number, END OR CONT - -

If the user responds "END", then execution is terminated. Any other response indicates execution is to be continued at the next sequential line number.

*Example:*

```
/EXEC BASIC
%LOOl PROGRAM LOADING
 NEW OR OLD
 *NEW
 NEW PROGRAM NAME--EYU
 READY
 *10 PRINT "THIS IS A TEST PROGRAM"
 *20 STOP
 *RUN
THIS IS A TEST PROGRAM
 STOP AT 00020, END OR CONT--END
```

The STOP is useful where we want to RUN part of our program. STOP, examine the results, then RUN more of the program.

*Example:*

```
 *30 PRINT "THIS IS ANOTHER LINE"
 *40 STOP
 *RUN
THIS IS A TEST PROGRAM
 STOP AT 00020, END OR CONT--CONT
THIS IS ANOTHER LINE
 STOP AT 00040, END OR CONT--END
 *
```

# 3. ADVANCED BASIC

## MORE ABOUT PRINT

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

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

Shorter zones can be manufactured by use of the semicolon; and the zones are four spaces long for 1-digit numbers, six spaces long for 2-digit, 3-digit, and 4-digit numbers; and nine spaces long for 5-digit and 6-digit numbers. Numbers that cannot be represented as 6 or less digits are represented in E-notation and occupy either 11 or 12 characters; these numbers are printed in 15 space long zones.

For example, if you were to type the program

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

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

```
20  PRINT I,
```

you would have the numbers printed in the zones, reading

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

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

```
20  PRINT I;
```

and the result would be printed

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

You should remember that a label inside quotation marks is printed just as it appears and also that the end of a PRINT signals a new line, unless a comma or semicolon is the last symbol.

**MORE ABOUT PRINT (Cont'd)**

Thus, the instruction

```
40  PRINT S,T
```

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

```
40  PRINT S,T
```

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

*Examples:*

```
   10 LET S=12
   20 LET T=24
   30 LET R=36
   40 PRINT S,T,
   50 PRINT R
   60 END
*RUN
12             24             36
*
```

```
   10 LET S=12
   20 LET T=24
   30 LET R=36
   40 PRINT S,T
   50 PRINT R
   60 END
*RUN
12             24
36
*
```

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

```
250  PRINT
```

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

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

**MORE ABOUT PRINT (Cont'd)**

This 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), etc., if there were room. Line 140 directs the teletypewriter, after printing the B(1,1) value corresponding to M = 1, to start a new line and to do the same thing after printing the value of B(2,2) corresponding to M = 2, etc.

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

1. If a number is an integer, the decimal point is not printed. If the integer contains more than six digits, the teletypewriter will give you (a) a decimal point followed by the first digit, (b) the next five digits, and (c) an E followed by the appropriate integer and sign. For example, it will take 32,437,580,259 and write it as .324376E+11.
2. For any decimal number, no more than six significant digits are printed.
3. Trailing zeros after the decimal point are not printed. The following program, in which we print out the first 45 powers of 2, shows how numbers are printed.

```
READY
*10 FOR I=1 TO 45
*20 PRINT 2**I
*30 NEXT I
*40 END
*RUN
 2
 4
 8
 16
 32
 64
 128
 256
 512
 1024
 2048
 4096
/RESUME
 *20 PRINT 2**I;
 *RUN
 2  4  8  16   32   64   128  256  512  1024  2048  4096  8192
 16384   32768   65536   131072  262144  524288  .104858E+07
 .209715E+07   .419431E+07   .838861E+07   .167772E+08   .335544E+08
 .671089E+08   .134218E+09   .268436E+09   .536871E+09   .107374E+10
 .214749E+10   .429497E+10   .858994E+10   .171799E+11   .343598E+11
 .687195E+11   .137439E+12   .274878E+12   .549756E+12   .109951E+13
 .219903E+13   .439805E+13   .87961E+13  .175922E+14   .351844E+14
```

## FUNCTIONS

♦ There are two functions which were listed in Section 2 but not described. These are INT and RND.

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

One use of the INT function is to round numbers. We may use it to round to the nearest integer by asking for 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 this will indeed do the rounding guaranteed for it (it will round a number midway between two integers up to the larger of the integers).

It can also be used to round to any specific number of decimal places. For example, 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 round X correct to D decimal places.

The function RND produces a random number between 0 and 1. The form of RND requires an argument and so we write RND(X) or RND (Z).

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

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


.188369     .869783     .858106E-01   .657089     .285174
.375162     .182469     .528736       .185368     .129168
.556681     .497407     .542784E-02   .490766     .655323E-02
.543786     .678303     .361059E-01   .678639     .603213
```

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

20 PRINT INT (10*RND(X))

and we would then obtain

```
1 8 0 6 2 3 1 5 1 1 5 4 0 4 0 5 6 0 6 6
*
```

**FUNCTIONS**
**(Cont'd)**

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

```
 20 PRINT INT(9*RND(X)+1);
*RUN
2  8  1  6  3  4  2  5  2  2  6  5  1  5  1  5  7  1  7  6
*
```

or we can obtain random numbers which are the integers from 5 to 24 inclusive by changing line 20 as in the following example:

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

8 22  6 18  10  12  8 15  8 7 16  14  5 14  5 15
18  5 18  17
```

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

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

If you were to run the first program of this section again, you would get the same twenty numbers in the same order. But we can get a different set by "throwing away" a certain number of the random numbers. For example, 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 will see, by comparing this with the first program, that the first ten of these random numbers are the second ten of the earlier program.

```
*10 FOR I = 1 TO 10
*20 LET Y = RND (X)
*30 NEXT I
*40 FOR I = 1 TO 20
*50 PRINT RND(X),
*60 NEXT I
*70 END
*RUN

.556681   .497407   .542784E-02   .490766       .655323E-02
.543786   .678303   .361059E-01   .678639       .603213
.272974   .933245   .943764       .93821        .876867
.294905   .338766   .313263       .715332E-01   .751434
*
```

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

**FUNCTIONS (Cont'd)**

The handiness of such a function can be seen in a program where you frequently need the function $e^{-x^2}$. You would introduce the function by the line

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

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

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

The use of DEF is generally limited to those 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 the program. For these functions, the GOSUB statement may frequently be useful, and it is described next.

**GOSUB AND RETURN**

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

```
90 GOSUB 210
```

directs the computer to jump to line 210, the first line of the subroutine. The last line of the subroutine should be a return command directing the computer to return to the earlier part of the program. For example,

```
350 RETURN
```

will tell the computer to go back to the first line numbered greater than 90 and to continue the program there.

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

**GOSUB AND RETURN (Cont'd)**

You may use a GOSUB inside a subroutine to perform yet another subroutine. This would be called "nested GOSUBs". In any case, it is absolutely necessary that a subroutine be left only with a RETURN statement, using a GOTO or an IF-THEN to get out of a subroutine will not work properly. You may have several RETURNs in the subroutine so long as exactly one of them will be used.

The user must be very careful not to write a program in which a GOSUB appears inside a subroutine which refers to one of the subroutines already entered. (Recursion to 15 levels is allowed.)

```
*10  PRINT " A", " B", " C", "GCD"
*20  READ A, B, C
*30  LET X = A
*40  LET Y = B
*50  GOSUB 200
*60  LET X = G
*70  LET Y = C
*80  GOSUB 200
*90  PRINT A, B, C, G
*100 GO TO 20
*110 DATA 60, 90, 120
*120 DATA 38456, 64872, 98765
*130 DATA 32, 384, 72
*200 LET Q = INT(X/Y)
*210 LET R = X - Q*Y
*220 IF R = 0 THEN 300
*230 LET X = Y
*240 LET Y = R
*250 GO TO 200
*300 LET G = Y
*310 RETURN
*320 END
*RUN
```

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

**INPUT**

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

```
40 INPUT X, Y
```

**INPUT (Cont'd)**

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

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

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

and the machine will type out

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

Without the semicolon at the end of the line 20, the question mark would have been printed on the next line.

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

If excessive data is entered, it is ignored; however, data must be entered according to the type of variable in the INPUT statement. When the data required is numeric, for example, then the submission of alphabetic or special characters causes a run-time error. There is one exception to this; if the first four characters of input are STOP, then program execution is terminated.

**SOME MISCELLANEOUS STATEMENTS**

◆ Two other BASIC statements that may be useful from time to time are REM and RESTORE.

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

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


200  REM  THIS IS A SUBROUTINE FOR SOLVING EQUATIONS
     .....
300  RETURN
     .....
520  GOSUB 200
```

## SOME MISCELLANEOUS STATEMENTS (Cont'd)

Sometimes it is necessary to use the data in a program more than once. The RESTORE statement permits reading the data as many additional times as it is used. Whenever RESTORE is encountered in a program, the computer restores the data block pointer to the first number. A subsequent READ statement will then start reading the data all over again. A word of warning – if the desired data are preceded by code numbers or parameters, superfluous READ statements should be used to pass over these numbers. As an example, the following program portion reads the data, restores the data block to its original state, and reads the data again. Note the use of line 57( to "pass over" the value of N, which is already known.

```
100  READ N
110  FOR I = 1 TO N
120     READ X
     .....
200  NEXT I
     .....
560  RESTORE
570  READ X
580  FOR I = 1 TO N
590     READ X
     .....
```

# 4. EXTENSIONS TO BASIC

## MATRIX OPERATIONS

♦ Although you can work out for yourself programs which involve matrix computations, there is a special set of eleven instructions for such computations. They are identified by the fact that each instruction must start with the word 'MAT'.

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

The following is a list of available matrix commands. Use of each of the commands is described in detail in Section 2.

| | |
|---|---|
| MAT READ A,B,C, | Read the matrices, A,B,C, their dimensions having been previously specified. Data is read in row-wise sequence. |
| MAT PRINT A,B;C, | Print the matrices A,B,C, with A and C in the regular format, but B closely packed. |
| MAT C = A + B | Add the two matrices A and B and store the result in matrix C. |
| MAT C = A - B | Subtract the matrix B from the matrix A and store the result in matrix C. |
| MAT C = A * B | Multiply the matrix A by the matrix B and store the result in matrix C. |
| MAT C = INV(A) | Invert the matrix A and store resulting matrix in C. |
| MAT C = TRN(A) | Transpose the matrix A and store the resulting matrix in C. |
| MAT C = (K) * A | Multiply the matrix A by the value represented by K. K may be either a number or an expression, but in either case it must be enclosed in parentheses. |
| MAT C = CON | Set each element of matrix C to one. (CON=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, yielding an identity matrix. |
| MAT A = B | Equivalent to MAT A = 1*B. |

**MAT READ and MAT PRINT**

♦ Data may be read into or printed from a matrix without having to reference each element of the matrix individually by using the MAT READ and MAT PRINT commands.

*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 (i.e., $A_{1,1}, A_{1,2}, \ldots, A_{1,m}, A_{2,1}, A_{2,2}, \ldots, A_{2,m}, \ldots, A_{n,m}$).

Information is read from DATA statements until the matrix array is completely filled. Partial matrices may not 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
```

L is defined as a 2 by 3 matrix and M as a 2 by 2 matrix (line number 110). The MAT READ statement reads from the DATA statement located at line number 500 in row order. The matrix element, $L_{2,2}$ is recomputed at line number 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**

♦ Matrices can be added, subtracted and multiplied using the matrix arithmetic commands. The matrix dimensions must be conformable for each operation. If dimensions are not conformable, execution is terminated and you receive a dimension error message.

The matrix arithmetic statements may take three forms. They are:

| | | |
|---|---|---|
| MAT C = A+B | or | MAT P = Q*R |
| MAT J = G–P | | MAT A = B |

**Matrix Addition, Subtraction, and Multiplication** *(Cont'd)*

Only one operation may be performed per statement.

*Example:*

```
612 MAT H=A*B

615 MAT H=H+E

618 MAT H=H-K
```

**Scalar Multiplication**

- A matrix can be multiplied by a scalar expression using the command:

  MAT X = (expression) *D

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

*Examples:*

```
10 MAT F=(2)*G

50 MAT Q=(2.33+M)*Q

75 MAT B=(N)*A
```

**Identity Matrix**

- An identity matrix is defined by the statement:

  MAT B = IDN

  or MAT R = IDN (expression, expression)

In the first statement, the matrix variable 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:*

```
 90 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)
```

**Assignment**

- Matrix assignment is accomplished by:

```
255 MAT A = B
```

**Matrix Transposition**

- Matrices are transposed using the form:

  MAY Y = TRN(Z)

where Y and Z are both matrices. The matrix Z transpose will replace matrix Y. Y and Z must conform for transposition.

*Examples:*

```
300 MAT G=TRN(H)

400 MAT U=TRN(V)
```

Matrix transposition in place (MAT A = TRN(A)) is not allowed.

**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 for inversion.

*Examples:*

```
550 MAT K=INV(L)

560 MAT A=INV(B)
```

Matrix inversion in place (MAT A=INV(A)) is not allowed. If a matrix is singular, you will receive the message ALMOST SINGULAR MATRIX.

**Matrix Zer and Con 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 CONstant function is used to set all elements of a matrix to one. As an example

MAT C = CON

will set all elements of matrix C to one.

**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. This 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)

**Dimensioning** *(Cont'd)*

The DIM statement defines both a P and Q matrix. P is defined as a 12 element matrix, and Q is defined 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).

Prior to 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 themselves are used to accomplish this dimensioning. They are:

```
MAT READ C(M,N)
MAT C = ZER(M,N)
MAT C = CON(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 instructions may be used to redimension a matrix during running. A matrix may be redimensioned to either a larger or a smaller matrix provided the new dimensions do not require more storage space than was originally reserved by the DIM statement. To illustrate, consider the following statements:

```
10 DIM A(8,8),B(8,8),C(8,8)

50 MAT READ A(2,2),B(2,2)

60 MAT C = ZER(2,2)

            .
            .
            .

100 MAT A = IDN(8,8)

110 MAT READ B(4,4),C(4,4)
```

Observe 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 two rows and two columns.

The MAT READ also reads the number of values required by the dimensions into the storage which was reserved by the DIM statement. The MAT READ reads the values in row-wise sequence. In the initial MAT READ, the elements in the order read are A(1,1), A(1,2), A(2,1), A(2,2), B(1,1), B(1,2), B(2,1), and B(2,2). Statement 60 illustrates the ZER being used to specify dimensions and to zero the elements of the matrix C. Statements 100 and 110 illustrate the redimensioning concept, where matrix A is redimensioned as an eight row, eight column identity matrix and matrices B and C are redimensioned as four row, four column matrices into which data is to be read.

**Dimensioning** *(Cont'd)*

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

**Example**

◆ The following program illustrates some of the capabilities of the MAT instructions.

```
  100 DIM A(2,2),B(2,2),C(2,2)
  110 DIM D(2,2)
  120 MAT A=IDN
  130 MAT B=ZER
  140 MAT C=CON
  150 MAT READ D
  160 DATA 1,2,3,4
  170 MAT PRINT A;B;C;D
  180 DIM E(2,2)
  190 MAT E=A+D
  200 MAT PRINT E
  210 MAT E=B-D
  220 MAT PRINT E
  230 MAT E=C*D
  240 MAT PRINT E
  250 MAT E=INV(A)
  260 MAT PRINT E
  270 MAT E=TRN(A)
  280 MAT PRINT E
*RUN
1  0
0  1
0  0
0  0
1  1
1  1
1              2
3              4
2              2
3              5
-1            -2
-3            -4
 4             6
 4             6
 1             0
 0             1
 1             0
 0             1
*BYE
```

## ALPHANUMERIC DATA AND STRING MANIPULATION

♦ Alphanumeric data, names, and other identifying information can be handled in the BASIC language using string variables. You can input, store, compare and output alphanumeric and certain special characters in the Spectra 70 character set.

A STRING is any sequence of alphanumeric and certain special characters in the Spectra 70 character set not used for control purposes in TSOS.

STRING size is limited to 15 valid characters.

A STRING VARIABLE is denoted by a letter followed by a "$". For example: A$,B$,X$.

### The DIM Statement

♦ Strings can be set up as one-dimensional arrays only. Requests for two-dimensional arrays are not allowed and when encountered will initiate an error comment.

*Examples:*

```
10 DIM A(5),C$(20),A$(12),D(10,5)

20 DIM R$(35)

30 DIM M$(15),B$(15)
```

In statement 10, only C$ and A$ are string variables. R$, as dimensioned in statement 20, will set aside space in core for 35, 15 character arrays. Any or all of these strings may be less than 15 characters.

### The LET Statement

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

*Example:*

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

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

*Example:*

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

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

### The IF--THEN Statement

♦ Only one string variable is allowed on each side of the IF–THEN relation. All of the six standard relations (=,<>,><,>=,<=) are valid. When strings of different lengths are compared, the shorter string is taken to be the lesser of the two.

**The PRINT Statement**

♦ The PRINT statement also can contain string variables intermixed with ordinary BASIC variables. When a string variable is encountered which has not been assigned, the PRINT statement will produce no print out. A semicolon after a string variable in a PRINT statement causes the string to be printed and the variable following that string to be directly connected to it.

*Examples:*

```
35 PRINT A,16,B$,C$;N

40 PRINT 100+I,"DATA",L$;M$;N$

50 PRINT S$
```

**COMPUTED GO TO STATEMENT**

♦ The computed GO TO statement is included in BASIC, providing a multi-branched switch. The form of the statement is:

ON expression GO TO $ln_1$,$ln_2$ , . . .

where:

*expression* is a valid BASIC expression.

$ln_1$, $ln_2$, . . . is a sequence of line numbers to which the statement will transfer depending on the expression value.

For example:

```
ON X+Y GO TO 575, 490, 650
```

The above statement will transfer control to 575, 490, or 650 respectively, depending upon whether the value of the expression X+Y yields 1, 2, or 3 respectively.

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

**MULTIPLE VARIABLE REPLACEMENT**

♦ The LET statement permits multiple variable replacement.

For example:

```
LET X=Y=Z=21*N/2
```

The statement places the value of the expression "21*N/2" in variables X, Y, and Z. Any valid expression may be used.

**PRINT FUNCTION TAB**

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

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

The statement will cause the print head to move over to the Nth position after printing the value of X and to the (2*N)th position after printing the value of Y.

The use of the comma and semi-colon remains unchanged in the PRINT statement. Thus, when a comma follows a variable in a PRINT statement, a fixed field width is reserved before the next entry in the statement is recognized. The semi-colon causes this field width to be minimized. Thus, when the teletype is being tabbed, the semi-colon should be used.

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

All arguments of TAB are modulo 75. Teletypewriter print positions are assumed to run 0 through 74.

```
LIST
 100 X=1
 110 ON X GOTO 119,180,200
 119 PRINT "00000000011111111112222222223333333333"
 120 PRINT "12345678901234567890123456789012345678901"
 130 PRINT TAB(10);1
 140 PRINT TAB(20);1
 150 PRINT TAB(30);1
 160 PRINT TAB(8);1:TAB(18);1;TAB(28);1
  *RUN
 00000000011111111112222222223333333333
 12345678901234567890123456789012345678901
          1
                    1
                              1
        1         1         1
  *
```

**FUNCTION SGN**

♦ The function SGN (argument) yields +1, -1, or 0 depending upon the value of the argument. The following table describes the options:

| Function | Argument Value | Yield |
|---|---|---|
| SGN | Zero | 0 |
| SGN | positive, non-zero | +1 |
| SGN | negative, non-zero | -1 |

*Examples:*

```
SGN (0) yields 0

SGN (-1.82) yields -1

SGN (989) yields +1

SGN (-.001) yields -1

SGN (-0) yields 0
```

**FUNCTION RND**

♦ The function RND(x) is a psuedo random number generator designated as follows:

(a) If $x > 0$, then RND(x) is always the same function of x.

(b) If $x < 0$, the system supplies a different arbitrary random number between 0 and 1.

(c) If $x = 0$, the system supplies a pseudo random number which is a function of the previous random number generated by RND. If $x = 0$ the first time RND is called in a program, the system will supply a fixed number.

To generate a sequence of pseudo random numbers, the user would call any of these options followed by repeated calls to option (c).

# 5. ENHANCEMENTS TO BASIC

## FUNCTION TIM

◆ The function TIM(X) is available in BASIC. The function TIM effectively tells the user the amount of time required to run his program. The variable X, although a dummy argument, is required. The value of TIM(X) is processor time, in seconds, used since the RUN command was given.

*Example:*

```
25 PRINT "TIME="; TIM(X)
```

## DATA FILE OPERATIONS

### FILES Statement

◆ Files are referenced according to the order in which they appear in the FILES statement. The first filename in the first FILES statement is associated with the file designator, #1. In the following example, file B would be referenced by the file designator #2, a third file by #3, and a fourth file by #4.

If more than four files are referenced, a diagnostic would be printed.

*Example:*

```
 NEW
NEW PROGRAM NAME--READ
READY
*10 FILES A;B
*20 FOR I=1 TO 10
*30 READ#1,A(I)
*40 NEXT I
*35 PRINT A(I)
*RUN
1
2
3
4
5
6
7
8
9
10
*
```

**FILES Statement** *(Cont'd)*

Had the data file A not been previously created using the WRITE statement, the computer would have responded:

```
*RUN
  30  READ #1,A(I)
 > FILE IN WRITE MODE
*
```

Files are initially in the input mode after the program has been compiled, provided the files have been previously created. If BASIC is creating a file for the first time, the file is automatically opened in the output mode.

```
 LIST
   10 FILES A;B
   20 SCRATCH #1
   30 FOR I=1 TO 10
   35 INPUT A(I)
   40 WRITE #1,A(I)
   50 NEXT I
   60 END
*RUN
```

**Data–WRITE Statement**

♦ The WRITE statement results in data values being written to the referenced file. The data values may be either expressions or specific Hollerith strings. Hollerith strings must be limited to 15 characters.

**Data–SCRATCH Statement**

♦ The SCRATCH statement does the following:

1. Any data in the file is erased and the file mode is set to output.
2. The EOF (end of file) condition for this file is reset.

**File–READ Statement**

♦ The READ statement for files results in variables being assigned values from the data file referenced by the file-designator. If the file is exhausted of data (EOF condition raised for this file), a WHEN statement must be used to:

1. Prevent a diagnostic to be printed and prevent BASIC from terminating the program.
2. Allow the user to continue his program by going to a specified line number.

**WHEN Statement**

◆ The WHEN statement is used in conjunction with the FILE-READ statement as described above.

If no WHEN statement referencing this file has been executed, a diagnostic is printed and execution terminates.

*Example:*

```
 LIST
READY
*LIST
  100 FILES A
  110 FOR I=1 TO 10
  120 READ #1,A(I)
  130 PRINT A(I)
  140 NEXT I
*
```

If a WHEN statement referencing this file has been executed, then control passes to the line-number specified in the WHEN statement.

Any attempt to read a file while the EOF condition is raised for this file results in a diagnostic being printed and execution being terminated.

*Example:*

```
 100 FILES A
 110 FOR I=1 TO 10
 120 READ #1,A(I)
 130 PRINT A(I)
 140 NEXT I

 110 FOR I=1 TO 12
*RUN
1
2
3
4
5
6
7
8
9
10
    120 READ #1,A(I)
 >  SUBSCRIPT VALUE EXCEEDS BOUNDS
*10 DIM A(12)
```

**WHEN Statement** *(Cont'd)*

```
*RUN
1
2
3
4
5
6
7
8
9
10
    170 READ #1,A(I)
 >  EOF ENCOUNTERED
*
```

The WHEN statement contains the line number to be branched to if the EOF condition is raised for this file. The line-number is saved in a table and remains in effect until another WHEN statement referencing the same file is executed. Note that the WHEN statement is global. It need only be issued at the beginning of the program, and need be reissued only if the "EOF" line-number is to be changed.

*Example:*

65 WHEN EOF#1 GO TO 200

The WHEN statement may also be used to test for a run-time error to avoid program termination.

Whenever a run time error is detected, a diagnostic is printed and execution is terminated. If a when statement has been previously executed, the diagnostic is printed and execution continues at the specified line-number. This facility should aid debugging.

*Note:*

The statement in which the error occurred has not gone to completion. Consequently, results of this statement are unpredictable.

*Example:*

```
    0 REM THIS IS AN EXAMPLE OF WHEN ERROR
   10 WHEN ERROR GOTO 100
   20 A=5
   30 B=123E45
   40 A=B*B
   50 PRINT " END "
   60 END
  100 REM ERROR ROUTINE
  110 PRINT "A=";A, "B=";B
*run
    40   A=B*B
 > OVERFLOW
A= 5           B= .123E+48
```

**RESTORE Statement**

- The RESTORE statement causes the following:

  The file pointer to be reset to the first item in the file.

  The file mode to be set to input.

  The EOF condition for this file to be reset.

The RESTORE statement must be used prior to reading files that have been written in the same program.

*Example:*

```
*RES
READY
*LIST
  100 FILES A
  110 SCRATCH #1
  120 FOR I=1 TO 10
  130 WRITE #1,I
  140 NEXT I
  150 RESTORE #1
  160 FOR I=1 TO 10
  170 READ #1,A(I)
  180 PRINT A(I)
  190 NEXT I
*RUN
1
2
3
4
5
6
7
8
9
10
```

# 6. EDITING COMMANDS

♦ The RCA Spectra 70 BASIC editing commands provide the user with additional program preparation facilities. Each command may be entered when BASIC has responded with an asterisk (*). An abbreviated command, consisting of the first three characters of the complete command name, can be used for all commands. Commands may not be entered with line numbers. Thus, each editing command is executed immediately upon entry. The following editing commands are available with Spectra 70 BASIC.

## NEW

♦ The NEW command erases the current contents of the user's work space. The system responds:

NEW PROGRAM NAME –

requesting the user to supply a name for the program to be constructed. The system responds:

READY

when a satisfactory name has been supplied. This name is referred to as 'the program name'.

## OLD

♦ The OLD command erases the current contents of the user's work space. The system responds:

OLD PROGRAM NAME –

requesting the user to supply the name of a program which he has previously saved in his library. If a correct name is supplied, that program is loaded into the user's work space, its name becomes the program name, and the system responds:

READY

Otherwise, the user is informed that the program he named does not exist in his library.

User 1 may load User 2's BASIC program (provided the catalog entry specifies SHARE=YES) by specifying the program name as follows:

$user2.BAS.filename

When a file is being read from the user's library, every statement is syntax checked as it is read. If a syntax error is discovered, the entire statement is printed to the terminal and is not entered in the program.

*Note:*

When the user logs on and requests TSOS BASIC, he is immediately prompted to enter either of the commands NEW or OLD by the system response

NEW OR OLD?

**OLD (Cont'd)**

Program construction (modification of an old program) may not begin until the work space is given a name. If the user does not issue the old or new command, NEW is assumed and the program takes on the default name "NONAME".

**RENAME or REN**

♦ The RENAME (or REN) command allows the user to rename the program currently contained in his work space without erasing the work space. The system will respond:

NEW PROGRAM NAME –

requesting the user to supply the new name.

**SCRATCH or SCR**

♦ The SCRATCH (or SCR) command erases the current contents of the user's work space but retains the program name.

**LENGTH or LEN**

♦ The LENGTH (or LEN) command prints, on the user's terminal, the total amount of space, in bytes, occupied by the program currently in his work space. The value is rounded to the closest multiple of 100 bytes.

**STATUS or STA**

♦ The STATUS (or STA) command prints, on the user's terminal:

1. The program name.
2. The current date and time of day.
3. The amount of CPU time used since logon.

**SAVE or SAV**

♦ The SAVE (or SAV) command is used to save, in the user's library, a copy of the source program currently contained in his work space. The system first checks to see if a previous (BASIC) program with the current program name has been saved. If one has, the system responds:

OVERWRITE PREVIOUS FILES (YES,NO)?

Otherwise, the program in the user's work space is saved in his library and cataloged using the current program name.

**UNSAVE or UNS**

♦ The UNSAVE (or UNS) command erases the catalog entries for the files named, and releases the corresponding library space used for saving the named programs.

*Format:*

UNS[AVE] file-name $\{$,file-name$\}_0^n$

**CATALOG or CAT**

♦ The CATALOG (or CAT) command prints a list of the names of all (BASIC) programs previously saved on the user's library.

**BYE**

♦ The BYE command returns control to the TSOS Executive.

**RUN**

♦ The RUN command directs the system to compile and execute the program contained in the user's work space.

**LIST or LIS**

♦ The LIST (or LIS) command directs the system to print, on the user's terminal, the sequence of lines referenced. If no line number list is given, the entire program is printed. Lines will be reformatted by the system.

When creating a paper tape, the following steps are advised:

1. Place the terminal in T mode and create a header of nulls (control, shift, P) or rubouts.

2. Place the terminal in K mode and enter the List command without typing ETX.

3. Place the terminal in KT mode and type XOFF,ETX.

When using this tape for input, the first input line will be null and BASIC will reply with a question mark. However, the tape is now correctly positioned for the first input line of the program.

*Format:*

LIS[T] [line-number-list]

**DELETE or DEL**

♦ The DELETE (or DEL) command directs the system to delete, from the user's work space, the sequence of lines referenced.

*Format:*

DEL[ETE] [line-number-list]

*Example:*

DEL 120, 160-210

The line numbered 120 and all lines numbered 160 to 210, inclusive, are deleted.

A single line can also be deleted by typing its line number followed by ETX (no text). If no line number list is specified, the DELETE command is equivalent to SCRATCH (the entire program is deleted).

**EXTRACT or EXT**

♦ The EXTRACT (or EXT) command directs the system to extract, from the user's work space, the sequence of lines referenced. All other lines of source text are automatically deleted.

If no line number list is specified, the EXTRACT command is ignored (the entire program is extracted).

*Format:*

EXT[RACT] [line-number-list]

**RESEQUENCE or RES**

♦ The RESEQUENCE (or RES) command directs the system to renumber the sequence of lines referenced, using the range of line numbers specified. Note that only a single sequence of lines may be resequenced.

*Format:*

RES[EQUENCE] [line-number {-|TO} line-number]
[AS range]

*Example:*

RES 300-350 AS 1000(10)

The lines numbered 300 to 350, inclusive, are renumbered 1000, 1010, etc., andthen deleted from the program.

The system adjusts all references to the resequenced line numbers.

If the "AS range" option is not specified, the implied range is 100(10).

If no line number sequence is specified, the entire program is resequenced.

**DUPLICATE or DUP**

♦ The DUPLICATE (or DUP) command directs the system to duplicate the sequence of lines referenced, using the range of line numbers specified. The duplicated lines are not deleted from the program.

The system does not adjust any line number references.

If the "AS range" option is not specified, the implied range is 100(10).

If no line number sequence is specified, the entire program is duplicated using the implied range 100(10).

*Format:*

DUP[LICATE] [line-number {-|TO} line-number
[AS range]]

**MERGE or MER**

♦ The MERGE (or MER) command directs the system to merge a specified sequence of the program previously saved under the given file name into the program contained in the user's work space.

*Fomat:*

MER[GE] file-name [Δline-number {–|TO} line-number]
[AS range]

If the named file cannot be found on the user's library space, the system responds:

EITHER FILENAME INCORRECT OR FILE NOT CATALOGED.
REENTER COMMAND.

If the sequence option "line-number {–|TO} line-number" is omitted, the entire saved program is merged.

If the "AS range" option is specified, the saved program is loaded from the library, resequenced using the given range values, and then merged with the program in the user's work space.

*Example:*

MERGE DATABASE AS 900(1)

The program "DATABASE" is resequenced using line numbers 900,901,. . ., and then merged with the current work space program.

*Note:*

A TSOS BASIC program consists of any sequence of BASIC statements. In particular, a program could consist of all DATA statements, hence providing a facility for constructing data files.

If the "AS range" option is not specified, the saved program is not resequenced prior to the merge operation. The user should beware of conflicting line numbers in this situation.

*Note:*

If a source statement having the same line number as a previous statement, is entered into the user's work space (either from the terminal or a saved file), the new statement replaces the old statement.

**WEAVE or WEA**

♦ The WEAVE (or WEA) command causes two or more files to be merged. For example:

WEA[VE] file-name $\{$,file-name$\}_{0}^{n}$

is equivalent to:

```
MERGE file-name1
MERGE file-name2
      .
      .
      .
MERGE file-namen
```

**SYNCHK or SYN**

♦ The SYNCHK (or SYN) command causes all source input from saved files to be syntax checked. This is the default case in BASIC.

This command is used in connection with the OLD, MERGE and WEAVE commands.

**NOSYNCHK or NOS**

♦ The NOSYNCHK (or NOS) command causes source input from saved files to bypass the syntax checker and be entered directly into the user's work space. This command should be issued before the command which accesses the disk. It greatly increases the speed in which source files are brought into work space.

This command is used in connection with the OLD, MERGE and WEAVE commands.

**CAUTION:**

This command should be used only with files which have been saved by the BASIC system. BASIC writes these files in a specified format and any other format may cause BASIC to work incorrectly. When BASIC writes a file, a guard character is placed in each line. If this guard character is not present during input, the mode is automatically changed to SYNCHK and the current input line plus all future input lines will be syntax checked.

APPENDIX A

# COMPILER DIAGNOSTICS

| Message | Description |
|---|---|
| FUNCTION PREVIOUSLY DEFINED | A user defined function (DEF statement) has been multiply defined. |
| ARRAY PREVIOUSLY DIMENSIONED | An array (matrix) has been multiply dimensioned (DIM statement). |
| NO SUCH LINE # | Reference is made (GOTO, IF, ON statement) to a non-existent line number. |
| FOR NESTING (MAX=7) | The maximum level of nesting of FOR-loops is 7. |
| NESTED FOR'S WITH SAME INDEX | The construction:<br>FOR I = 1 to 10<br>.<br>.<br>.<br>FOR I = 2 to 6<br>} no NEXT I<br>is illegal. |
| NEXT BEFORE FOR | The matching NEXT statement must follow its corresponding FOR statement (in the logical sequence of statements). |
| ILLEGAL FOR-NEXT NESTING | FOR-loops may be nested, but they must not overlap.<br>e.g.<br>FOR I = 1 to 10<br>FOR J = 1 to 10<br>.<br>.<br>.<br>NEXT I |
| 31 CONSTANTS ALLOWED PER EXPR | The maximum number of constants allowed in any arithmetic expression is 31. |
| OVERFLOW | A numeric constant exceeds the maximum single-precision floating-point value (about $10^{75}$). |
| UNDERFLOW | A numeric constant is smaller than the minimum single-precision floating-point value (about $10^{-78}$). |
| ILLEGAL OPERATION | Any of<br>MATA = A*B<br>MATA = B*A<br>MATA = INV(A)<br>MATA = TRN(A) |
| MEMORY EXCEEDED | The generated object code exceeds 16 pages (65,536 bytes) in size. |

## COMPILER DIAGNOSTICS (Cont'd)

| Message | Description |
|---|---|
| 4 DATA FILES MAX | Only four files can be referenced in a BASIC program. |
| FILENAME > 8 CHARS | A filename is a letter followed by at most 7 alphanumeric characters. |
| DATA FILE DEFINED TWICE | There can be only one file-designator associated with each data file. |

APPENDIX B

# POST-COMPILATION DIAGNOSTICS

| Message | Description |
|---|---|
| READ OR RESTORE, BUT NO DATA | The user has implied the need for DATA elements, but none were supplied. |
| FOR BUT NO NEXT, LINES a, b, c | The program contains dangling FOR statements in lines numbered a, b, and c. |
| UNDIMENSIONED MATRICES: x, y, z | Variables x, y, and z were used as matrices but not explicitly dimensioned (DIM statement). |
| USED AS VECTOR AND ARRAY: x, y, z | Variables x, y, and z were used as both vectors (1-dimensional) and arrays (2-dimensional). |
| UNDEFINED FUNCTIONS: x, y, z | The functions x, y, z were not defined. |
| MEMORY EXCEEDED | The object code generated, plus array storage allocation (which caused the error) exceeds 16 pages. |
| NO DATA FILES DEFINED | Data files have been referenced, but no FILES statement appears in program. |
| filename – PASSWORD PROTECTED | Enter the password via the TSOS password command. |
| filename – OPEN ERROR | The file could not be opened for some reason. Retry. |
| filename – LOCKED | Two BASIC users are referencing the same file, one in read mode and the other in write mode. The first person to access the file gets it and locks it to other users. Several users may read a file simultaneously, but only one user at a time can write the file. |
| filename – CATALOG ERROR | Space could not be obtained for the file. Retry. |

APPENDIX C

# RUN-TIME (EXECUTION) DIAGNOSTICS

| Message | Description |
|---|---|
| INVALID INDEX | A number is converted to integer for use as an index (subscript or ON expression) and its value does not lie in the range $0 \leqslant x < 2^{16}$. |
| VALUE OUTSIDE RANGE | The value of an ON expression exceeds the number of line-numbers in the ON statement. |
| SUBSCRIPT VALUE EXCEEDS RANGE | The value of a subscript exceeds the upper bound declared for an array. |
| GOSUB NESTING (MAX=15) | The maximum level of nesting for subroutine calls is 15. |
| RETURN BEFORE GOSUB | A RETURN statement is executed prior to a GOSUB (subroutine call). |
| INVALID FOR PARAMETERS | On initial entry to a FOR-loop the condition (final value – initial value) * step $> 0$ is not satisfied. |
| OVERFLOW | A number exceeding the maximum single-precision floating-point value during expression evaluation. |
| UNDERFLOW | A number smaller than the minimum single-precision floating-point value during expression evaluation. |
| DIVISION BY ZERO | Division by zero was specified. |
| INVALID EXPONENT | A**B, where B $< 0$. |
| EXP (LARGE NOS.) | A specific case of numeric overflow. |
| LOG (–X) | Logarithm for a negative number was specified. |
| SQR (–X) | Square root of a negative number was specified. |
| 0**0 | Zero to a zero power was specified. |
| 0**(–X) | Zero to a negative power was specified. |
| FUNCTION NESTING (MAX=8) | The maximum level of nesting for (defined) function calls is 8. INT(RND(0)) is a nested function of level 2. |
| ERROR IN SIN-COS ROUTINE | Overflow and underflow resulting from expression evaluation. |

## RUN-TIME (EXECUTION) DIAGNOSTICS (Cont'd)

| Message | Description |
|---|---|
| NON-MATRIX | MAT operation uses a vector or array whose row or column bound is 0. |
| NON-SQUARE MATRIX | The matrix addition, subtraction, multiplication, inversion, transposition and scalar multiplication operations does not fulfill the obvious conformities of matrix dimensions. |
| CURRENT DIM>MAX. DIM | The current (active) bound of a matrix exceeds the maximum specified bound.<br>e.g.<br>DIM A(5,5)<br>MAT A=IDN(6,6) |
| ALMOST SINGULAR MATRIX | Detected during matrix inversion. |
| OUT OF DATA | The set of DATA elements has been exhausted and a READ statement is executed. |
| ILLEGAL CONSTANT | A string (numeric) DATA element is read into a numeric (string) variable. |
| INVALID NUMERIC DATA | Invalid numeric constant read in during execution of an INPUT statement. |
| INVALID FILE DESIGNATOR | A file designator has value < 1 or greater than the number of data files in the program. |
| FILE IN READ MODE | Scratch-statement must be issued before writing a file. |
| FILE IN WRITE MODE | Restore-statement must be issued before reading a file which has just been written. |
| INVALID DATA ON FILE | Trying to read alphanumeric data into numeric variable or vice versa or bad data on file. |
| ALPHA STRING > 15 CHAR | A write statement contains a Hollerith string > 15 characters. |

APPENDIX D

# SYSTEM ERROR DIAGNOSTICS

| Message | Description |
|---|---|
| **System Diagnostics** | |
| INPUT LINE TOO LONG, REENTER | Either the input line (counting all carriage returns, backspaces, and non-BASIC characters) exceeded 88 characters, or the input line after editing exceeded 80 characters. Reenter the line within the above limits. |
| SYSTEM ERROR. REENTER | An error has occurred either in BASIC or the TSOS Executive. If reentry fails, save all relevant data and consult your RCA representative. |
| OUT OF SRC TXT MEM. TRY A SAVE,OLD,UNSAVE OF THIS PROGRAM | You have run out of virtual memory space for your program. The last line entered was not accepted. If you have been editing your source program, try saving it on disc and recalling it. If the above message is printed again, then save your program on paper tape and try to run it later. |
| "machine" INTERRUPT AT L'XXXXXX' | An error has occurred within BASIC or the TSOS Executive. Please save all relevant data and consult your RCA representative. |
| **Editing Command Diagnostics** | |
| ERROR DURING CATALOGING, REENTER | Basic could not obtain space for this file. Some reasons for this are:<br>1. A write password has been placed on the file.<br>2. A hardware error occurred.<br>Retry the SAVE command. |
| LINE NUMBER > 5 DIGITS | During DUP, RES, or MER, a line number has exceeded 5 digits. List the file and make appropriate corrections. |
| INCORRECT FORMAT | An incorrect format was specified for an editing command. Correct and reenter. |
| SYSTEM ERROR. REENTER | Same as system monitor message. |
| EXTRACT COMMAND REQUIRES INCREASING LINE-NUMBERS | The extract command must have the line number list in numerically ascending order. Reenter the command. |

## SYSTEM ERROR DIAGNOSTICS (Cont'd)

| Message | Description |
|---|---|
| **Editing Command Diagnostics** *(Cont'd)* | |
| INCREMENT < 1 | The increment in the AS option of DUP, RES and MER must be a positive integer. |
| EITHER FILENAME INCORRECT OR FILE NOT CATALOGED. REENTER COMMAND | The file specified does not exist. Check the filename and if incorrect reenter the command. |
| ENTER PASSWORD VIA TSOS PASSWORD COMMAND. REENTER COMMAND | This file has been protected by a password. If the user does not know the password, he cannot access the file. If he knows the password, then he must escape to the TSOS Executive and use the password command to enter the password before he can access the file. |
| ERROR DURING FILE OPENING. REENTER | An error occurred while accessing the file. Reenter the command. |
| AN ERROR OCCURRED WHILE PROCESSING COMMAND. REENTER | Some error occurred while processing the command. Try again. |
| PROGRAM NAME INCORRECT, REENTER COMMAND | The name specified is not syntactically correct. Reenter the command. |
| OUT OF DISK SPACE | Space could not be obtained for the file. The users public space allotment is exhausted or the system has become saturated. |

APPENDIX E

# SAVED FILE ORGANIZATION

♦ All files saved by BASIC have the qualifier BAS. prefixed to the name supplied by the user before the filename is cataloged. Thus the BASIC file JOE is cataloged and known in the TSOS system as BAS.JOE. The program is saved in source program form in the same format used by the LIST command. As BASIC files are being read in (via the OLD, MERGE, or WEAVE commands), every line is syntax checked just as if it were being read from a terminal. If an error is discovered, the line is rejected and is written to the terminal preceded by a question mark. The line is *not* entered into the program file and the user must wait for the READY message before he can reenter such lines from the terminal. Thus the user must be extremely careful when processing BASIC programs created by other TSOS products.

The following paragraphs assume the user has some knowledge of DMS (Data Management Systems, see 79-00-614).

All BASIC files are created under SAM (Sequential Access Method). Variable length blocked format is used with a blocksize of 1070 bytes. Initially BASIC requests two tracks for the file and depends on the secondary space allocation feature of DMS to obtain more tracks as needed. The BASIC file JOE would be cataloged and allocated as follows:

```
/CATALOG    FILENAM=BAS.JOE,SHARE=NO,ACCESS=WRITE
/ALLOCATE   FILENAM=BAS.JOE,FORG=SEQ,SPACE=
  (TRACK,2,1)
```

BASIC files have a retention period of 30 days.

Only the user who created the file can access it. This user can read and write to the file and no protection passwords (read or write) are needed to access the file.

The above default attributes can be changed in several ways by the user:

1. The user can catalog and allocate space for the file using the TSOS Executive before saving to the file.

2. After the file has been saved, the user can issue a catalog command in the STATE=UPDATE mode to modify the catalog entries.

3. The user can issue a file command before the file is written with LINK=BASIC to select a specific device on which to write the file.

## SAVED FILE ORGANIZATION (Cont'd)

Several examples should clarify the above ideas:

1. User 1 wants to make BASIC file PETE sharable and wants to associate the read password C'1234' with it. He could do this in two ways:

   a. Before issuing the BASIC SAVE command he would escape to the TSOS Executive and issue the catalog and allocate commands as follows:

      /CAT FILENAM=BAS.PETE,SHARE=YES,RDPASS=
      C'1234'

      /ALLOC FILENAM=BAS.PETE,FORG=SEQ,SPACE=
      (TRACK,2,1)

      *Note:*

      When cataloging a file with STATE=NEW (default), the user must also specify the allocate command. When BASIC knows a file has been cataloged, BASIC assumes space has been allocated for the file. If the user does not allocate space, an error will occur during open processing.

   b. After issuing the BASIC SAVE command he would escape to the executive and issue the catalog command.

      /CAT FILENAM=BAS.PETE,STATE=UPDATE,SHARE=
      YES,RDPASS=C'1234'

   Another user would access this file as follows:

   /PASSWORD C'1234"
   /EXEC BASIC
   NEW OR OLD
   * OLD
   OLD PROGRAM NAME – $USER1.BAS.PETE

   Note that the password is specified in the TSOS password command and that BASIC allows a fully qualified name only when the $USERID option is specified. If a $ is the first character of the filename, the name is *not* syntax checked.

2. User 2 wants to make BASIC file MARK reside on the private disk pack BAS001. He could do this in two ways. Before saving the file he would escape to the executive and type in either of the following:

   a. /CAT FILENAM=BAS.MARK

      /ALLOC FILENAM=BAS.MARK,FORG=SEQ,SPACE=
      TRACK,2,1),DEVICE=D564,VOLUME=BAS001

   b. /FILE LINK=BASIC,FILENAM=BAS.MARK,FORG=SEQ,
      SPACE=(TRACK,2,1),DEVICE=D564,VOLUME=BAS001

**SAVED FILE ORGANIZATION (Cont'd)**

3. User 3 wants to make BASIC file BOB reside on the private tape volume 000149. In addition, he wants the retention period of the file to be 60 days. Before saving the file, he would escape to the executive and enter the following file command:

   /FILE LINK=BASIC,FILENAM=BAS.BOB,DEVICE=TAPE,
   VOLUME=000149,RETPD=60

   The user would have to issue the same command to read the tape during another logon session.

   *Note:*

   If the user has used the file command with LINK=BASIC, he should issue the TSOS command

   /RELEASE BASIC

   before reading or writing to another BASIC file.

The above examples are intended to give a flavor of what can be done with the executive commands. Many other combinations are possible.

## APPENDIX F

# GLOSSARY

*Alphameric characters.* A generic term for ALPHA-betic characters, nuMERIC digits and special characters.

*Argument.* A variable upon whose value the value of a function depends. The arguments of a function are listed in parentheses after the function name, whenever that function is used. The computations specified by the function definition occur using the variables specified as arguments.

*Arithmetic statement.* A type of BASIC statement that specifies a numerical computation; a LET statement.

*Constant.* A quantity that does not change either from one execution of a program to another, or during execution of that program; a number that remains fixed.

*Line error.* An error in the transmission of data over telephone lines.

*Debugging.* Process of locating errors in a program and correcting them.

*Editing statement.* A command to the system to do something with a program. It is not retained as part of the program.

*Expression.* A series of constants, variables, and functions which may be connected by operation symbols and punctuated by parentheses, if required, to cause a desired computation. Another word for expression is *FORMULA.*

*Integer number.* A number without any decimal point which can be generated with the INT function.

*Library.* A collection of user-written programs stored within the system usually on disc.

*Line number.* A number assigned by the user by which each statement can be identified. It is associated with a single BASIC statement and by which reference may be made to that statement.

*List.* A string of items, written in a meaningful format, which designate quantities to be transmitted for input/output.

*Loop.* Repeated execution of a portion of a program.

*Magnitude.* The size of a quantity as distinct from its sign. Thus, +10 and -10 have the same magnitude.

*Operators.* Characters that designate mathematical operations, such as +, -, etc.

*Program.* A set of instructions that will direct a computer in performing certain specified operations.

*Program statement.* An instruction that becomes a part of a program.

*Real number.* A number with a decimal point. A floating-point number.

*Routine.* A logical section of a program. A sequence of program statements. A Subroutine.

*Statement.* An instruction to the computer to perform some sequence of operations.

*System.* The total system consisting of the computer and the control system program under which user programs operate. User programs are prepared with supporting components which behave as sub-systems.

*Terminal.* A data communications device through which commands, programs, and data are transmitted.

# GLOSSARY
**(Cont'd)**

*Time sharing.* The simultaneous access and use of a single computer system by multiple users.

*Transfer.* To terminate one sequence of instructions and begin another sequence.

*Transfer statement.* Any instruction causing a transfer, whether conditional or not. A branch statement.

*Truncation.* Shortening of a number by dropping digits without rounding. The result is that portion of the number preceding the decimal point.

*Variable.* A symbol whose numeric value changes from one iteration of a program to the next or within each iteration of a program.