N O R D B A S I C

Reference Manual

N O R D B A S I C

Reference Manual

REVISION RECORD

Revision	Notes
1/73	Original Printing
1/75	Total Revision

Publ. No. ND-60.040.02
January 1975

A/S NORSK DATA-ELEKTRONIKK
Lørenveien 57, Oslo 5 - Tlf.: 21 73 71

TABLE OF CONTENTS

--ooOoo--

--ooOoo--

STATEMENTS, COMMANDS AND FUNCTIONS LISTED ALPHABETICALLY

1 INTRODUCTION

1.1 What is a Computer?

A computer is a very simple and at the same time a very complex machine. On the one hand, it merely follows elementary instructions to carry out such simple tasks as adding two numbers or determining if a given number is negative. These simple tasks also include "looking" at the next character in a string of alphabetic characters and other nonnumeric activities.

On the other hand, a modern electronic digital computer must be surrounded by a number of storage devices and input-output mechanisms which supply it with tasks to perform, store the results of its computations, and present these results in a convenient form for evaluation or future use. A computer performs its work so fast that these peripheral devices are needed to correlate the many tasks the computer is capable of performing.

1.2 What is a Program?

As noted above, a computer merely carries out simple instructions, albeit at very high speeds. It works so quickly that human beings cannot be directly involved in making more than a small fraction of the decisions that arise in carrying out a complicated task, so that almost all situations must be contemplated in advance. Also, in most cases the bulk of the data upon which the calculations are made must be accurately prepared in advance and entered into the computer so that the calculations may proceed at full speed without having to wait for more data. Thus, a set of instructions for performing a task and the relevant data must be prepared in advance and supplied to the computer. The set of instructions for carrying out a task is called a program. One can think of a program as being a recipe for coming up with the solution to a problem, given the data.

Any mistakes in a program render it just about useless. As with recipes for baking cakes, program errors are of two types. First, one can have errors of form or grammar. These would include misspellings and punctuation. Second, one can have substantive errors even though the form is correct. In the case of recipes for baking cakes, misspelling and typographical errors are examples of errors of form; some of these may make the recipe unreadable. An example of a substantive error would be a direction to use baking soda instead of baking powder.

Since a computer has much less intelligence or common sense than a human being, programs for it must adhere strictly to rules of form or grammar. These rules are particularly complicated for the language that the physical equipment of the computer is constructed to obey. This language is called machine language, and its difficult nature has led computer specialists to invent other more easily used languages that can be converted or translated to machine language.

1.3 What is BASIC?

One such language which is easy to learn and to use is BASIC. BASIC was first developed in 1963-64 at Dartmouth College, and has since then been revised several times. An advantage of BASIC is that its rules of form and grammar are quite simple and easy to learn. It is the purpose of this manual to present the language BASIC and to show how it is used to solve simple problems and deal with many situations common in computing. More complicated problems can be solved by combining the simpler steps shown here.

2 A BASIC PRIMER

2.1 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:

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

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

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. 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 enclosed in quotation marks which are to be printed out, as in line number 65 on the previous page. Thus, spaces may be used, or not used, at will to "pretty up" a program and make it more readable. Statement 10 could have been typed as 10READ A,␣B,␣D,␣E and statement 15 as 15LET G=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 obviously used to denote multiplication.) In this statement we direct the computer to compute the value of AE - BD, and to call the results 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 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

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

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

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

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

As before, it finds the solution in 37 and 42 and prints them out in line 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

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

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 statement. The computer then informs you that it is out of data, printing on the paper in your teletypewriter "RE 1 IN LINE 30".

RE 1 means Run time Error no. 1. Run time errors (RE) are errors detected during execution of a program whereas errors detected during compilation of a program are called compile time errors and abbreviated CE followed by a number. A complete error list is given in Chapter 7.

For a moment, let us look at the importance of the various statements. For example, what would have happened if we had omitted line 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) its answers, it would not do it, and the solutions would be the computer's secret. What would have happened if we had left out line 20? In this problem just solved nothing would have happened. But, if G were equal to zero, we would have set the computer the impossible task of dividing by zero, and it would tell us so, printing "RE 5 IN LINE 37". If we had left out statement 60, the computer would have solved the first system, printed out the values of X and Y, and then gone to line 65 where it would be directed to print "NO UNIQUE SOLUTION". It would do this and then stop.

One very natural question arises from the seemingly arbitrary numbering of the statements: Why this selection of line numbers? The answer is that the particular choice of line numbers is arbitrary, as long as the statements are numbered in the order we want the machine to follow in executing the program. We could have numbered the statements 1, 2, 3, 4, ... 13, although we do not recommend this numbering. We would normally number the statements 10, 20, 30, ... 130. We put the numbers a certain distance apart so that we can later insert additional statements if we find that we forgot them when we originally wrote the program. 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 data elements in the DATA statements: Why were they placed as they were in the sample program? Here again the choice is arbitrary and we need only to 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 next C, etc.). In place of the three statements numbered 70, 80 and 85, we could have put

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

or we could have written, perhaps more naturally,

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

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

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

```
10 READ A, B, D, E
15 LET G=A*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
 6.66667E-01       1.66667E-01
-3.66667           3.83333
RE 1 IN LINE 30
```

After typing the program, we type RUN followed by a CARRIAGE RETURN. Up to this point the computer stores the program and checks the form of the statements. This process is called compiling. It is the RUN command which directs the computer to execute your program. The message out-of-data error code here may be ignored. However, in some cases it indicates an error in the program: for more details, see Section 2.7.2.

2.2 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, and these are listed in the following table:

Symbol	Example	Meaning
+	A + B	Addition (add B to A)
-	A - B	Subtraction (subtract B from A)
*	A * B	Multiplication (multiply B by A)
/	A / B	Division (divide A by B)
↑	X ↑ 2	Raise to the power (find X^2)

We must be careful with parentheses to make sure that we group together 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 $A + BC^D$. If this is not the order intended, then we must use the 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 the 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 operations of the same priority, the operations are performed from left to right.

The 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 the following list shows:

Functions	Interpretation	
SIN (X)	Find the sine of X	X interpreted as a number, or as an angle measured in radians.
COS (X)	Find the cosine of X	
TAN (X)	Find the tangent of X	
ATN (X)	Find the arctangent of X	
EXP (X)	Find e^X	
LOG (X)	Find the natural logarithm of X (ln X)	
ABS (X)	Find the absolute value of X ($\|X\|$)	
SQR (X)	Find the square root of X ($\sqrt{X}$)	
LG10 (X)	Find the common logarithm of X	

Three other functions are also available in BASIC: INT, RND, and SGN: these are reserved for explanation in Chapter 4. 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).

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

```
10 PRINT (5/6) ↑ 17
20 END
```

and the computer will find the decimal form of this number and print it out in less time than it took to type the program.

2.2.1 Numbers

A number may be positive or negative and it may contain up to approx. nine significant digits, but it must be expressed in decimal form. For example, all of the following are numbers in BASIC: 2, -3, 675, 1234567. -7654321 and 483.4156. The following are not numbers in BASIC: 14/3 and $\sqrt{7}$. We may ask the computer to find the decimal expression 14/3 and $\sqrt{7}$, and to do something with the resulting number, but we may not include either in a list of DATA. We gain further flexibility by use of the letter E, which stands for "times ten to the power". Thus. we may write .00123456789 in a form acceptable to the computer in any of several forms: .123456789E-2 or 123456789E-11 or 1234.56789E-6. We may write ten million as 1E7 (or 1E+7) and 1965 as 1.965E3 (or 1.965E+3). We do not write E7 as a number, but must write 1E7 to indicate that it is 1 that is multiplied by 10^7.

2.2.2 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. However, all variables are set to zero before a RUN. Thus, it is not necessary to assign a value to a variable before using the variable in a computation.

2.2.3 Relational Operators

Six other mathematical symbols are provided for in BASIC, symbols o. relation, and these are used in IF - THEN statements where it is necessary to compare values. An example of the use of these symbols was given in the sample program in Section 2.1.

Any of the following six 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)
< = or = <	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)
> = or = >	A > = B	Is greater than or equal to (A is greater than or equal to B)
< > or > <	A < > B	Is not equal to (A is not equal to B)

2.2.4 PI

The symbol PI is a constant (3.14159265) which may be used in any arithmetic expression.

2.3 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)
     .............
990  PRINT 99, SQR (99)
1000 PRINT 100, SQR (100)
1010 END
```

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

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

Statement 10 gives the value of 1 to X and "initializes" the loop. In line 20 both 1 and its square root are printed. Then, in line 30, X is increased by 1, to 2. Line 40 asks whether X is less than or equal to 100; an affirmative answer directs the computer back to line 20. Here it prints 2 and $\sqrt{2}$, and goes to 30. Again X is increased by 1, this time to 3, and at 40 it goes back to 20. This process is repeated, line 20 (print 3 and $\sqrt{3}$), line 30 (X = 4), line 40 (since $4 \leq 100$ go back to line 20), etc. - until the loop has been traversed 100 times. Then after it has printed 100 and its square root, 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 simple. They are 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. 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 to 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 clause, a step size of +1 is assumed.

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

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

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

If the initial value is greater than the final value (less than for negative step size), then the body of the loop will not be performed at all, but the computer will immediately pass to the statement following the NEXT. For example, the following program for adding up the first n integers will give the correct result 0 when n is 0.

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

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

Allowed

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

Not allowed

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

Allowed

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

2.4 Arrays

In addition to the ordinary variables used by BASIC, there are variables which can be used to designate the elements of an array. These are used where we might ordinarily use a subscript, for example the coefficients of a polynomial $[a_0, a_1, a_2, \ldots\ldots]$ 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 array, 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 array A(0), A(1), A(2), ... 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, 5, 2.2, 4, -9, 123, 4, -4, 17
```

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 DIM statement to indicate to the BASIC system that it has to save extra space for the array. 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 = TO 15, but the form as typed would allow for the lengthening of the array by changing only statement 60, so long as it did not exceed 25.

We would enter a 3 x 5 array 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 an array with no dimension statement, and it will handle all the entries from B(0,0) to B(10,10). If you try to enter an array with a subscript greater than 10, without a DIM statement, you will get an error message telling you that you have a subscript error. This is easily rectified by entering the line:

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

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

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

Shown below is a list and run of a problem which uses both a singly and a doubly subscripted array. The program computes the total sales of each of five salesmen, all of whom were selling the same three products. The array P gives the price/item of the three products and the array S tells how many items of each product each man sold. You can see from the program that product no. 1 sells for $ 1.25 per item, no. 2 for $ 4.30 per item, and no. 3 for $ 2.50 per item; and also that salesman no. 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 sales array in lines 40 - 80, using data in lines 910 - 930. The same program could be used again, modifying only line 900 if the price changes, and only lines 910 - 930 to enter the sales in another month.

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

Since the DIM statement is used to save space for arrays, the DIM statement must be executed before the space is being used. Normally the DIM statements will be placed near the beginning of the program.

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

RUN
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

DONE
```

2.5 Use of the 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 have the computer solve our problem?

First, ascertain that the BASIC system is present. If no, the system is loaded as explained in Section 7.3. When the computer types READY you should begin to type your program. Make sure that each line begins with a line number which contains no non-digit characters. Be sure to press the CARRIAGE RETURN key at the completion of each line. Spaces may be inserted at any point in the line, including before the line numbers.

If, in the process of typing a statement, you make a typing error and notice it immediately, you can correct it by pressing the backwared arrow "←". This will delete the preceding character, and you can then type in the correct character. Pressing this key a number of times, say n,

will erase from this line the n last characters. To delete all of the present line, press CTRL Q. (Press the key marked CTRL and type Q.) Programs or data may be annotated by typing the remark and then deleting the line (as far as the system is concerned) with CTRL Q. BASIC types " \" to show that a line has been deleted.

When a line is finished, you press the return key. Then the statement is analyzed by the computer and if any syntax error is found, an error message is printed. The computer will now check the next input character and if you print a question mark, the whole erroneous line will be printed with minor errors underlined and with an arrow pointing to where the computer stopped compiling.

After typing your complete program, you type RUN, press the CARRIAGE RETURN key, and hope. If the program is one which the computer can run, it will then run it and print 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 "grammatical errors". If it had errors of this type, the computer would have printed an error code as soon as the error was detected during the typing of the program. Errors detected after RUN are structural (loop nesting, matching GOSUB and RETURN) or arithmetical errors. A list of the error codes is given in Chapter 7 together with the interpretation of each.

If you are given an error message, you can correct the error by typing a new line 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 CARRIAGE RETURN. 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.

If it is obvious to you that you are getting the wrong answers to your problem, even while the computer is running, you can type ESC and the computing will cease. It will type BREAK and you can start to make your corrections. If you are in serious trouble, use the break character (ESC) and type SCRATCH. When the system is ready to accept a new program, READY will be typed.

A sample use of the system is shown below.

```
10  FOR N=1 TO 7
20  PRINT N, SQR(N)
30  NEXT N
50  END

RUN
 1            1
 2            1.41421
 3            1.73205
 4            2
 5            2.23607
 6            2.44949
 7            2.64575

DONE
```

2.6 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 likely that errors will be present and will have to be corrected. Errors are of two types: errors of form (or syntax errors) which prevent the running of the program, and logical errors in the program which cause the computer to produce wrong answers or no answers at all.

Errors of form will cause error codes to be printed, and the various error codes are listed and explained in Chapter 7. Logical errors are often much harder to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines, or by deleting lines from the program. As indicated in the last section, a line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those of two existing lines; and a line is deleted by typing its line number and pressing the CARRIAGE RETURN key. Notice that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers will start out using line numbers that are multiples of five or ten, but that is a matter of choice.

These corrections can be made at any time - whenever you notice them - either before or after a run. Since the computer sorts lines out and arranges them in order, a line may be retyped out of sequence. Simply retype the 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 .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, 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 largest of these two numbers M. Then it will pick the larger of M an 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 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 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; 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.

```
NEW
NEW PROGRAM NAME -- MAXSIN
READY
 10  READ D
 20  LWR X0 = 0

CE11
 20  LET X0 = 0
 30  FOR X = 0 TO 3 STEP D
 40  IF SINE ← (X)  ← ←<= M THEN 100
 50  LET X0 = X
 60  LET M = SIN (X)
 70  PRINT X0, X, D
 80  NEXT Z ← X0
 90  GO TO 20
100  DATA .1, .01, .001
110  END
RUN

RE 13 IN LINE 80
```

A message indicates that LET was mistyped in line 20, so we retype it, this time correctly.

Notice the use of the back arrows to erase a character in line 40, which should have started IF SIN (X) etc., and in line 80.

The error message RE13 indicates a FOR statement without a NEXT. Upon checking we see that the variable in the FOR and NEXT are different, so we correct statement 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.

```
80  NEXT X
40  IF SIN (X) < = M  THEN 80
RUN
 0.1            0.1          0.1
 0.2            0.2          0.1
 0.3            0.3          0.1
BREAK
```

M has never been assigned an initial value and is assumed to be zero. We decide to give it a value less than the maximum value of the sine, say -1.

```
20  LET M = -1
RUN
 0              0            0.1
 0.1            0.1          0.1
 0.2            0.2          0.1
BREAK
```

This is incorrect. We are having every value of X0,X, and the interval size printed, so we direct the machine to cease operations by typing ESC even while it is running. Notice that the ESC does not print, but the word BREAK is printed.

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

```
70
85  PRINT X0, M, D
RUN
 1.6            9.99574E-01      0.1
 1.6            9.99574E-01      0.1
BREAK
```

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.

```
90  GO TO 10
5  PRINT "X VALUE", "SIN", RESOLUTION"
CE 13 ?
5  PRINT "X VALUE", "SIN", RESOLUTION"
                           ----------↑
```

There is an error on our PRINT statement. As we do not see it immediately, we type a question mark. Following the arrow the debugging is easy; no left quotation mark for the third item.

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

```
5  PRINT "X VALUE", "SIN", "RESOLUTION"
RUN
X VALUE        SIN             RESOLUTION
 1.6            9.99574E-01     0.1
 1.57           1               0.01
 1.571          1               0.001

RE 1 IN LINE 10
```

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

Having changed so many parts of the program, we ask for a list of the corrected program. Listing the corrected program, from time to time, is an important part of debugging. Using LISTH will list the program name as a header:

```
LISTH

MAXSIN
5  PRINT "X VALUE", "SIN", "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 X0=X
60  LET M=SIN(X)
80  NEXT X
85  PRINT X0,M,D
90  GO TO 10
100  DATA .1, .01, .001
110  END

SAVE F-P
```

The program is saved for later use by punching it on the Fast Punch. This tape may be read from the Tape Reader on some later occasion.

In solving this problem, there is a common device which we did not use, namely 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.

With harder problems we can use the STOP statement, 58 STOP.

With STOP, execution is halted and control is returned to Teletype. Then we can change the program, or check the value of the variables using the PRINT command. (PRINT without statement number.)

2.7 Summary of elementary BASIC Statements

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

2.7.1 LET

This statement is not a statement of algebraic equality, but 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] . More generally, several variables may be assigned the same value by a single LET statement.

Examples: (of the first type):

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

(of the second type):

```
50  LET X=Y3=A(3,1)=1
90  LET W=Z=3*X -4*X↑2
```

2.7.2 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 create 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, it is assumed that the program is done and we get an out-of-data error code.

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 is of the form:

DATA [sequence of numbers]

Examples:

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

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

 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.

2.7.3 PRINT

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

a) To print out the result of some computations.

b) To print out verbatim a message included in the program.

c) A combination of the two.

d) To skip a line.

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

Examples of type a):

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

The first will print X and then, a few spaces to the right of that number, its square root. The second will print five different numbers:

X, Y, Z, B^2 -4AC, and e^{A-B}.

The 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 the three numbers called for in a PRINT statement (as long as the labels do not exceed 14 characters) 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.

Examples of type d):

```
250  PRINT
```

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

2.7.4 GO TO

There are times in a program when you do not want all commands executed 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 to the END statement yet, but to go through the same process for a different value of D. Therefore, we directed the computer to go back to line 10 with a GO TO statement. Each statement has the form GO TO [line number]. (It is possible to go to a non-executable statement; control then passes to the sequential executable statement.

Example:

```
150  GO TO 75
```

2.7.5 IF - THEN or IF - GO TO

There are times when 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. The more common form of the statement is:

```
IF [formula] [relation] [formula] [THEN] [line number]
IF [formula] [relation] [formula] [GO TO] [line number]
```

Examples:

```
40  IF SIN (X)  < =  M THEN 80 or
40  IF SIN (X)  < =  M GO TO 80

20  IF G  =  0 THEN 65 or
20  IF G  =  0 GO TO 65
```

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.

2.7.6 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]

Any simple (not subscripted) variable may be used as the FOR variable. 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 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 used 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 proceed to the statement immediately following the corresponding NEXT statement.

2.7.7 DIM

Whenever we want to enter an array with a subscript greater than 10, we must use a DIM statement to inform the computer to save us sufficient room.

Examples:

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

The first would enable us to enter an array of 35 items (36 if we use H(0)), and the latter a 5 x 25 array (6 x 26 if we use row 0 and column 0).

2.7.8 STOP

A STOP statement may be entered anywhere in a program. With STOP execution is terminated and control is passed to the Teletype.

2.7.9 END

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

Example:

```
999 END
```

2.7.10 The ON. . . GO TO Statement

Using an IF... THEN statement provides only a two-way branch in a program. A decision between only two alternatives can be made. More branches can be achieved by using multiple IF...THEN statements. However, a single statement, ON..GO TO, allows a manyway branch. For example, the following lines in a longer program

```
 90 READ X
100 IF X = 1 THEN 500
110 IF X = 2 THEN 600
120 IF X = 3 THEN 700
130 DATA 3
```

could be replaced by these three lines:

```
 90 READ X
100 ON X GO TO 500, 600, 700
130 DATA 3
```

The format of the ON...GO TO instruction is as follows:

```
100 ON X GO TO xxxx, yyyy, zzzz,...
```

where X is any number or formula, and xxxx, yyyy, zzzz,... are line numbers. If X is equal to 1, the computer takes its next instruction from line xxxx, if X is 2, control passes to yyyy, and so on. If the value of X is not an integer, its integer part is used. If the value of X is less than one or greater than the number of line numbers listed, control is transferred to the next line of the program. There may be any number of line numbers listed in the instruction, as long as the entire instruction fits on a single line.

3 INTERACTIVE USE OF THE BASIC SYSTEM

The BASIC system was built for interactive use, and when you are sitting at the Teletype working with your program some BASIC features may be of help in coding and debugging.

If you discover a mistyped character before typing carriage return, you can delete the most recently typed character by using ← (left arrow, shift O). Applying this character many times may delete the whole line.

```
30 LET A = SIN← ← ← COS(X)
```

Here SIN is deleted using ← three times.

A quicker way of deleting the whole line is typing ctrl-Q (press the pushbutton CTRL and type Q). The system responds printing reverse slant (\) and issuing a new line.

Example:

```
100 FRO X=1 TO 5 \
```

When you terminate a line typing carriage return, this line is handed to the compiler for syntax check. With errors proper messages are given and if the next character input is a question mark, the line is printed with the errors marked.

```
200 FOR X=2A TO B ,C
CE 1 ?
200 FOR X=2A TO B ,C
          -       ↑
```

If you want to change your program, you can use the editing facilities to list, delete, insert or move lines. You may test one part of the program at a time as you can start execution at any specified line and insert an END statement where you want to stop.

Inserting STOP statements you may halt execution to check and change the program, then execution may be continued.

At any time your program may be saved for later retrieval. For these activities commands are used. A command is not given a line number and the system takes action when you type carriage return.

3.1 Initializing the User Terminal

The BASIC system is initialized when it is entered, but if you have been programming for some time and want to change to a new program, all tables should be reset. This is achieved using the commands NEW, OLD and SCRATCH.

3.1.1 Entering the BASIC System

When the BASIC system is called using the command

BASIC

in TSS or printing the character ESC in BASIC Time Sharing System, the start-up procedure is as follows:

```
BASIC ON LINE
NEW OR OLD - - NEW (alternatively OLD)
NEW PROGRAM NAME - - TEST
READY
```

The terminal is now ready to accept BASIC commands and statements.

3.1.2 NEW, OLD and SCRATCH

If you type OLD, the system expects you to continue working with a problem which is saved earlier. The system then resets tables and asks where the program is to be found, printing:

OLD FILE NAME - -

If the program was saved on a mass memory file (disk file), you should give the name of this file. If you saved the program on paper tape, you should put the tape in the paper tape reader and print the file name for the tape reader, T-R. If the program is punched on cards, you may use the card reader, C-R, as the input file name.

With carriage return the system starts reading and when finished it will respond by: READY. (Possibly error messages may be printed, see GET.)

The program is now in core and you may start working on it.

With NEW the system expects you to type a new program and asks for the name of this program. Then, the tables are reset and the system prints READY.

If you want to work out a new program using the current program name, just type SCRATCH. This command responds READY when the old program is removed.

3.1.3 Naming of Programs

Program names are used as a header with listings and runs if the commands RUNH or LISTH are used (H = header).

The program name should start with a letter and have no more than 12 characters. Quotes, spaces and other nonprintable characters should not be used.

If you use OLD, the file name is used as program name.

You may set a program name by typing the command NAME followed by the new program name.

```
NAME SQUARES
```

will set SQUARES as the current program name.

If no program name is given when the system asks for it (carriage retu is typed), the name TEST is used.

3.2 Saving and Retrieving BASIC Programs

When you are working on a program and want to continue later, you should save the program by using the SAVE command with the appropriate file name. A hard copy is produced using Teletype or line printer, a tape may be punched using the Teletype punch or the fast punch. With mass memory available, the program may be saved on mass memory files.

A saved program is entered later using GET followed by the appropriate file name.

If you have other programs on tape, on cards or on a mass memory file, you can naturally use the same command to read the program.

Notice that devices such as card readers, line printers and so on must be reserved before use. (See Section 3.4.5.)

3.2.1 The SAVE Command

The SAVE command will save a BASIC program. The appropriate names for the SAVE command are as follows:

TTY designates the user Teletype.
F-P designates the fast punch.
L-P designates the line printer.

With other file names the system expects that you want the program saved on a mass memory file.

SAVE SQUARE

will save the current program on the mass memory file named SQUARE. If you have no file with such a name, the file must be created. To do this you should enclose the file name in quotes.

SAVE "SQUARE"

will create the file SQUARE and save the current program onto this file.

Program names may be used as file names, with the exception of the names of I/O-equipment. Further information on file naming may be found in the documentation for the file system.

To save the program on paper tape using the tape punch on the Teletype, you may do as follows:

- Type "SAVE TTY", but type no carriage return.
- Turn the Teletype to local.
- Turn on the Teletype tape punch.
- Produce a leader by pressing the pushbutton HERE IS several times.
- Turn the Teletype to line.
- Type carriage return.
- When the program is listed, a trailer is produced by turning the Teletype to local and pressing HERE IS several times.
- Turn off the Teletype punch.
- Turn Teletype to line.

3.2.2 The GET Command

The GET command will read a BASIC program.

The appropriate names for the GET command are as follows:

TTY designates the user Teletype.
T-R designates the fast paper tape reader.
C-R designates the card reader.

With other names the mass memory directory is searched, first the user catalogue and then the system catalogue. You can find more information about the mass memory file system in the manual for the file system.

When the correct file is found, the system starts reading one line at a time. Every line is checked and compiled. Error messages are buffered. The input stream should be terminated by the correct character ETB (ASCII - ETB, typed ctrl-W on the Teletype). If not, an error message is issued with end-of-file.

When the program is read, error messages are printed and you can type the corrected version of these lines. If a long list of errors messages is bothering you, just press the break character and the rest is discarded.

3.3 Executing Your Program

When you think your program, or part of it, is finished, you can try to run it using the command RUN.

Before execution starts, the system will reset variable values and check the program. If no errors are found, program execution is started. Execution will continue until either BREAK or an END statement is found or until an error condition occurs. Then execution is terminated, variables reset and control passed to the Teletype.

If a STOP statement is encountered, program execution is halted and control passed to the Teletype. You may then examine and change your program. Execution is restarted using the CON or RUN commands.

Possibly your program will produce erroneous results or it may be executing some endless loops. You can then force execution to be terminated by using the break character.

3.3.1 The RUN Command

This command is used to initiate execution. You may type RUNH which means the computer should start by printing the program name as identification. Normally, the command is used without a parameter, which means that execution should start with the first line. But if you print a line number as parameter, execution is started at that line.

```
RUNH 500
```

will start execution at line 500.

Before program execution is started, the system will reset variables and check the program for mismatching FOR - NEXT, errors with multi-line DEF FN functions, the initialization of numeric and string data and so on. Numeric variables are set to zero, string data are set undefined.

3.3.2 Terminating Execution

Program execution is continued until an END statement is reached, an error is found or you break execution by typing the break character (ESC).

The break character in BASIC is normally ESC. In special operating systems, however, the system's break character must be used. For this, you should consult the manual for the appropriate operating system.

If you use the NORD-TSS, the break character is ESC. When you press the ESC pushbutton, the NORD-TSS command processor is entered and you may restart BASIC printing: CON

When program execution is terminated, there is a response from the system:

- With errors an error message is printed.
- With the break character "BREAK" is printed.
- With END the systems prints "DONE".

When execution is terminated, variables are reset, files are closed and control is passed to the Teletype.

3.4 Editing Programs

If you want to change your program, BASIC has editing facilities to list, delete and renumber a part of or the whole program.

Arguments to specify which lines should be listed, deleted or renumbered are as follows:

	No arguments mean all lines.
s1,s2,...,sn	Lines s1, s2,..., sn
s1-s2	Lines s1 to s2 inclusive
s1-	Line s1 and lines following.

3.4.1 The DELETE Command

The DELETE command is used to remove the lines specified from the program.

```
DELETE 100-
```

will remove line 100 and all lines following.

If you want to delete only one line, it may be more convenient to print the line number followed by carriage return.

```
50
```

is equivalent to printing

```
DELETE 50
```

3.4.2 The LIST Command

The LIST command is used to obtain a listing of the lines specified.

```
LIST 50, 60, 120
```

will list line 50, line 60 and line 120.

If you want the program name to be printed as a header for identification, you can use LISTH.

```
LISTH
```

will print the program name and the whole program.

3.4.3 Changing a Line

If you type a new line with the same line number as an old one, the new line will replace the old one.

3.4.4 The RENUMBER Command

The RENUMBER command is used to change the statement line numbers and the references to these line numbers. Line numbers in comments are not changed. As statements in BASIC are ordered according to the line numbers, you can use the RENUMBER command to move individual lines or parts of the program.

You can specify what lines should be renumbered as mentioned above.

```
RENUMBER 100 - 400
```

will set new line numbers and change references for the lines from 100 to 400.

The new line numbers are specified as follows:

FIRST followed by a positive integer indicates what the line number of the first renumbered line should be. Successive line numbers for renumbered lines are obtained by incrementing the FIRST number. You may set the increment printing STEP followed by a positive integer.

```
RENUMBER 100 - 400 FIRST 10 STEP 5
```

will set the line numbers of lines 100 to 400 to 10, 15, 20, 25 and so on.

If you give no first line number, the first renumbered line will be set to 100 and if no increment is given, an increment of 10 will be used.

Notice that if you RENUMBER lines, you may generate line numbers which already exist. This will remove the old lines with these line numbers.

3.4.5 Reserving Peripherals

In a time sharing system where many users share the same peripherals, the common devices must be reserved before use to prevent more than one user at a time.

If your BASIC system is running under a monitor system like NORD TSS, reservations are done entering the monitor system. The appropriate commands will be found in the documentation for the operating system.

In a BASIC time sharing system without mass memory, reservations are done using the command RESERVE followed by the name of the peripheral. These names are:

L-P - line printer
F-P - fast punch
C-R - card reader
T-R - tape reader

```
RESERVE L-P
```

will reserve the line printer. If you try to reserve a device which is already reserved, the system will give a message to indicate for whom the device is reserved.

When you are through using the device, you should release it to enable others to use it. This is done by printing RELEASE followed by the name of the peripheral device.

```
RELEASE C-R
```

will release the card reader.

3.4.6 The TABLE Command

The error message RE31 is issued if the system's table area is over flowed. The size of this area may be changed using the command TABLE.

The command is used with integers from 3 to 9 as parameters. Using this command an area is allocated by the system for book-keeping purposes. The area is proportional to the integer (size is n*256).

The area is used for a lot of purposes, such as: Input buffer, buffers for compiling and for run-time purposes, garbage collection, variable tables.

When this area is increased, core for run-time purposes is decreased by the same amount.

3.4.7 The DIGIT Command

The number of significant digits used in the printout may be changed using the DIGIT command. The command takes as parameter integers from 3 to 9.

3.4.8 The CHAIN Statement

A programming task may be solved using many BASIC programs. One program may then start a new program using the statement CHAIN.

If the first program is started using RUNH, the program name is printed as identification for each CHAIN-ing.

Data is passed from one program to another using sequential or random files.

Example:

```
10 PRINT    "THIS IS A GAME SELECTOR"
15 PRINT    "THE GAMES ARE: LUNAR,";
20 PRINT    "FOOTBALL, TIC-TAC-TO"
30 PRINT    "PRINT YOUR CHOICE, PLEASE"
40 INPUT A$
60 CHAIN A$
70 END
```

FORMAT for the CHAIN statement:

line no. CHAIN <FILE NAME>

3.5 Terminating

When you have finished programming and saved what programs you would like to use later, you should print BYE. This command will enter the monitoring system. In BASIC systems without mass memory other users may take advantage of the core released when you leave the system.

4 MORE ABOUT BASIC

4.1 Functions

There are three functions which were listed in Section 2.2, but not described. These are INT, SGN, and RND.

4.1.1 Integral Function

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 off numbers. We may use it to round to the nearest integer by asking for INT(X + .5). This will round off 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 off 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 any specific number of decimal places. For example, INT(10*X + .5)/10 will round off X correctly to one decimal place, INT(100*X + .5)/100 will round off X correctly to two decimal places, and INT(X*10↑D + .5)/10↑D will round off X correctly to D decimal places.

4.1.2 SGN

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

Function	Argument Value	Yield
SGN	Zero	0
SGN	Positive, not zero	+1
SGN	Negative, not zero	-1

Examples:

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

4.1.3 Pseudo-Random Number Generator

In BASIC programs which attempt to simulate complicated systems or apply Monte Carlo integration techniques, there is need for a facility which provides random numbers. Since a digital computer is a deterministic device, it can not produce a truly random number. However, there are simple techniques for producing numbers which have many of the properties of random numbers, and do not have any easily recognized pattern. The BASIC function RND is a pseudo-random number generator. RND is a function which requires no arguments and produces a number greater than or equal to 0 and less than 1.

As an example of the use of RND for simulation, consider the following:

```
100 IF RND > 1/3 THEN 940
200 PRINT "GAME DELAYED.  DOG ON FIELD."
300 PRINT
400 GO TO 940
```

A pseudo-random number greater than or equal to A and less than B can be obtained simply by writing

```
100 LET X = RND * (B-A) + A
```

Normally the same sequence of pseudo-random numbers is generated by successive calls to RND each time the program is executed. This is an aid for debugging programs, but simulations should be run with different sets of random numbers every time. After a program is debugged, inserting the statement

```
1 RANDOM
```

in the beginning of the program will reset the pseudo-random number generator, so that different sequences of pseudo-random numbers will be obtained.

4.2 Arithmetic Expressions in BASIC

BASIC will admit general arithmetic expressions in most connections where numbers are allowed. Exceptions are: Line numbers must be positive integers. Numbers are used in data statements and with input.

$0,4\,(B-A) + A = 0,4\,B + 0,6\,A$

4.2.1 Arithmetic Symbols in BASIC

In the examples in this chapter, arithmetic formulas are used, and examples of the way they are evaluated by the computer are given. Five symbols representing arithmetic operations can be used in formulas. These symbols are listed in the table below; the first four are used in the programs in this chapter.

Symbol	Formula	Meaning
+	A + B	Addition: add B to A
-	A - B	Subtraction: subtract B from A
*	A * B	Multiplication: multiply A by B
/	A / B	Division: divide A by B
↑	A ↑ B	Exponentiation: raise A to power B (On many terminals the symbol for exponentiation is ^.)
-	-A	Unary minus: a minus which starts an expression or which follows immediately after = or (

The way a formula is written determines how the computer will evaluate it.

1) 10 ↑ 2 + 1

The computer evaluates this formula as 100 + 1 = 101. It will perform the exponentiation before the addition.

2) 10 ↑ 2 / 2 * 3

The value given for this formula is 100 / 2 * 3 = 50 * 3 = 150. The computer performs the exponentiation first. When multiplication and division appear together, the left-most operation is performed first. Thus in this example, the division is performed second and finally the multiplication.

3) 5 + 2 * 3 - 1

The value of this formula is 5 + 6 - 1 = 11 - 1 = 10. The computer performs the multiplication first. As with multiplication and division, the positions of the + and - symbols determine which operation is performed first. Addition and subtraction are performed from left to right. So, in this example, the addition is performed second and the subtraction last.

4) 32 / 4 ↑ 2 + 3 * 3 - 1

This formula uses all the available symbols for arithmetic operations, and the steps by which the computer evaluates it are as follows. First exponentiation is performed and the formula is reduced to 32 / 16 + 3 * 3 - 1. Then division and multiplication are performed from left to right and the

simplified formula is 2 + 9 - 1. Finally addition and subtraction are performed from left to right and the value of the formula is seen to be 10.

The placement of parentheses in a formula can alter the sequence in which the operations are performed. Two of the preceding examples have been rewritten to illustrate this.

1) 10 ↑ (2 + 1)

The computer evaluates this formula as 10 ↑ 3 = 1000. The formula inside the parentheses is evaluated first, and then the exponentiation is performed.

2) ((32 / 4) ↑ 2 + 3) * (3 - 1)

This formula will be evaluated as follows:
(8 ↑ 2 + 3) * (3 - 1) = 67 * (3 - 1) = 67 * 2 = 134. The formula inside the "innermost"parentheses is evaluated first. Within parentheses the described sequence of performing the operations applies.

Since two BASIC arithmetic operators may not be adjacent, parentheses are needed in some formulas containing negative numbers. For example, "X raised to the -2 power" would be written X ↑ (-2), and "-3 subtracted from 2" would be written 2 - (-3).

In summary, to insure the proper interpretation of formulas you should remember that the computer performs exponentiation first, multiplication and division second, and addition and subtraction last unless otherwise indicated by placement of parentheses. When in doubt about how a formula will be evaluated, use parentheses.

4.2.2 Exponentiation in BASIC

The symbol for exponentiation in BASIC is the up-arrow (↑). Exponentiation is performed as follows.

Consider the expression A↑B. The manner in which the value of this expression is calculated depends on the values of A and B.

1) If A is > 0, A↑B = EXP (B* LOG (A))

2) If A = 0

   a) and if B is > 0, A ↑ B = 0

   b) and if B is = 0, A ↑ B = 1

   c) and if B is < 0, A ↑ B = 0
   An error message is printed when this exponentiation is attempted.

3) If A < 0

a) and B integer > 0, A ↑ B = 1 multiplied by A a total of B times.

b) and B is = 0, A ↑ B = 1

c) and B integer < 0, A ↑ B = 1/(A ↑ ABS(B))

d) and B is not an integer. B is rounded to integer. Then A ↑ B is computed according to a, b and c above.

4.2.3 More about LET

In the LET statement, values can be assigned to variables, as with the READ and INPUT statements (e.g., 100 LET X = 2). However, the LET statement is also a command to the computer to perform certain computations and to assign the answer to a certain variable (e.g., 110 LET X = X + 1).

More generally, several variables may be assigned the same value by a single LET statement. Two examples follow:

```
100  LET X = Y3 = 1 E 2
110  LET A(X) = X = X + 1
```

In line 110, the new value of X is used for the subscript of A. That is, after execution of line 110, A(101) and X are equal to 101, and A(100) remains unchanged. Note also that numeric constants may be represented in scientific notation (Section 2.2.1), as well as in integer or fractional notation, anywhere in a program.

4.3 Other useful Statements

4.3.1 RANDOM

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

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

```
1 RANDOM
```

before starting execution runs.

4.3.2 INPUT

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

```
40 INPUT X,Y
```

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

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

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

and the system will type

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

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

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

4.3.3 REM

REM provides a means for inserting explanatory remarks in a program. The system completely ignores the remainder of that line, allowing the programmer to follow the REM with directions for using the program, with identifications of the parts of a long program, or with anything else that he wants. Although what follows REM is ignored, its line number may be used in a GOSUB, IF - THEN, GO TO, or ON-GO TO 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
130 '

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

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

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

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

In line 130 the line number is followed by an apostrophe and the rest of the line is left blank. Such blank lines are used to increase the readability of the program listing.

4.3.4 RESET

Sometimes it is necessary to use the data in a program more than once The RESET statement permits reading the data as many additional times as it is used. Whenever RESET is encountered in a program, the system resets the data block pointer to the first number. A subsequent statement will then start reading the data all over again. A word of warning - - if the desired data is preceded by code numbers or parameters, superfluous READ statements should be used to pass over these numbers. For example, the following program portion reads the data, restores the data block to its original state, and reads the data again. Note the use of line 570 to "pass over" the value of N, which is already known.

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

4.4 Representations of Strings

The BASIC programs described thus far have all dealt with numbers. In the statement

```
100 LET A = B + 3.1415926
```

the sequence 3.1415926 is a representation of a number; the character B is the name of a number which can vary as the program is executed by the computer. The character A is the name of a number which may be changed by the execution of that statement. Although computers are excellent machines for performing high-speed arithmetic, some of their most important uses are in the manipulation of entities which do not represent numbers. A string is such an entity.

A string is a sequence of characters; these include letters, digits, blanks, and other special characters such as those which appear on the terminal. One way of representing a string in BASIC is to enclose it in quotation marks. Such string constants have already been introduced in PRINT statements where they have been called labels. For example, the string in

```
100 PRINT "TYPE THE INITIAL BALANCE"
```

is a string constant just as the number 3.1415926 in the preceding example is a numeric constant.

Just as BASIC has names for numbers, it also has names for strings. A name of a simple string is formed exactly as a name for a number, except that it includes a trailing dollar sign ($). That is, a string name is a single letter, followed by an optional single digit, followed by a dollar sign. Thus A$, Z$, Q3$, W7$ are legitimate string names, but 4$, BB$ are not. The string A$ is entirely distinct from the number A, and both names can appear in the same BASIC program.

4.4.1 Assigning Values to Strings and String Comparisons

A string variable can take on a string value through a READ statement. The following BASIC program reads three strings and prints them.

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

Note that the items in the DATA statement are representations of strings, not numbers. This program prints the word TIMESHARING on the terminal. Since the quotation marks are used to delimit the strings, it is not possible to create a string containing a quotation mark in this manner.

Strings can also be assigned values through the use of LET statements. For example

```
10 LET A$ = "H2SO4"
20 LET B$ = A$
30 PRINT B$
40 END
```

will print the string H2SO4 on the terminal.

Another way that a string can take on a value is by having the program request the input of a string from the terminal through an INPUT statement. For example:

```
10 PRINT "A MIXTURE OF FUEL AND OXIDIZER WHICH"
20 PRINT "BURNS SPONTANEOUSLY IS TERMED";
30 INPUT A$
40 IF A$ = "HYPERBOLIC" THEN 70
50 PRINT "WRONG"
60 GO TO 80
70 PRINT "RIGHT"
80 END
```

After printing the textual message the program will print a question mai Suppose the user enters the word "HYPERVENTILATED" in response. Statement 40 is a stringconditional statement. If the string A$ is the same as the string "HYPERBOLIC", then statement 70 will be executed next. Since the user did not enter "HYPERBOLIC" he has WRONG printed on his terminal.

Any of the relational operators described in Section 1.3.3 may be used in an IF...THEN statement to compare strings. The relational operator "<" is interpreted as meaning "earlier in alphabetical order than " and the relational operators are defined appropriately. The ordering of characters is arbitrarily defined by the ASCII code which is explained in Section 8.3. In any string comparison. trailing blanks in a string are ignored; thus "YES" = "YES "

4.4.2 Relaxation of Requirement for Quotation Marks

Strings which are entered in response to an INPUT statement need not be bracketed by quotation marks as long as the items being entered do not contain commas or do not begin with blanks.

Strings containing commas must be enclosed in quotation marks because commas are treated as special characters by BASIC. They are used to separate multiple items entered in response to an INPUT statement containing more than one variable in the input list. In addition, if the last string on a line of input being entered in a list via a MAT INPUT statement ends with an ampersand (&), the string must be enclosed in quotation marks.

A string in a DATA statement must be enclosed in quotation marks if it begins with a blank, a digit, a plus sign, a minus sign, or a decimal point, or if it contains a comma or an apostrophe. Ampersands, however, do not have the special significance in DATA statements that they do in items being entered in response to INPUT statements. If strings are enclosed in quotation marks, the quotation marks are not considered to be part of the string and are ignored.

4.4.3 The RESET Statement

In DATA statements numbers and strings may be intermixed. When a numeric variable appears in a READ statement the next number appearing in the DATA statements is assigned to that numeric variable; when a string variable appears in a READ statement, the next string appearing in DATA statements is assigned to that string variable. Thus, numeric and string DATA are managed independently of each other in BASIC. A RESET statement will reset pointers for both types of data so that subsequent READ statements will reread the data. A RESET * statement will reset the pointer for numeric data. A RESET $ statement will reset only the pointer for string data.

The following program illustrates the use of RESET.

```
100 READ A$, A, B$
110 PRINT "FIRST TIME", A$, A, B$
120 DATA 1, "2APPLES", PEARS
130 RESET
140 READ C$
150 PRINT "SECOND TIME", C$
160 END
```

Running this program produces the following output:

```
FIRST TIME      2APPLES      1      PEARS
SECOND TIME     2APPLES

DONE
```

4.4.4 String Lists and String Tables

BASIC can also operate on multiple strings arranged either as lists or as tables. These entities are denoted by a single letter, followed by a dollar sign, followed by one or two subscripts enclosed in parentheses. Thus A$(3) denotes the third string in a list of strings A$. Similarly B$(4,5) denotes a string in the 4th row and 5th column of a table of strings B$. A$ cannot be both a string list and a string table in the same program.

A DIM statement such as

```
100 DIM A$(25)
```

is required if any subscript will exceed 10. Individual entries of string lists or string tables can be assigned in LET statements as in the following example.

```
220 LET T$ = A$(J+1)
230 LET A$(J+1) = A$(J)
```

4.4.5 Standard Functions Regarding Strings

The functions ASC, LEN, CHR$ and SEG$ may be used in LET as well as PRINT statements. However, ASC and LEN must not be used in expressions.

The ASC Function

It is awkward to memorize the correspondence between numbers and graphics defined by the ASCII code. Rather than being forced to remember that A corresponds to 65, the programmer can make use of the ASC function and write ASC("A").

The function will take a string as an argument and deliver a number as a result.

Example:

```
10 PRINT ASC("A")
```

The LEN Function

The LEN function takes a string as an argument and returns the number of characters as a result.

Example:

```
10 PRINT LEN(X$)
```

The CHR$ Function

CHR$(Z) delivers a one-character string which corresponds to the numeric value of the expression Z. According to ASCII code as outlined in Section 7.5 the maximum value of Z is normally 127. However, as far as printing graphics is concerned, characters are equivalent modulo 128; that is, the remainder when the number is divided by 128 is used. For example, 511 = 127 modulo 128. So, CHR$(511) = CHR$(127). A single line statement which will print a quotation mark follows.

```
100 PRINT CHR$ (34)
```

The SEG$ Function

SEG$(A$, X, Y) takes a string and two expressions as arguments, and returns a substring as a result. The substring starts at character no. X from the input string, and ends at character no. Y.

```
50 PRINT SEG$(A$, 3, 3)
```

will print the third character from the string A$.

4.4.6 An Operator for Combining Strings

One operation has been defined as working specifically on strings. This is concatenation, denoted by the ampersand (&). Concatenation puts one string directly after another, without any intervening characters.

Example:

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

Running this program causes "TIME-SHARING" to appear on the terminal. It is possible to use string constants in quotation marks in place of string variables with the & operator, if desired.

Concatenation may appear in LET and PRINT statements. In LET however, only two strings may be combined in one operation.

4.5 Formatting Output

When you write BASIC programs to prepare reports, graphs, tables, and other formatted (or specially arranged) output, it is important that you will be able to control output format very closely. This section describes statements which permit construction of neatly alinged tables, labels, and so on.

4.5.1 Commas in PRINT Lists

The terminal line is considered to be divided into five zones of 15 characters each. Each line begins with column zero. When multiple items appear in a PRINT list separated by commas, the first item is printed starting at the beginning of the first zone (column 0), the second at the next zone (column 15), etc. The comma can be considered to cause the terminal print head to space up the next zone preparatory to printing. If the fifth zone has just been filled, the terminal print head will move to the first print zone of the next line. Thus the statement

```
100  PRINT , , , , "COL60"
```

will print the five character "COL60" beginning at column 60, the beginning of the fifth zone.

If a PRINT list ends in a comma, the terminal print head simply spaces up to the next 15-character zone and does not move to the beginning of a new line in preperation for the next PRINT statement unless the fifth zone has been filled.

For example, the program

```
100 FOR I = 1 TO 15
110   PRINT I,
120 NEXT I
130 END
```

will cause the following output to be printed:

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

DONE
```

4.5.2 Vacuous PRINT Statements

A PRINT statement which does not end in any special punctuation mark, such as a comma, will print the information in the PRINT list, and the terminal will be prepared so that further output will begin at the beginning of the next line. Thus a vacuous PRINT statement such as

```
100 PRINT
```

will simply advance the paper one line, leaving a blank line if the terminal print head is already at the beginning of a line. It can be used to cause the completion of a partially filled line as illustrated in the following program.

```
100 FOR I = 1 TO 4
110     FOR J = 1 TO I
120         LET B(I, J) = I
130         PRINT B(I, J),
140     NEXT J
150     PRINT
160 NEXT I
170 END
```

This program will print B(1, 1) on the first line. Without line 150, the terminal print head would then go on printing B(2, 1), B(2, 2) on the same line. Line 150 directs the terminal print head to start at the beginning of a new line after printing the highest J value for a given I. Thus, items are printed in a triangular format. Output from the preceding program follows:

```
1
2       2
3       3       3
4       4       4       4

DONE
```

4.5.3 Packed PRINT Lists

Using the comma to separate items in PRINT lists, you will find that it is not possible to print more than five numbers or strings on one line. A semicolon may be used to print items closely packed on a line. For example, the program

```
100 FOR I = 1 TO 15
110     PRINT I;
120 NEXT I
130 END
```

will cause the following output to be printed.

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

DONE
```

To determine what will be printed using the semicolon separator, it is necessary to know how strings and numbers are printed. In general, when you are using the semicolon to format output, no blanks will be output other than those automatically output when a string or number is printed as described in the following section.

4.5.4 Printing Formats for Numbers and Strings

This section described the spacing of numbers and strings as they are printed by a simple PRINT statement.

Strings are printed just as they are, with no leading or trailing spaces. A space is printed after the right-most digit of a number; negative numbers are preceded by a minus sign, and positive numbers are preceded by a blank.

The number of spaces which will be occupied by the decimal representation of a number varies according to the magnitude and type (integer or non-integer) of the number. The following discussion of how numbers are printed will help in determining the expected printed output.

Numbers may be printed using one of three notations:

I — A number printed using integer notation is printed without a decimal point and contains from 1 to 6 digits. (For example, twenty printed as 20 is in integer notation.)

II — A number printed in fractional notation contains from one to six digits and a decimal point. Trailing (right-most) zeros are dropped, and a number less than one is printed with a zero to the left of the decimal point. (For example, twenty printed as 20. is in fractional notation.)

III A number printed in scientific notation has the following form.

Z E+Y or Z E-Y

where Z is a number greater than 1 and less than 10 printed in fractional notation (II) and Y is the appropriate power of 10.

Numbers are printed in one of these notations according to their magnitude and type. All numbers are rounded off to six significant digits.

1) An integer whose absolute value is less than 10 ↑6(1000 000) is printed in integer notation (I).

2) An integer whose absolute value is greater than or equal to 10 ↑6 is printed in scientific notation (III).

3) A number whose absolute value is greater than or equal to .1 and less than 999999, and which is not an integer is printed in fractional notation (II).

4) A number whose absolute value is less than .1 which can be expressed using 6 digits after trailing (right-most) zeros are are dropped is printed in fractional notation (II).

5) A number whose absolute value is less than 1 which does not satisfy the condition in (4) is printed in scientific notation (III).

6) A number whose absolute value is greater than 999999 and which is not an integer is printed in scientific notation (III).

By printing powers of two, the following program illustrates how numbers falling into each of these six categories are printed.

```
100 FOR I = 1 TO 30 STEP 3
110     PRINT 2↑(-I), I, 2↑I
120 NEXT I
130 END
```

This program yields the following printout.

```
0.5             1       2
0.0625          4       16
7.8125E-03      7       128
9.76562E-04     10      1024
1.2207E-04      13      8192
1.52588E-05     16      65536
1.90735E-06     19      524288
2.38419E-07     22      4.1943E+06
2.98023E-08     25      3.35544E+07
3.72529E-09     28      2.68435E+08
DONE
```

4.5.5 The TAB Function

In addition to the previously described standard means of controlling printing formats, it is possible to set up nonstandard columns and to print material in special forms. The TAB function is one way of producing such specialized output.

If the first column in which information can be printed on the terminal is labeled column 0, then the comma can be thought of as performing a tabulation to the next tab-stop; these stops are set at columns 15, 30, 45, and 60. There is a way to tab to any desired column using the TAB function. The TAB function can appear only in a PRINT list. It does not cause anything to be printed, but it simply positions the terminal print head to begin printing in the column denoted by the argument of the TAB function. For example,

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

will cause the X-value to be printed starting in column 0, the Y-value in column 12, and the Z-value in column 27.

The TAB function may contain any formula as its argument. The value of the formula is computed, and the integer part is taken. This number is treated modulo the current margin setting to obtain a column number. The terminal print head then spaces to this position; in the event that it has already passed this position the TAB is ignored.

If you wish to print two series of characters on the same line or to overprint a line, you may do so by printing CHR$(13), a carriage return character (13 is the ASCII code for a carriage return); a semicolon should separate this character from the next item to be printed. The following line will print two asterisks on the same line. The first one printed is in column 11; the second one is printed in column 9.

```
100 PRINT TAB (11); "*"; CHR$(13); TAB (9); "*"
```

If you use this option, it may be necessary to issue not one carriage return, but two (or a carriage return followed by a rubout) before the second TAB to allow the terminal print head to return to the physical left margin of the terminal before beginning the second tabulation.

4.5.6 The MARGIN Statement

The MARGIN statement sets the maximum number of characters which may be printed on a line. The margin is initially set to 70. If a line is partially filled and there is not enough room remaining for an output string, the line will be printed and the output begin on a new line. If the output string is so long that there is not enough room even on a complete single line, as much as will fit will be printed on that line; the rest of the output string will be continued on the next line, and the process will be repeated as many times as necessary to print the entire string. Even if a line is partially filled when a MARGIN statement is executed, the statement will change the margin for the rest of the line.

The program MARGIN is illustrative.

```
100 MARGIN 10
110 PRINT "IN THIS "; "SECTION"
120 PRINT "THE MARGIN STATEMENT IS DISCUSSED."
130 PRINT "THE ";
140 MARGIN 75
150 PRINT "MARGIN ";
160 PRINT "CHANGES AS SOON AS THE MARGIN STATEMENT"
170 PRINT "IS EXECUTED, EVEN IF THE LINE IS PARTIALLY
180 PRINT "FILLED."
190 END
```

The output is:

```
IN THIS
SECTION
THE MARGIN
STATEMENT
IS DISCUS
SED.
THE MARGIN CHANGES AS SOON AS THE MARGIN STATEMENT
IS EXECUTED, EVEN IF THE LINE IS PARTIALLY FILLED.
```

With a margin of ten, the word "SECTION" does not fit on the first line of output, so it is printed on the second line: The string to be printed by line 120 will not fit on one line, so it is printed 10 characters per line until the string is exhausted. The margin statement in line 140 takes effect immediately, so the line started by the PRINT statement in line 130 with a margin of 10 has a margin of 75 when line 150 is encountered.

4.5.7 The PRINT USING Statement

In addition to the standard formats defined above, it is possible to define your own formats and use them. This feature allows you to print numbers in columns so that decimal points line up and to produce tables easily.

Instead of employing

```
100 PRINT A, B, C$
```

you can modify the PRINT statement to

```
100 PRINT USING X$, A, B, C$
```

Here, X$ contains a "picture" of the line to be printed. Spaces where the values of the variables are to be inserted are marked by special conventions. Literal labels may also be part of the picture string. If desired, a string constant may be used in place of X$, and constant information may be printed in place of variables.

A sample use of the PRINT USING statement follows:

```
100 LET A = 20
110 LET B = 15
120 LET C$ = "CDE"
130 LET X$ = "A IS- #, B IS - #, AND THE STRING C IS  # #"
140 PRINT USING X$, A, B, C$
150 END
```

When this program is run,

```
A IS 20,  B IS 15,  AND THE STRING C IS CDE
```

appears on the terminal.

There are 8 special characters for defining PRINT USING fields or areas where variables are to be printed. These 8 characters are: # - + . ^ S < and > . The number sign, " # ", reserves a place for one character in a field, but it cannot be used at the beginning of a field.

The effect of these characters is summarized in the following chart:

Sign	Valid in	Effect
-	Numeric fields only	Start field; print floating minus for negative numbers.*
+	Numeric fields only	Start field; print floating plus or minus as appropriate.*
.	Numeric fields	Mark decimal place; when used outside numeric fields it is printed literally.
$	Numeric fields only	Start field; print floating dollar sign; must be followed by + or - .
^	Numeric fields only	Specify exponent field; must be in group of 5.
<	String fields only	Start field; print string left-justified.
>	String field only	Start field; print string right-justified.
#	Any field	Place holder.

* These characters may be immediately preceded by "$".

A numeric field, an area in which a number is to be printed, begins with either "-" or "+". If "+" is used, a plus or minus sign will be printed just before the left-most digit of the number, depending on whether it is positive or negative. If "-" is used, there will be a sign only before a negative number. A "-" alone can be used to specify a one-character numeric field; a non-negative number less than 10 may be printed using such a format. Additional places in a numeric field can be specified by repeating "#" as many times as desired.

Numbers are rounded and truncated before they are printed. They are printed right-justified in the field, so that the integer digits line up on successive lines. A sample program NUMBERS1 is:

```
100 PRINT USING "LINE 100 - # # #", 200.34
110 PRINT USING "LINE 110 - # # #", 20.03
120 PRINT USING "LINE 120 - # # #", 2.00
130 END
```

Output from a run of NUMBERS1 follows:

```
LINE 100 200
LINE 110  20
LINE 120   2
```

If a number has too many digits to be printed in the field given, asterisks are supplied instead. So with the format "- #", the number "200" appears as "**" on the terminal when the statement is executed; the field "- #" could be used to print numbers in the range $-10 < X < 100$. If a field has more places than there are significant digits allowed in BASIC, question marks are supplied for the digits which might be misleading. In an eleven-space field, a number input as "1111111111" is printed as "111111111?".

If numbers are not to be printed as integers, a period is used to mark the location of the decimal point in the numeric field. (A . is interpreted as a character to be printed literally if it is not in a numeric field.) If the number "20.356" is printed with the format "- # #. # #, two decimal places are given, the number is rounded and truncated accordingly, and the result is printed as "20.36". Again, the number is right-justified in the field so that the decimal points line up on successive lines. For numbers in the range $-1 < X < +1$, a leading zero is provided. As an example, consider the program NUMBERS2 as follows:

```
100  PRINT USING "LINE 100 - # # . # #", 20.356
110  PRINT USING "LINE 110 - # # . # #", 2.0356
120  PRINT USING "LINE 120 - # # . # #", .20356
```

A run of NUMBERS2 produces the following output:

```
LINE 100   20.36
LINE 110    2.04
LINE 120    0.20
```

To print a number with an exponent, put a group of five up-arrows (the symbol for exponentiation) into the format string; the count of 5 is mandatory. If 2.356 is printed with the format "- # . # ↑↑↑↑↑ ", " 2.4 E+00" appears. With the ↑ format, one space is reserved for a possible sign, and the number begins with the next space. The exponent is adjusted to compensate for any shifting which occurs. With a field "- # # # . # # ↑↑ ↑↑↑ ", the number 2.356 appears as "235.60 E-02", and the number 20.356 is printed "203.56 E-01".

The following program NUMBERS2 exemplifies these conventions:

```
100 PRINT USING "LINE 100 - # # . # # ↑ ↑ ↑ ↑↑ , 203.56
110 PRINT USING "LINE 110 - # # . # # ↑ ↑ ↑ ↑↑ , 20.356
120 PRINT USING "LINE 120 - # # . # # ↑ ↑ ↑ ↑↑ . 2.0356
130 PRINT USING "LINE 130 - # # . # # ↑ ↑ ↑ ↑↑ . .20356
140 END
```

Running this program gives the following output:

```
LINE 100  20.36 E+01
LINE 110  20.36 E+00
LINE 120  20.36 E-01
LINE 130  20.36 E-02
```

An exception to the rule that a numeric field must begin with a "+" or "-" is the option of preceding these two characters by a "$". The use of a dollar sign forces the printing of a dollar sign just before the first digit or sign of the number.

It is possible to have literal information, including commas, in a format field. In particular, it is possible to include blanks to group digits conveniently. With the format string "- # # , # # # . # # ", "99999" will be printed as "99,999.00". Since a field must begin with - + $ < or > , it is possible to interrupt it with literal information. This literal information must not include any of the special characters, except that a period in a non-numeric field is printed literally.

The field for printing a string must begin with either < or > . These characters are valid only in string fields, just as +, -, ↑ , and $ are valid only in numeric fields. A < causes the string to be printed left-justified in the field specified; if necessary, the field is filled with blanks or the string truncated from the right. As with numeric fields, " # " serves to hold a place for printing. Left-justification of strings is shown in the following example; program STRINGS1:

```
100 PRINT USING "LINE 100 < # # ", "AB"
110 PRINT USING "LINE 110 < # # ", "ABC"
120 PRINT USING "LINE 120 < # # ", "ABCD"
130 END
```

Running this program gives:

```
LINE 100 AB
LINE 110 ABC
LINE 120 ABC
```

A > sign causes the string to be printed right-justified in the field specified. If necessary, the string is preceded by enough blanks to fill the field or is truncated from the left. Altering the last program to STRINGS2:

```
100 PRINT USING "LINE 100 > # # ", "AB"
110 PRINT USING "LINE 110 > # # ", "ABC
120 PRINT USING "LINE 120 > # # ", "ABCD"
130 END
```

we get

```
LINE 100  AB
LINE 110 ABC
LINE 120 BCD
```

Again literal information can be included within the field; with "< #X##", the string "ABCD"is printed as "ABXCD".

Note that it is not possible to specify any of the special characters # - + ↑ $ < or > as material to be printed literally. If these special characters are to appear in the output, they can be specified as constants to be printed in separate fields. To print a "+", the following statement suffices.

```
900  PRINT USING "<", "+"
```

The items to be printed according to the defined format must be separated by commas, and a comma must separate the USING string from the variables. The order of numeric and string variables to be printed must match the order of the types in the format string. For example,

```
900 PRINT USING "- # . # # < # # # ", "ABCD", 23.4
```

causes an error message and termination of the execution because the field types in the format string do not match the types of information to be printed. A string cannot be printed with a numeric field, nor can a number be printed with a string field.

If there are fewer variables in the list of a PRINT USING statement than there are fields specified in the format string, the extra fields are not used. On the other hand, if there are more variables than fields, the format string is used again, starting on a new line. If the nformation to be printed will not fit on a single line, the part of the format not used on the first line is continued on the second line, and so on until all the items in the list are printed.

Ending a PRINT USING statement with a semicolon causes suppression of the carriage return and line feed characters after all items in the list have been printed as described in Section 2 7.3 for the simple PRINT statement. Using this option, you may complete a partially filled line with subsequent PRINT or PRINT USING statements. You may not end a PRINT USING statement with a comma as you can a simple PRINT statement.

BANKUSING is a program which illustrates that output can be arranged in columns so that the decimal points line up normally. Additionally, a dollar sign can be printed immediately before each amount.

```
100 PRINT "ITEM", "  AMOUNT", " BALANCE"
105 PRINT
110 LET C=0
120 LET D=0
130 REM C COUNTS THE NUMBER OF CHECKS
135 REM D COUNTS THE NUMBER OF DEPOSITS
140 READ B
141 REM
142 REM SET UP FORMAT STRINGS IN F$ AND G$
143 LET F$="<# # # # # # # # #      $- # # #.# #     $+# # # #.# #"
144 LET G$="<# # # # # # # #                        $-# # # #.# #"
145 REM
146 REM A SPECIAL FORMAT IS NEEDED FOR THE
147 REM OPENING AND CLOSING BALANCES, WHICH
148 REM HAVE NO TRANSACTIONS
149 REM
150 PRINT USING G$, "OPENING",B
160 REM
170 READ T
180 IF T=0 THEN 400
190 IF T<0 THEN 300
200 REM
210 REM HERE FOR A DEPOSIT
220 LET D=D+1
230 LET B=B+T
240 PRINT USING F$, "DEPOSIT", T,B
250 GOTO 170
260 REM
```

```
300 REM HERE FOR A CHECK
310 LET C=C+1
320 LET B=B+T
330 PRINT USING F$, "CHECK", -T, B
340 IF B >=0 THEN 170
350 LET B=B-1
360 PRINT USING F$, "OVERDRAFT", 1, B
370 GO TO 170
380 REM
400 REM HERE FOR CLOSING
410 LET S=.03 * D + .06 * C + .60
420 LET B=B-S
430 PRINT USING F$, "SERVICE", S, B
440 PRINT USING G$, "CLOSING", B
470 REM
500 DATA 100.00
510 DATA -23.75, -10.40, 50.00, -7.25, -42.50
520 DATA -45.67, -22.95, 40.00, -50.33, 66.75, 0.00
999 END
```

A run of this program. BANKUSNG, is below:

```
ITEM          AMOUNT          BALANCE

OPENING                       $100.00
CHECK         $23.75          $+76.25
CHECK         $10.40          $+65.85
DEPOSIT       $50.00         $+115.85
CHECK          $7.25         $+108.60
CHECK         $42.50          $+66.10
CHECK         $45.67          $+20.43
CHECK         $22.95           $-2.52
OVERDR:        $1.00           $-3.52
DEPOSIT       $40.00          $+36.48
CHECK         $50.33          $-13.85
OVERDR.        $1.00          $-14.85
DEPOSIT       $66.75          $+51.90
SERVICE        $1.11          $+50.79
CLOSING                        $50.79
```

ADDITIONAL INFORMATION:

1) A shorthand of Print Using is: USING.

2) The actual MARGIN is also valid for PRINT USING.

3) It is possible to let the output go to a specified file as with the PRINT statement.

4.6 Input Control

There are some occasions when a user wishes to override the normal BASIC input conventions. For example, commas usually are used to separate a fixed number of entries on a line. The following statements allow somewhat greater flexibility.

4.6.1 The LINPUT Statement

If a program calls for data to be entered from the terminal using an INPUT statement, and the data consists of strings containing such characters as quotation marks, leading blanks, ampersands, or commas, then the data used in the BASIC computation may not be the ones desired, for BASIC normally treats such characters in special ways. The LINPUT (remember it as "line-input") statement provides for the entering of an arbitrary sequence of characters into a single string. The characters typed may consist of any ASCII characters, other than a carriage return, which terminates the string; the carriage return character is not included in the string. An example of a LINPUT statement appears in the following program, which counts the number of commas in the input string.

```
100  LINPUT A$
110  LET N = 0
115  LET X = LEN(A$)
120  FOR I = 1 TO X
130     LET B$ = SEG$ (A$, I, I)
140     IF B$ < >"," THEN 160
150     LET N = N+1
160  NEXT I
170  PRINT "THERE ARE"; N; "COMMAS IN THIS LINE."
180  END
```

A run of the program follows.

```
? A,B,C,,D,E
THERE ARE FIVE COMMAS IN THIS LINE.
```

More than one variable may follow the word LINPUT if the variable names are separated by commas. A new ? appears for each variable in the list.

4.6.2 The MAT INPUT Statement

The MAT INPUT statement allows the user to enter data when the program does not know how much data will be input. This feature circumvents cumbersome programs such as the following, which is designed to perform the simple task of adding up a few numbers typed in from the terminal

```
100  LET T = 0
105  INPUT N
110  LET T = T + N
120  IF N < >0 THEN 105
130  PRINT "THEN TOTAL IS"; T
140  END
```

To use such an awkward program, you must type one number and one carriage return in response to each question mark which is printed by the INPUT statement. When a zero is entered, the program assumes that all the numbers have been entered, and the total is printed. Besides being time-consuming, intermediate zeros cannot be entered.

The following program using the MAT INPUT statement is much more convenient to use and performs the same function as the previous program.

```
100 DIM A(100)
105 LET T = 0
110 MAT INPUT A
120 FOR I = 1 TO NUM
130     LET T = T + A(I)
140 NEXT I
150 PRINT "THE TOTAL IS"; T
160 END
```

In response to the MAT INPUT statement in line 110, the user may type any number of numbers separated by commas. When the input line is terminated with a carriage return, the first number entered is in A(1), the second is in A(2), and so on. The number of numbers entered is made available by the function NUM. This function has no arguments and will deliver the number of entries until a new MAT INPUT statement is executed.

Zero, one, or any number of entries may appear on a line, the only limit being the size of the line. If one wishes to enter more numbers than can be typed on one line, it is possible to continue typing on additional lines. If the last number on a line is followed by an ampersand (&) with no preceding comma and then by a carriage return, BASIC will accept the input typed so far and then print a question mark so that data may be continued on the following line. Of course, if more than 10 numbers are to be entered using a MAT INPUT statement, a DIM statement must be provided in order to reserve sufficient storage.

The MAT INPUT statement may also be used to enter strings into a list. Rules for enclosing the strings in quotation marks are the same as those given in Section 4.4.2 for the INPUT statement with this addition: the last string entered on a line in response to a MAT INPUT statement must be enclosed in quotation marks if its last character is an ampersand (&).

The possibility of having variable amounts of input is available only with lists. If the MAT INPUT statement is used with a table, an item must be entered for each element in the table; variable input is not allowed. See Section 6.5.2 for more information on the MAT INPUT statement.

4.7 Program Organization Statements

When larger BASIC programs are written, they should not be looked upon as a simple series of statements. They should be organized into units analogous to blocks or sections or paragraphs, so that overall action of the program can be managed in terms of "building blocks" of statements. Once these blocks of statements are written and checked, they can be utilized by a programmer who knows only the function they perform, without his having to bother with individual, detailed statements.

BASIC is a language which is designed to be understandable both by machines and by human beings. A program must be understandable to the machine if it is to perform a computation. A program must be understandable to a human being if he is to be able to verify its correctness, improve the technique, change the theoretical basis of the technique, or explain its value to others. Also, when programs are being developed, they do something--not necessarily what is finally desired; all programs do something, even if it is stopping immediately. It must be possible to determine how a program does what it does, even when it is incorrect. English-language comments (or other natural-language comments) can be incorporated into the body of the text of a BASIC program in order to improve its readability and to aid in its interpretation. These comments do not interfere with the operation of the BASIC program.

4.7.1 The Apostrophe Convention

A comment may appear on the same line as a BASIC statement if the comment follows the statement and is separated from it by an apostrophe. This is especially useful for explaining the intent of a single BASIC statement when the importance of that statement is not necessarily clear from the BASIC statement alone. A comment may appear on a line by itself if it is preceded by an apostrophe as shown in the following program segment.

```
100 IF ABS(X) <= 1 THEN 130  'PREVENT NEG SQ ROOT IN 130
110 PRINT "ABS (X) IS GREATER THAN I IN LINE 100."
115 ' AVOID LINE 130 WITH A GO TO STATEMENT
120 GO TO 140
130 LET Y = SQR (1 - X * X)
140 LET Z = Z - Y
```

4.7.2 More about the REM Statement

As was pointed out in Section 4.3.3, if the first three characters following the line number of a BASIC statement are REM, then any remarks whatsoever may follow on that line. REM statements may be used to convey the function of a block of statements in a program. Knowing the purpose of the BASIC program (or the purpose of each part of it) facilitates checking each of the BASIC statements to verify that the program is correct. Well-written REM statements greatly increase the value of a BASIC program to other users by making the intent of the programmer known, i.e., what the program as a unit is supposed to do and how different parts of the program work toward this end.

Since REM statements have line numbers, they can be referred to in GO TO statements or other statements which cause a transfer of control such as the ON...GO TO and IF...THEN statements. It is especially appealing to transfer to a REM statement which describes the purpose of a following block of code. The example in Section 4.8.1 illustrates how blank lines can be used to set off a REMARK or other BASIC statement. Blank lines are ignored by BASIC and are used to improve the readability of programs.

4.8 Subroutines

In BASIC programs, it often happens that similar calculations must be carried out at several places in the computation. We denote a related group of BASIC statements required to carry out such a calculation as a subroutine. It would be tedious and wasteful to have to copy the statements of the subroutine at every place in the entire BASIC program that such a calculation was to be performed. The GOSUB statement provides a way to transfer control to a subroutine. Control returns to the statement following the GOSUB when a RETURN statement is reached in the subroutine. Alternatively, the ON...GOSUB statement allows branching to one of several subroutines and the IF...GOSUB statement allows a conditional subroutine jump.

4.8.1 The GOSUB and RETURN Statements

The GOSUB and RETURN statements are illustrated in the following example where the subroutine in lines 300-410 calculates the greatest common divisor of two numbers X and Y. The program uses this subroutine to calculate the greatest common divisor of three numbers A, B, and C, relying on the fact that GCD(A,B,C) = GCD(GCD(A,B),C).

```
110 PRINT "A","B","C","GCD"
120 READ A,B,C
130 LET X=A
140 LET Y=B
150 GOSUB 300
160 LET X=G
170 LET Y=C
180 GOSUB 300
190 PRINT A,B,C,G
200 GO TO 120
210 DATA 60,90,120
220 DATA 38456,64872,98765
230 DATA 32,384,72
250 '
```

```
300  REM  SUBROUTINE TO CALCULATE GCD
305  LET Q=INT(X/Y)
310  LET R=X-Q*Y
320  IF R=0 THEN 400
330  LET X=Y
340  LET Y=R
350  GO TO 300
400  LET G=Y
410  RETURN 'TO LINE 160 OR 190
420  END
```

When the program is run, X and Y are set equal to A and B. Line 150 contains a GOSUB to line 300. This is the beginning of a calculation which sets G equal to the greatest common divisor of X and Y. Line 410 is the RETURN statement which returns to 160, the line following the GOSUB. Subsequently X and Y are given the values of G and C in order to GOSUB to the GCD subroutine once more. Upon return to 190, the line after the second GOSUB, the answers are printed and the process recycles. In operation, the statement

```
180  GOSUB 300
```

records information about the location of the GOSUB before transferring control to line 300. This is done in such a way that a statement like

```
410  RETURN
```

uses the information stored by the GOSUB statement to return control to the line directly following the GOSUB. Consequently, a subroutine may have many RETURN statements in it, but the first one which is actually encountered causes control to be returned to the main part of the program.

A GOSUB may be executed inside a subroutine to call still another subroutine. In this nested subroutine arrangement, the first RETURN statement to be executed returns control one level to the statement following the most recently executed GOSUB. The next RETURN statement returns control to the statement following the previously executed GOSUB, and so on.

4.8.2 The ON...GOSUB Statement

The ON...GOSUB statement provides a way of transferring control to one of several subroutines. The statement

```
100  ON X-1 GOSUB 700, 800, 900
```

will cause execution of the subroutine beginning in line 700 if the value of X-1 is 1, execution of the subroutine beginning in line 800 if the value of X-1 is 2, and execution of the subroutine beginning in line 900 if the value of X-1 is 3.

The expression "X-1" could have been any arithmetic expression, including a simple variable. The value of this expression must not be less than 1 and not greater than the number of line numbers listed; if so control is transferred to the next line of the program. If the value is not an integer, it will be truncated to an integer.
When a RETURN statement is encountered, control is returned to the statement following the ON...GOSUB statement.

4.8.3 The IF...GOSUB Statement

The IF...GOSUB statement provides a way of transferring control to a subroutine if some specified condition is met. The statement

```
100 IF A$ = "MARRIED" GOSUB 900
```

will transfer control to line 900 if the condition is true.

The condition may be of either a numeric or a string type.

When a RETURN statement is encountered, control is returned to the statement following the IF...GOSUB statement.

4.9 The DEF Statement

BASIC has a number of built-in functions, such as SIN, LOG, SQR, etc. If the user requires an extension to this set of functions, he has the ability to write a definition for a new function in BASIC using a DEF statement.

4.9.1 One Line DEF Statements

Sometimes a function definition can be written in a single BASIC statement. Suppose an arcsine function is required.

```
100 DEF FNA(X) = ATN(X / SQR(1 - X*X))
110 PRINT FNA(.707)
120 END
```

Line 100 defines the new arcsine function. Defined functions are given three-character names, the first two letters of which are FN and the third is alphabetic. In the definition of FNA(X), the variable X is not related to any variable of the same name elsewhere in the program. The DEF statement simply defines the function and does not cause any calculation to be carried out; the variable X is called a dummy argument. The appearance of FNA in some other place in the BASIC program (this is known as the place where the function is called) causes the calculation denoted in the DEF statement to be executed. When the function is called, the value of the argument of the function (.707 in the above example) is substituted for the dummy argument throughout the definition of the function.

DEF statements may appear anywhere in a program and may define functions of more than one variable. For example:

```
100 LET D1 = FNR (201.83 , 199.01)
110 PRINT D1
120 DEF FNR(X,Y) = SQR(X*X + Y*Y)
130 END
```

When a function of more than one variable is defined, the list of dummy arguments is separated by commas.

DEF statements may involve both dummy arguments and variables which have the same meaning as elsewhere in the program. In the following example

```
100 DEF FNX(X,Y) = X*COS(T) + Y*SIN(T)
110 DEF FNY(X,Y) = -X*SIN(T) + Y*COS(T)
120 LET T = 1.7  'ANGLE IN RADIANS
130 INPUT A, B
140 PRINT "ROTATED", FNX(A,B), FNY(A,B)
150 GO TO 130
160 END
```

the DEF statements involve both the dummy variables X and Y whose values depend on the arguments of the function, and a variable T which has the same value as it does elsewhere in the BASIC program. If a variable in a DEF statement is to have its current value in the program when the function is called, it is not included in the list of dummy arguments.

4.9.2 Multiple Line DEF Statements

The use of the DEF statement described above is limited to those functions which can be defined in a single BASIC arithmetic statement. Many functions cannot be computed using a single BASIC arithmetic expression, particularly those which require IF...THEN statements. The following example demonstrates the format of multiple line DEF statements and their use for a function which returns the larger of two numbers.

```
10 DEF FNM (X,Y)
20 LET FNM = X
30 IF Y < = X THEN 50
40 LET FNM = Y
50 FNEND
55 '
60 PRINT FNM(5,4), FNM(-5,-4)
70 PRINT FNM(1, FNM(2, FNM(3,0)))
80 END
```

The definition of the function extends from line 10 to line 50.

The absence of the equal sign in line 10 indicates that this is a multiple line DEF; the end of the DEF is indicated by the FNEND statement. The value which the function delivers must be stored in the variable having the same name as the function (in this case, FNM) when control reaches the FNEND statement. As illustrated in line 70, function calls may be nested. The preceding program printes the numbers 5, -4, and 3.

As with the single line function definition, variables appearing in parentheses after the function name in a multiple line definition are called dummy arguments, and values are substituted for these arguments when the function is called. Variables not listed in the DEF statement will use their current value. There must not be a transfer from inside a multiple line DEF to outside, nor vice versa. Function definitions may not be nested. Naming conventions are the same as for single line definitions. Multiple line function definitions may be placed anywhere in a program.

If a value is not stored (as in line 40 above) for the function when control reaches the FNEND statement, a value of zero is returned when the function is called. Any variable assignments made to variables other than the dummy arguments of the function within the scope of a multiple line definition affect the values of variables of the same name appearing elsewhere in the program.

Note that strings are not allowed as parameters.

5 FILES IN BASIC

5.1 Introduction

Files are the retrievable units in which information is stored. All the programs discussed so far in this manual are examples of files. These files are printed on paper and you can retrieve the information by reading them. Another example is a program punched on paper tape. The paper tape format is easily transferred to a computer equipped with a paper tape reader.

Files are classified according to how the information is accessed.

Sequential files are accessed one character after the other. In Chapter 3 the saving and retrieval of program files are explained. These files are sequential files, and because the files are used with input and output, the format used is the format suitable for the pheripheral in question. Accordingly such files are called Terminal-format files.

Data in random access files are accessed using an address. If data are used in random manner, retrieval using an address is normally much faster than sequential searching. In BASIC random files are used to hold data arrays too big for the core available and data are only manipulated using BASIC programs.

5.2 Terminal Format Files

The two major uses of terminal format files, in addition to programs, are the initial storage of data to be used as input for a program, and the storage in listable form of the output from a program. The use of a file to store the input data for a program is discussed first in this chapter.

5.2.1 Reading a Terminal Format File from a Program

Throughout the next few sections of this chapter, several versions of the same fundamental program will illustrate the use of the statements related to terminal format files. This program computes an average grade for each of several students in a group.

The first version of this program, AVERAGE1, uses data stored in a terminal format file called GRADES.

The main advantage of storing the input data in a separate file, as opposed to storing it in the same file in DATA statements, is that files can contain much more data than can a program. For practical purposes, there is almost no limit to the number of students the program can process in one run. There are strict limits on the length of a program to be compiled and these limits include the DATA statements. Another advantage is that since the program file is never modified (as it would have to be if DATA statements were used), there is no chance of the program itself being inadvertently changed during the typing of a new data set.

A listing of AVERAGE1 follows:

```
100 REM  PROGRAM NAME--AVERAGE1
110 '
120 REM  THIS PROGRAM COMPUTES AVERAGE GRADES FOR
130 REM  A SET OF STUDENTS.  EACH STUDENT IS ASSUMED
140 REM  TO HAVE THE SAME NUMBER OF INDIVIDUAL
150 REM  GRADES TO BE AVERAGED.  THE DATA IS IN A
160 REM  TERMINAL FORMAT FILE CALLED "GRADES".
170 REM  THE FIRST LINE CONTAINS S, THE NUMBER OF
180 REM  STUDENTS, AND G, THE NUMBER OF GRADES PER
190 REM  STUDENT.  THE REST OF THE FILE CONSISTS OF
200 REM  S SETS OF (G+1) LINES.  THE FIRST LINE IN A SET
210 REM  CONTAINS THE NAME OF A STUDENT, AND THE
220 REM  FOLLOWING G LINES IN THE SET EACH CONTAIN
230 REM  ONE OF THE STUDENT'S GRADES.
240 '
250 OPEN # 1 : FOR INPUT "GRADES"
260 PRINT "NAME", "AVERAGE"
270 PRINT
280 INPUT  # 1 : S,G
290 FOR I = 1 TO S
300 LET A = 0
310 INPUT  # 1 : N$
320 FOR J = 1 TO G
330 INPUT # 1 : X
340 LET A = A + X
350 NEXT J
360 LET A = A/G
370 PRINT N$,A
380 NEXT I
390 CLOSE # 1 :
400 END
```

There are three new statements in this program, the OPEN# statement, INPUT# statement and the CLOSE# statement. The OPEN# statement assigns a file name to a file number. The file name may be expressed as a string constant (as in the program above) or as a string variable. Thereafter, all references to the file are made through the file number rather than the file name. There may be up to 6 open files within a program, with numbers between 0 and 5. File numbers need not be assigned sequentially: a statement assigning a file to the number 5 could precede another statement assigning a file to the number 1.

When the OPEN statement is used with a sequential file it must indicate whether the file should be used for input or for output using the arguments FOR INPUT or FOR OUTPUT.

The CLOSE# statement is used when you are finished using a file. The statement will set the file ready to be opened again, and leave an empty entry in the file table.

In AVERAGE1 only one file, GRADES, is used. The OPEN# statement assigning the file GRADES to file number 1 is in line 250. Thereafter, the file GRADES is referred to as file # 1 in lines 280, 310, and 330 of the program.

The INPUT# statement differs from the simple INPUT statement only by the inclusion of the number sign, a file number, and a colon. Any list of variables that is legitimate in a simple INPUT statement is also legitimate in an INPUT# statement.

Now let us briefly run through the whole program before going on to consider the construction of the data file GRADES. Lines 100-220 are remarks describing the program, its limitations, and instructions for using it. The OPEN# statement has already been described. Lines 260 and 270 print a heading for the output. Line 280 requests the input of two numbers, S and G, from file #1, the file GRADES. S is the numbe. of students and G is the number of grades per student. A loop indexed by I begins in line 290 and continues through line 380. The program ends after this loop has been executed S times, once for each individual whose grades are to be averaged.

Within this loop, line 300 initializes A, the variable used to store the sum of the grades for an individual. Line 310 requests the input of a string from file # 1, GRADES. This string is the name of the next individual whose grades are to be averaged. Another loop begins in line 320 and ends in line 350. This loop is executed G times, once for each grade. Within the loop indexed by J, line 330 inputs a grade, X, from GRADES and line 340 adds this grade to A, the sum of the grades so far. When this loop has been executed G times, line 360 divides the sum of the grades, A, by the number of grades, G, to get the average grade which is stored in A. Line 370 prints the name of the individual, N$, and his average, A. Then the loop indexed by I is executed for the next individual, until all averages have been computed and printed.

Now let us consider the data file. The format used in constructing a terminal format file to be read by a program is determined by the way in which the INPUT# statements are set up in the program. INPUT# statements, like simple INPUT statements, contain lists of variables to receive values. Whereas a simple INPUT statement requests the user of the program to supply these values at run time, the INPUT# statement requests the values from a terminal format file. It considers the contents of the next line in the file (beginning with the first line in the file)

as response to its request. If there are more numbers or strings in the line than were requested, the excess is ignored. If there are not the same number of items as there are variables in the INPUT list, the next line in the file is interrogated in an attempt to find more numbers or strings. If the items on the line interrogated do not correspond in type to the variables in the input list, the process is continued.

The first INPUT# statement in AVERAGE1 requests two numbers, S and G. These numbers may either be on the same line in the data file or on two different lines. The rest of the numbers and strings in GRADES must be written one per line since they will be read by INPUT# statements requesting one number at a time. If they were erroneously written more than one per line, all but the first number on each line would be ignored (as "excess data"). In an attempt to compensate for the number ignored, the computer would look for values beyond the end of the file, and the program run would terminate. The file GRADES must not have line numbers--just the data requested by the INPUT# statements in the program. The following is a listing of the file GRADES as written for use with AVERAGE1. Note that when more than one item is listed on the same line, the items are separated by commas, as in the first line of GRADES.

```
3,4
GERALD FRIEND
78
86
61
90
PHILIP CLOUGH
66
87
88
91
ADA SHAW
56
77
81
85
```

This file could be created by using the editor QED. (For information about QED consult the QED Users Manual.)

The following is a run of AVERAGE1 using the data in the file GRADES.

```
AVERAGE1
NAME              AVERAGE
GERALD FRIEND     78.75
PHILIP CLOUGH     83
ADA SHAW          74.75

DONE
```

5.2.2 Writing a Terminal-format File from a Program

In this section we will consider how to alter the program AVERAGE1 so that it writes its output into a terminal-format file instead of printing it on the terminal. Using a file in this manner allows the user to obtain multiple copies of the output without rerunning the program. In addition, if there is a lot of output, it is often more convenient and possibly faster to direct the output to a file and then list the file than to print the output directly on the terminal.

Two changes need to be made in AVERAGE1; first, another OPEN# statement must be added to assign the output file to a file number; and second, the simple PRINT statement must be changed to PRINT# statements. The following program, AVERAGE2, incorporates these changes. The output is printed in a terminal-format file called AVERAGES.

```
210 REM  PROGRAM NAME--AVERAGE2
220 ,
230 REM  THIS PROGRAM IS LIKE AVERAGE1 EXCEPT THAT
240 REM  THE OUTPUT IS PRINTED IN A TERMINAL-FORMAT
250 REM  FILE CALLED "AVERAGES".
270 ,
290 OPEN # 1:  FOR INPUT "GRADES"
300 OPEN # 2:  FOR OUTPUT "AVERAGES"
310 PRINT # 2:  "NAME", "AVERAGE"
320 PRINT # 2:
330 INPUT # 1:S,G
340 FOR I = 1 TO S
350 LET A = 0
360 INPUT # 1:N$
370 FOR J = 1 TO G
380 INPUT # 1: X
390 LET A = A+X
400 NEXT J
410 LET A = A/G
420 PRINT # 2: N$,A
430 NEXT I
440 CLOSE # 1:
450 CLOSE # 2:
460 END
```

The input file GRADES is assigned to file #1 and the output file AVERAGES is assigned to file #2.

When the program is run, line 300 will set the file AVERAGES ready to receive output. If the file does not exist, it will be created. Any information in the file will be destroyed and you should do as follows if you want to save the information:

a) Enter the editor QED (see above).

b) Read the file.

c) Save the file using a new name.

It is still easier to use the NORD TSS command: COPY.

After the program AVERAGE2 has been run, you can list the file AVERAGES using COPY or the QED editor. The following printout results:

NAME	AVERAGE
GERALD FRIEND	78.75
PHILIP CLOUGH	83
ADA SHAW	74.75

Note that the output of AVERAGE2 and that of AVERAGE1 is identical; the only programming difference is that the first program prints its output to a file and AVERAGE1 prints output directly on the terminal. The format of the output in AVERAGES is the same as that of the output printed on the terminal when AVERAGE1 is run.

5.2.3 The Use of the Terminal Itself as a File

Suppose now that we wanted to rewrite AVERAGE2 so that the use of files for input and output was optional. We could write separate sections in the program to deal with each option and then to branch to the appropriate section. However, there is an easier way. Both the INPUT# and the PRINT# statements interpret a reference to file #0 as a reference to the terminal itself and in this case work exactly like the simple INPUT and PRINT statements.

The following program, AVERAGE3, is a revision of AVERAGE2 in which the user may decide whether or not he wishes to use files. In addition he may choose the names of the data and output files if he does want to use files.

```
100 REM  PROGRAM NAME--AVERAGE3
110 '
120 REM  THIS PROGRAM IS LIKE AVERAGE2 EXCEPT
130 REM  THAT THERE ARE OPTIONS FOR READING
140 REM  DATA FROM A FILE AND PRINTING THE OUTPUT
150 REM  INTO A FILE.  DATA CAN BE IN A TERMINAL-FORMAT
160 REM  FILE OR CAN BE TYPED IN AT RUN TIME.  IF THE
170 REM  DATA ARE IN A FILE, THE FORMAT IS THE SAME
180 REM  AS THAT OF "GRADES" USED IN AVERAGE1 AND
190 REM  AVERAGE2.  IF THE DATA ARE TO BE TYPED
200 REM  IN AT RUN TIME, THEY MUST BE ENTERED
210 REM  ACCORDING TO THE SAME FORMAT THEY WOULD
220 REM  HAVE WERE THEY IN A FILE.  IF OUTPUT IS
230 REM  TO GO TO A FILE, THE FILE SHOULD BE SAVED
240 REM  BEFORE THE PROGRAM IS RUN.
250 '
270 LET F1=F2=0
280 PRINT "ARE DATA IN A FILE - ANSWER NO OR GIVE FILE NAME";
290 INPUT A$
300 IF A$ = "NO" THEN 330
310 OPEN # 1 : FOR INPUT A$
320 LET F1 = 1
330 PRINT "SHOULD OUTPUT GO TO A FILE - ANSWER NO OR GIVE"
340 PRINT "FILE NAME";
350 INPUT A$
360 IF A$ = "NO" THEN 390
370 OPEN # 2 : FOR OUTPUT A$
380 LET F2 = 2
390 PRINT # F2 :
400 PRINT # F2 : "NAME", "AVERAGE"
410 INPUT # F1 : S,G
420 PRINT # F2 :
430 FOR I = 1 TO S
440 LET A = 0
450 INPUT # F1 : N$
460 FOR J = 1 TO G
470 INPUT # F1 : X
480 LET A = A+X
490 NEXT J
500 LET A = A/G
510 PRINT # F2 : N$,A
520 NEXT I
530 END
```

The following is a sample run of AVERAGE3 using the option to input the data at run time. This listing shows clearly the correspondence between the simple INPUT statement and the INPUT# statement.

```
AVERAGE 3

ARE DATA IN A FILE - ANSWER NO OR GIVE FILE NAME? NO
SHOULD OUTPUT GO TO A FILE - ANSWER NO OR GIVE
FILE NAME? AVERAGES
? 3,4
? GERALD FRIEND
? 78
? 86
? 61
? 90
? PHILIP CLOUGH
? 66
? 87
? 88
? 91
? ADA SHAW
? 56
? 77
? 81
? 85

DONE
```

5.2.4 Other Input/Output Statements

The LINPUT# statement is used to read strings which might contain such special characters as quotation marks, leading blanks, ampersands, and commas from terminal-format files. The format of this statement is

```
100 LINPUT #N: <list of string variables>
```

where N is a file number. Rules governing the use of the LINPUT statement (Section 4.6.1) apply to the LINPUT# statement. If N is equal to zero, the terminal itself is referenced, and input from the terminal is requested.

There are also three MAT statements which may be used with terminal-format files: MAT PRINT#, MAT INPUT#, and MAT LINPUT#. These statements are discussed in Section 6.5.3.

5.2.5 Margins on Terminal-format Files

MARGIN #N: M sets a margin of M on file #N just as the simple MARGIN statement sets a margin on lines output to the terminal. The margin for terminal-format files may be changed at any time. MARGIN #0: M has the same effect as MARGIN M. The interpretation of the margin setting is the same as in the simple MARGIN statement. See Section 4.5.6 for details.

5.3 Random Access Files

The major use of random access files is to hold big amounts of data which should be accessed in a random manner. The data will normally be loaded from a Terminal-format file using a BASIC program or be generated by a program.

Random files are used to hold numbers and strings. The data are manipulated internally in BASIC and accordingly the internal format is used. Numbers are represented in the standard floating point format and strings are saved in ASCII code two characters to a word.

The addressing mode of arrays is used to address the individual items in a random access file and when an array is assigned to a random access file, the associated indexed variable may be used the same way as for core arrays. The MAT statements may not be used with random files.

5.3.1 Using a Random Access File

Before use the file must be associated with a file designator. This is done using the OPEN # statement. A random file will be used bo. for input and for output and the OPEN # statement must be used without INPUT and OUTPUT

```
10 OPEN # 3 : FOR "AVERAGES"
```

will assign AVERAGES to file 3 and allow random access in the file.

To assign the addressing mechanism an array must be associated with a random file.

```
20 DIM # 3 : A(100, 100), A$(100, 100), B$(1000) = 40
```

The statement above indicates that AVERAGES contains 10201 numbers addressed by:

A(0,0), A(1,0), A(2,0).., A(100,0), A(0,1),... etc.

Then follows 10201 strings. The maximum number of characters in each is 16, because no size is given. The strings are addressed by:

A$(0,0), A$(1,0)...., A$(100,0), A$(0,1),..... etc.

Thereafter follows 1001 strings with a maximum of 40 characters in each, addressed by:

B$(0), B$(1), B$(2) .... etc.

Note that the same DIM statement must be used to describe the addressing within one file.

If it is desirable to access the file from other program systems the relative addresses can be computed based on the following facts:

1) The first element starts in address 0.
2) A number uses 3 addresses.
3) A string uses INT ((max. size +1) /2) addresses.

5.4 The OPEN# and CLOSE# Statements

The OPEN# statement is used both to associate a BASIC file designator with a file in the file system and to describe how the file should be used. Such a description is valid until the CLOSE# statement is used or the file is closed by the system. With the END statement and with break,all files are closed. With any error messages all files are closed.

6 ARRAY MANIPULATIONS

Up to this point in the manual a singly subscripted variable (a variable having only one subscript) has denoted a list and a doubly subscripted variable (a variable having two subscripts) has denoted a table. In this chapter it is appropriate to refer to lists as vectors and tables as matrices since we are describing them in a mathematical context.

Vectors and matrices are both arrays. That is, an array is denoted by a variable having one or more subscripts; a vector is an array having one subscript; a matrix is an array having two subscripts.

A string array is an array whose entries are strings.

BASIC provedes MAT statements which are designed to allow the programmer to work with arrays in a simple and straightforward manner. Although arrays have a row number 0 and a column number 0 in BASIC (Section 2.4), the MAT statements generally ignore them.

6.1 Initialization Statements

There are three MAT statements which facilitate the procedure of assigning values to individual array entries.

```
100 MAT A = ZER
```

This statement assigns a value of zero to each entry of the array A.

```
110 MAT A = CON
```

This statement assigns a value of one to each entry of the array A.

```
120 MAT A = IDN
```

This statement sets the matrix A equal to the identity matrix. For this statement to be valid A must be a square matrix: A must be doubly subscripted and have its number of rows equal to its number of columns. A may not be a vector.

All three of these MAT statements do not affect row 0 or column 0 of the arrays on which they operate.

6.2 Changing Dimensions using MAT Statements

As described in Section 2.4 the DIM statement is used to dimension (i.e. to reserve space in the computer for) subscripted variables. Space for entries in row 0 and column 0 of an array is a part of the total space reserved. For example the statement

```
100 DIM A(7), B(11,5)
```

results in 8 spaces being reserved for A with room for entries 0 through 7. (11 + 1) * (5 + 1) = 72 spaces are reserved for B with room for entries in rows 0 through 11 and columns 0 through 5. If subscripted variables are used in a program but do not appear in a DIM statement, BASIC implicitly saves 11 spaces for a vector and 121 spaces for a matrix (a maximum of 10 for each subscript).

It is possible to change the dimensions of the arrays used in some MAT statements by specifying the desired dimensions in the statement themselves. The initialization statements allow this flexibility. The statements

```
100 DIM A(8)
110 MAT A = ZER(5)
```

will reserve nine spaces for the vector A in line 100 and A will be redimensioned (that is, the space reserved for A in the computer will change) to a vector having 6 entries (entries 0 through 5) in line 110 with A(1) through A(5) set equal to zero. A reference to A(6) after line 110 will cause an error message to be output and the program run will terminate.

When redimensioning variables in the MAT statements care must be taken that the spaces required to satisfy the specified new dimensions do not exceed the spaces reserved for the variable in a DIM statement or reserved implicitly by BASIC.

In the previous example if we retype line 100 to read

```
100 DIM A(4)
```

an error message would be output when line 110 is reached. Line 100 reserves only five spaces for the vector A and line 110 requires six spaces for A.

Matrices may also be redimensioned in the MAT...CON statement.

```
100 DIM M(8,2)
110 MAT M = CON(5,3)
```

Twenty-seven spaces are stored for M in line 100 and line 110 requires 6 * 4 = 24 spaces for the redimensioning of M. Again, the spaces required for redimensioning may not exceed the spaces reserved.

Matrices may be redimensioned by using the MAT...IDN statement if enough space has been reserved for the redimensioned matrix. The desired number of rows and columns is included in parentheses as in the preceding examples.

```
100 DIM A(6,5)
110  MAT A = IDN (4,4)
120  END
```

Here the matrix A is dimensioned to be 6 by 5 and in line 110 it is set equal to the 4 by 4 identity matrix.

A vector may not be redimensioned to a matrix or vice versa.

As with subscripts, dimensions designated in MAT statements do not have to be integers: any arithmetic expression may be used, and if the value of the expression is not a whole number, its integer part is used.

Redimensioning of arrays may occur in other MAT statements. This feature will be noted as the remaining MAT statements are discussed.

6.3 Arithmetic Operations

```
110  MAT C = A + B
120  MAT C = A - B
```

The first statement causes the array C to be the sum of the two arrays A and B. In the second statement C is the result of subtracting array B from array A. A and B may be vectors or matrices as long as they both have the same dimensions. The array C assumes the dimensions of A and B provided enough space has been reserved for C in a DIM statement or implicitly by BASIC.

```
100  MAT A = B
```

This statement sets each entry of the array A equal to the corresponding entry of the array B. A is redimensioned to be the same size as B provided enough space has been reserved for A.

```
130  MAT C = A * B
```

This statement causes C to be set equal to the product of matrix A and matrix B provided enough space has been saved for C. The number of columns of matrix A must be equal to the number of rows of matrix B. C must be dimensioned to be a doubly subscripted variable. The product matrix C will have the same number of rows as matrix A and the same number of columns as matrix B. Thus, if A is an M by N matrix and B is an N by P matrix then C will be an M by P matrix.

While the statements

```
100  MAT A = A + B
110  MAT A = A - B
```

are allowed, the statement

```
120  MAT A = A * B
```

will result in an error message and the termination of the program. When adding or subtracting two arrays, any entry of the array is only used once so that the answer may be stored immediately in the array. If entries of the matrix being operated on during a multiplication are replaced, components needed to complete the matrix multiplication are destroyed.

The following matrix multiplication is valid, provided A is a square matrix.

```
100  MAT C = A * A
```

Performing more than one arithmetic operation in a single MAT statement is illegal. Thus, to evaluate the expression A + B - C two MAT statements are required. One way of evaluating the expression follows. We assume all dimensions are correct.

```
100  MAT D = A + B
110  MAT E = D - C
```

In general these MAT statements ignore row 0 and column 0 of the arrays on which they operate.

6.4 Functions

The transpose of a matrix may be found using the following statement.

```
100  MAT C = TRN (A)
```

This statement sets matrix C equal to the transposed version of A if enough space has been reserved for C. If A has N rows and P columns, C will be redimensioned to have P rows and N columns. The statement

```
110  MAT A = TRN (A)
```

is legal.

The statement

```
100  MAT C = (K) * A
```

causes each entry of array A to be multiplied by the value of K to form the corresponding entry of the array C. Enough space must have been stored for C, and C is redimensioned to be the same size as A. K may be any constant, variable name or arithmetic expression and must be enclosed in parentheses. The statement

```
100  MAT A = (K) * A
```

is legal.

The statement

```
100  MAT C = INV (A)
```

sets matrix C equal to the inverse of matrix A provided enough space has been saved for C. A must be a square matrix, and C is redimensioned to be the same size as A.

The function DET is available after an inversion is performed, and it is the value of the determinant of the matrix whose inverse was computed. It is important to point out that even though a matrix whose determinant is zero has no inverse, trying to compute the inverse of such a matrix in the above MAT statement will not cause the program run to stop or cause the output of any kind of error message. In this case DET is set equal to zero and the resulting "inverse" matrix is obviously not correct. It is up to the user to check the value of DET to determine whether or not the matrix has an inverse.

Since DET is not available until after the inverse is found, if the value of the determinant of a matrix is desired the inverse of the matrix must be computed first.

The following statement is legal.

```
100  MAT A = INV (A)
```

All three of these functions may change the values stored in row 0 and column 0 of the arrays involved. When inversion takes place, row 0 and column 0 of the inverse matrix are used to store intermediate calculations.

6.5 Input and Output Operations

6.5.1 The MAT READ and MAT PRINT Statements

There are MAT statements that cause entire arrays to be input or output. The program MATRIX

```
100 DIM M(3,5)
110 MAT READ M
120 DATA 1,2,3,4,5,6,7,8,9
130 DATA 10,11,12,13,14,15
140 END
```

will cause fifteen numbers to be read into the matrix M by rows. That is, the first row of M is read in, then the second and finally the third. Row 0 and column 0 are not affected. If the following line is added to MATRIX

```
135 MAT PRINT M
```

and line 110 is retyped as

```
110 MAT READ M(2,6)
```

the program will yield the following output when it is run:

```
MATRIX
1             2             3             4             5
6
7             8             9             10            11
12
DONE
```

M is redimensioned in line 110 to be a two by six matrix. These dimensions do not exceed the total spaces reserved for M in line 100 (see Section 6.2). Twelve numbers are read into M. Line 135 causes M to be printed in matrix format: the entries of each row are spaced five to a line and each row begins on a new line. Row 0 and column 0 are not printed, and a blank line is output before the first row of the matrix is printed.

If line 135 of MATRIX is changed to read

```
135 MAT PRINT M;
```

the following output is produced when MATRIX is run.

```
MATRIX
1      2      3      4      5      6
7      8      9      10     11     12
DONE
```

The semicolon after the matrix name causes M to be printed with the entries of each row closely packed on a line.

The MAT READ and MAT PRINT statements may be used with vectors as well as with matrices. The format of the statements is that described for matrices. The program VECTOR

```
100  DIM V(3)
105  MAT V = CON
110  MAT PRINT V
120  END
```

will cause V to be printed as a column of numbers:

```
VECTOR
 1
 1
 1

DONE
```

If line 110 of VECTOR is changed to read

```
110  MAT PRINT V,
```

the entries of vector V are spaces five numbers to a line in row format as follows.

```
VECTOR
 1            1            1

DONE
```

If a semicolon replaces the comma in the new line 110, V is printed in row format with the entries of V closely packed.

More than one array name may appear in a single MAT READ or MAT PRINT statement. In the MAT READ statement commas and semicolons are used both to delimit the names and to control the format in which the arrays are printed. For example, in the statement

```
100  MAT PRINT V, M;
```

If V is a vector and M is a matrix, the entries of V are printed in rows with five entries per row. M is printed as a matrix with the entries of each row closely packed.

Only array names may follow the word PRINT in a MAT PRINT statement. The following statements are illegal:

```
100  MAT PRINT M(2,3)
110  MAT PRINT TRN(A)
```

6.5.2 The MAT INPUT and MAT LINPUT Statements

A variable amount of input may be entered into a vector in response to a MAT INPUT statement; this capability is explained in Section 4.6.2. If an attempt is made to enter more input than the vector can hold, the excess data are ignored, and the program continues running. The function NUM is available after the execution of a MAT INPUT statement and it returns the number of data which were input. The vector is automatically redimensioned to the value of NUM.

A variable number of data can only be input into a vector; if a matrix is used in a MAT INPUT statement, the matrix name must be followed by the desired dimensions of the matrix in parentheses. Enough data must be input to entirely fill the matrix as the dimensions are given in the MAT INPUT statement. These dimensions may not exceed the total spaces reserved for the matrix as explained in Section 6.2. The following statements

```
100 DIM M(2,12)
110 MAT INPUT M(3,8)
```

will call for the input of 24 numbers. After statement 110 has been executed M is redimensioned to have rows 0 through 3 and columns 0 through 8.

The function NUM is available after a matrix has been input in a MAT INPUT statement. The value of NUM is the product of the dimensions specified for the matrix in the MAT INPUT statement.

A vector may also be explicitly redimensioned in the MAT INPUT statement. The desired dimension follows the vector variable name and is enclosed in parentheses. However, this format does not allow for the input of a variable amount of data. An amount of input data equal to the specified new dimension is required. The statements

```
100 DIM V(8)
110 MAT INPUT V(5)
```

will call for the input of five numbers. V will be redimensioned to have entries 0 through 5.

String vectors and matrices may also be used in the MAT LINPUT statement.

The LINPUT statement is described in Section 4.6.1; the MAT LINPUT statement allows more than one line of information (possibly containing commas, leading blanks, etc.) to be input in response to a single statement. The statements

```
100 DIM A$(6)
110 MAT LINPUT A$
```

call for the input of six strings. A variable amount of input is not allowed. String matrices may appear in a MAT LINPUT statement as well as string vectors and redimensioning may occur in the MAT LINPUT statement by placing the desired dimensions in parentheses following the variable name, as discussed for the MAT INPUT statement.

As with the other MAT statements, MAT INPUT and MAT LINPUT ignore row 0 and column 0.

More than one array name may be listed in a MAT INPUT statement if the names are separated by commas. The effect of listing more than one array name in a MAT INPUT statement is the same as listing each name in a separate MAT INPUT statement: a '?' is issued at the beginning of a new line for each variable.

As described earlier, a variable amount of input data is allowed only with vectors with no dimension in parentheses following the vector name. The value returned for the function NUM is the amount of data input into the last array listed.

```
100 DIM V(5), A(3), M(3,4)
110 MAT INPUT V, A(2), M(2,3)
120 PRINT "NUM ="; NUM
130 END
```

A sample run of this program is

```
? 1,2 &
? 3
? 1,2
? 1,2,3,4,5,6
NUM = 6
DONE
```

A variable amount of data may be entered for V (see Section 4.6.2 for the use of the & with the MAT INPUT statement), two data must be entered for A and six for M. The number 6 is printed for the value of NUM.

More than one array name may be listed in the MAT LINPUT statement; the array names are separated by commas, and a new '?' appears for each variable in the list.

6.5.3 MAT Statements and Files

The MAT PRINT#, MAT INPUT# and MAT LINPUT# statements may be used to write into and read from Teletype-format files. The formats of these statements are

```
100 MAT PRINT # N : <LIST OF ARRAYS>
110 MAT INPUT # N: <LIST OF ARRAYS>
120 MAT LINPUT # N : <LIST OF STRING ARRAYS>
```

where N is the file number of the file being read or written. Rules governing the use of the MAT PRINT# statement are the combination of rules applying to the PRINT# statement and the MAT PRINT statement. Analogous statements can be made for the MAT INPUT# and MAT LINPUT# statements.

For a complete discussion of files see Chapter 5.

6.6 Examples using MAT Statements

The following two examples illustrate some of the MAT statements discussed in this chapter.

6.6.1 Example One

```
100 READ N, P
110 MAT READ A(N, N)
120 MAT B = CON(N, N)
130 MAT C = A + B
140 PRINT "SUM OF A AND MATRIX OF 1'S IS"
150 MAT PRINT C
160 PRINT
170 PRINT "INPUT"; N*P; "VALUES FOR MATRIX B";
180 MAT INPUT B(N, P)
190 MAT C = A * B
200 PRINT
210 PRINT "PRODUCT OF A AND B IS"
220 MAT PRINT C;
230 MAT D = TRN (C)
240 PRINT
250 PRINT "TRANSPOSE OF THIS PRODUCT IS"
260 MAT PRINT D
270 DATA 2,3
280 DATA 1,2,3,4
290 END
```

Since the matrices used in this example do not appear in a DIM statement, BASIC implicitly dimensions them to be ten by ten and reserves 121 spaces for each matrix. Line 110 dimensions A to be 2 by 2, while it reads values for the entries of A from the DATA statement in line 280. Line 120 dimensions B to be 2 by 2 and sets all entries of B equal to 1. Line 130 adds A and B and stores the result in C. C is redimensioned to be a 2 by 2 matrix as is shown when it is printed in line 150. Line 180

requests the user to input enough values to fill a 2 by 3 matrix and B takes on these new dimensions. Line 190 sets C equal to the product of A and B and C is redimensioned to be 2 by 3. C is printed in closely packed format in line 220. Matrix D becomes the transpose of C in line 230 and D is redimensioned to 3 by 2. D is printed in regular format in line 260.

A run of this example follows:

```
EXAMPLE1
SUM OF A AND MATRIX OF 1'S IS
 2             3
 4             5
INPUT 6 VALUES FOR MATRIX B? 2,-1,7,18,6,-10

PRODUCT OF A AND B IS
38    11    -13
78    21    -19

TRANSPOSE OF THIS PRODUCT IS
 38            78
 11            21
-13           -19
DONE
```

6.6.2 Example Two

The second example inverts an N by N Hilbert matrix which has the form

```
1      1/2      1/3      ...   1/N
1/2    1/3      1/4      ...   1/(N+1)
.      .        .        ...
.      .        .        ...
.      .        .        ...
1/N    1/(N+1)  1/(N+2)  ...   1/(2N-1)
```

A listing of the program follows:

```
100  REM  THIS PROGRAM INVERTS AN N BY N HILBERT MATRIX
110  DIM  A(20,20), I(20,20), B(20,20)
120  DIM  C(20,20), D(20,20)
130  READ N
140  MAT A = CON(N,N)
150  FOR I = 1 TO N
160  FOR J = 1 TO N
170  LET A(I,J) = 1/(I+J-1)
180  NEXT J
190  NEXT I
200  MAT B = INV(A)
210  PRINT "INV(A) ="
220  MAT PRINT B,
230  PRINT
240  PRINT "DETERMINANT OF A = ";DET
260  MAT I = IDN (N,N)
270  MAT C = A * B
280  MAT D = I-C
290  FOR I = 1 TO N
300  FOR J = 1 TO N
310  IF X >= ABS(D(I,J)) THEN 330
320  LET X = ABS (D(I,J))
330  NEXT J
340  NEXT I
350  PRINT
360  PRINT "LARGEST ABSOLUTE DIFFERENCE =" ; X
370  DATA 4
380  END
```

The double loop in lines 150 - 190 sets up the Hilbert matrix A after the correct dimensions have been set up in line 140. A single instruction results in the computation of the inverse (line 200) and one more instruction prints it out in closely packed format (line 220). The value of the determinant of A is available after the inversion and is printed in line 240. I is set equal to the indentity matrix having N rows and N columns in line 260. Lines 270 through 340 find the largest absolute difference between an entry of the product matrix A * B and the corresponding entry of the identity matrix. This value is printed in line 360 and is a measure of the accuracy of the inverse since the product of a matrix and its inverse is the identity matrix.

The following run uses a value of 4 for N.

```
HILMAT
INV(A)=
  16          -120          240          -140
-120          1200         -2700          1680
 240         -2700          6480         -4200
-140          1680         -4200          2800

DETERMINANT OF A = 1.65344 E -07

LARGEST ABSOLUTE DIFFERENCE = 1.66893 E -06

DONE
```

While this example shows how several MAT statements are used, it also points out that the accuracy of the matrices generated by using MAT statements depends on the structure of the matrices and on the fact that the computer stores any number to only a limited number of significant digits. These two factors combine in this example when N is greater than or equal to 7 to cause severe roundoff errors which in turn cause a highly inaccurate inverse to be returned. When N = 7, a value for the absolute difference described previously is greater than one and continues to grow as N increases.

6.7 Simulating an N-Dimensional Array

Although arrays having more than two dimensions are not allowed in BASIC, the method outlined in the following program can be used to simulate an array having any number of dimensions. It makes use of the fact that defined functions may have any number of arguments, and a one to one correspondence is set up between the entries of the array and the entries of a vector. Formatting techniques cause the entries of the vector to be printed in a format reflecting the dimensions of the array.

This example simulates an array having three dimensions; it can easily be rewritten to accomodate four or more dimensions.

```
100 DIM  V(1000)
110 MAT  READ D(3)
120 DEF FNA(I,J,K) = ((I-1)* D(2) + (J-1)) * D(3) + K
130 FOR I = 1 TO D(1)
140 FOR J = 1 TO D(2)
150 FOR K = 1 TO D(3)
160 LET V(FNA(I,J,K)) = I + 2 * J + K↑ 2
170 PRINT V(FNA(I,J,K)),
180 NEXT K
190 PRINT
200 NEXT J
210 PRINT
230 NEXT I
240 DATA 2,3,4
250 END
```

When the program is run, the vector is printed as two 3 by 4 matrices.

```
3-ARRAY

 4          7          12         19
 6          9          14         21
 8          11         16         23

 5          8          13         20
 7          10         15         22
 9          12         17         24

DONE
```

6.8 The Row Zero and Column Zero

As mentioned earlier, BASIC reserves room for an entry numbered zero in vectors and for entries in row zero and column zero of matrices. In general the MAT statements ignore these components; they do not enter into computations except to store intermediate calculations and are not normally printed in MAT PRINT statements.

However, one condition under which the MAT statements do employ the zero component of arrays is when at least one of the arrays involved in the MAT operation is dimensioned to have only a row and/or column zero. For example, the statements

```
100 DIM A(0,5), B(0,5)
110 MAT C = A + B
120 MAT PRINT C
```

cause C to be the sum of A and B while C is redimensioned to 0 by 5. Usually the MAT PRINT statement does not print row 0 of a matrix; however, C is printed in regular matrix format in line 120.

For some operations with matrices, vectors are treated like matrices having columns numbered zero. For example

```
100 DIM V(5), M(0,5), N(5,0), P(5,5)
110 MAT C = V * M
120 MAT D = P * V
130 MAT M = TRN(V)
```

V is treated in the MAT statements as if it were dimensioned 5 by 0, making the three MAT statements legal. Product matrix C is redimensioned 5 by 5 in line 110 and D is redimensioned 5 by 0 in line 120.

These examples are not to be interpreted as illustrating general rules about the way row 0 and column 0 of arrays are used in MAT statements. They should be thought of as showing what happens in a particular case and suggesting what might take place in another instance.

It is best to avoid using entry 0 in vectors and row 0 and column 0 of matrices in conjunction with MAT statements. In many cases these components are not operated on by the MAT statements as one might expect. Also, the manner in which BASIC handles the entry 0 of vectors and row 0 and column 0 of matrices may be changed.

7 MISCELLANEOUS INFORMATION

7.1 Roundoff Errors

The smallest number BASIC can handle is approximately $1 \cdot 10 \uparrow -4920$ and the largest number is $1 \cdot 10 \uparrow +4920$, but input and output are restricted to be within the following limits: $1 \cdot 10 \uparrow -100 < |x| < 1 \cdot 10 \uparrow 100$.

BASIC stores numbers correct to approximately nine significant digits and prints generally numbers to six significant digits.

As mentioned in Section 2.3, the values of the expressions in the FOR statement need not be integers. However, the user must be cautioned that using a non-integer step size may result in Roundoff errors. These errors occur because the computer can only store about nine significant digits for each number it computes. The cumulative effect of these Roundoff errors over a loop executed many times may be significant: the expected value of the running variable may differ from the actual value.

```
100  FOR X = 0 TO 200 STEP 0.001
110  LET Y = Y + 1
120  REM Y COUNTS THE NUMBER OF TIMES
130  REM THE LOOP IS EXECUTED
140  NEXT X
150  PRINT X,Y
160  END
```

This program gave the following output when it was run:

```
100  FOR X = 0 TO 200 STEP 0.001
RUN
  199.999          199998

DONE
```

For the same program, X had a value of a little more than 199.999 on the last pass through the loop; it was not equal to 200 as we might expect. With the next addition of .001, however, the value of X would be greater than 200, so the conditions for ending the loop are satisfied. Also note that Y, which counts the number of times the loop is performed, is not 200001, the expected value, but 199998; the loop has been executed three times less than might be expected. Consequently, calculations involving the running variable or depending on the number of times the loop was performed would be in error because of Roundoff errors.

Thus, in general, use integer step sizes and integer FROM and TO elements to avoid Roundoff errors. If you want to step over a series of non-integer values, appropriate operations may be performed on the running variable within the loop to achieve this result. For instance, in the example above X may be made to range from 1 to 200 in steps of .001 using the following technique:

```
100  FOR I = 0 TO 200000
110  LET X = I/1000
120  LET Y = Y + 1
130  NEXT I
140  PRINT X,Y
150  END
160  END
```

This program prints a value of 200 for X and 200001 for Y. These values are the expected ones, and no Roundoff error has occurred.

7.2 Some Specifications and Limits

Largest line number	: 32767
Upper limit for numbers input/output	: 1.0E+100
Lower limit for numbers input/output	: 1.0E-100
Accuracy	: Approximately 9 significant digits
Maximum nesting of FOR NEXT loops	: 10 deep
Maximum nesting of GOSUB	: 12 deep

Size of statements is restricted to one line (80 characters).

7.3 Entering the BASIC System

7.3.1 Using NORD TSS

When using the NORD Time Sharing System you press the Teletype pushbutton ESC. The system log-in procedure is initiated as follows:

```
NORD TSS
USER NAME      KRISTIANSEN
PASSWORD
```

Only users identified to the system will be admitted and you must know the password, which is some sequence of characters terminated by carriage return. When the character @ is printed, the TSS command processor is active. You may then just print:

```
BASIC
```

and the BASIC system will be entered printing:

```
BASIC ON LINE
NEW OR OLD -----
```

7.3.2 Using the BASIC Time Sharing System

A NORD-1 without mass memory which is furnished with BASIC Time Sharing System may service BASIC users. The system is entered into the computer as follows:

a) Get the paper tape marked BASIC Multi User.

b) Put the tape in the tape reader and turn on the reader.

c) Press the pushbuttons marked MASTER CLEAR and LOAD on the NORD-1 operators panel.

d) The tape should now be read and entered into core. The input is checked, and if it is considered correct, the system is started by printing:

```
BASIC ON LINE
NEW OR OLD ---
```

on the console Teletype (provided this Teletype is turned on). All Teletypes in the system may now be activated. This is done by pressing the Teletype pushbutton ESC. The Teletype will respond by typing:

```
BASIC ON LINE
NEW OR OLD ---
```

7.3.3 NORD BASIC One User System

In this system all the monitoring routines are omitted to enable bigger user programs. Accordingly only one user is admitted and breaks are executed manually. The system is loaded as follows:

a) Get the tape marked BASIC ONE USER SYSTEM.

b) Put the tape in the tape reader and turn on the reader.

c) Press the pushbuttons marked MASTER CLEAR and LOAD on the NORD-1 operators panel.

d) The tape should now be read and entered into core. The input is checked, and if it is considered correct, the system is started by printing

```
BASIC ON LINE
NEW OR OLD ---
```

on the Teletype (provided the Teletype is turned on).

A break is executed as follows:

Press the pushbutton STOP on the NORD-1 operators panel. Set the OPR register to the restart address printed on the tape. Press the push-buttons SET ADR and CONT on the NORD-1 operators panel.

7.4 BASIC Error Messages

The messages you may encounter when writing or running a BASIC program are listed in this chapter. These error messages may originate in different parts of the system. In systems without mass memory, error messages are normally given as error codes.

Compiling

When you are writing statements the compiler will check for syntax errors. These error messages are denoted with error codes CE followed by an error number to be looked up in Section 7.4.1. The system will now check the input character. If this is a question mark, the erroneous line will be printed with the errors marked.

Run Time

When executing programs the BASIC system may print error messages like: RE 3 IN LINE s. The error number may be looked up in Section 7.4.2.

The arithmetic functions SIN, COS, LOG, etc. have a special error format documented in Section 7.4.3.

Error Messages in TSS

BASIC systems with mass memory use the general NORD file system. The error messages you may obtain from that system are documented in Section 7.4.4.

7.4.1 Compiler Error Messages

CE1	Illegal character in this context.
CE2	Minor arithmetic error, probably operator missing.
CE3	Something wrong with parentheses.
CE4	No line number or illegal line number.
CE7	Expected variable not found.
CE8	Expected number not found
CE9	Illegal word in this context.
CE10	Only string variables legal in this context.
CE11	= used illegally or omitted.
CE12	" omitted or used illegally.
CE13	Illegal string format.
CE14	Mixed mode string - arithmetic.
CE15	No relational operator found in IF statement.
CE16	Word (GOTO, GOSUB, TO, THEN) expected, not found.
CE17	No value for variable with FOR, LET.
CE18	Illegal FOR looping variable.
CE19	Illegal use of CHR$ or SEG$.
CE20	Array must have an index in this context.
CE21	Array index not legal in this context.
CE22	No end of array subscript found.
CE23	Illegal array index format.
CE26	Program name too long.
CE30	Illegal use of files.
CE31	Illegal file terminator.
CE32	FN-function used recursively.
CE33	Illegal format for FN-function.
CE39	CON legal only in execution code.
CE40	Array previously defined two-dimensional.
CE41	Array previously defined one-dimensional.
CE42	Only numeric arrays legal in this context.
CE43	Only two-dimensional arrays legal in this context.
CE44	Array dimensions not matching.
CE45	Illegal operator in MAT statement.
CE46	MAT multiply does not allow same array on both sides of assignment operator.
CE55	Print Using must start with string.

7.4.2 Run Time Error Messages

RE1	Out of numeric data for READ statement.
RE2	Out of string data for READ statement.
RE3	END not last statement.
RE5	Division by zero tried, overflow.
RE6	Undefined string variable used.
RE8	GOSUB nested too deeply, table full.
RE9	No return address with RETURN statement.
RE10	FOR - NEXT nested too deeply.
RE11	More NEXT than FOR.
RE12	More FOR than NEXT.
RE13	FOR-NEXT illegally nested, variables do not match.
RE14	FOR-NEXT loop illegally entered.
RE15	Increment 0 with FOR statement.
RE16	DEF FN and FNEND illegally nested.
RE17	Number of input data incorrect.
RE18	Input Data terminated illegally.
RE20	Negative index illegal.
RE21	Index too big, overflow.
RE22	Array dimensions not matching.
RE23	Only square matrices with MAT IDN and MAT INV.
RE25	Illegal file number.
RE26	Illegal file name, file used illegally.
RE27	Program tried to read EOF.
RE28	ETB (end-of-file mark) read.
RE29	Tried to read/write sequentially in random file.
RE30	System out of core.
RE31	Too many core tables used, table full, use the command TABLE.

RE32	Input buffer overflow.
RE33	Core table full, probably too may errors in the program.
RE34	Illegal FN-name
RE35	Statement reached illegally, only legal within multiple line DEF FN.
RE36	Illegal number of arguments in FN-function.
RE37	Undefined FN-function called.
RE39	Command CON used illegally.
RE40	Zero or negative margin illegal.
RE41	No margin with random file.
RE44	Illegal string size.
RE46	String too long.
RE47	MAT is used with disc arrays.
RE60	Print Using not allowed with disc strings.
RE61	Print Using format error.

7.4.3 Mathematical Library Error Messages

The library routines are documented independently in the manual: Re-entrant FORTRAN Mathematical Library.

Error messages:	Function:	Error condition:
RUN ERR AA	↑	For A ↑ B if A = 0 and B < 0 Result: A ↑ B = 0. For A ↑ B if B $\log_2$ A ≥ 2 ↑ 14 Result: A ↑ B = 1 ·E + 99.
EXPONENT ROUNDED	↑	For A ↑ B if A < 0 and B not integer. Result: A ↑ B = 1/A ↑ ABS(INT(B)).
RUN ERR AI	↑	For A ↑ B if B $\log_2$ A ≥ 2 ↑ 14 Result: A ↑ B = 1 ·E + 99.
RUN ERR CO	COS	If argument ≥ 2^{16} radians.
RUN ERR SI	SIN	Result set equal to 0.
RUN ERR EX	EXP	For EXP(x) if x/ln2 ≥ 2^{16} Result set equal to 1 ·E99.
RUN ERR LN	LOG	Argument (x) less or equal to zero. Result is set equal to 1.E99.
RUN ERR SQ	SQR	Argument < 0. Result is set to 0.

7.4.4 Error Messages from NORD TSS

For further information consult the manual: The NORD Time Sharing System.

NO MORE TRACKS AVAILABLE
NO SUCH USER
USER INDEX BLOCK FULL
FILE ALREADY EXISTS
NO SUCH FILE
OBJECT TABLE FILLED
OPEN FILE TABLE FILLED
INSUFFICIENT ACCESS
ALREADY OPEN FOR WRITE
BAD FILE NUMBER
NO SUCH TRACK
BAD TRACK NUMBER
TRACK ALREADY EXISTS
NO SUCH PAGE
FATAL ERROR
TRANSFER ERROR
BAD DEVICE TYPE
DISK AREA ALREADY IN USE
WRITE NOT PERMITTED
NOT A SEQUENTIAL FILE
FILE MUST BE CLOSED BY EVERYONE
AMBIGUOUS FILE NAME
BAD CHARACTER

7.4.5 Other Messages printed by the System

DEV RESERVED BY TTY n	- see the RESERVE command.
UNKNOWN LINE NO s1 REQUESTED IN LINE s2	- self-explanatory.
SYSTEMS ERROR	- an error in the BASIC compiler.
?	- the system requests input.
STOP IN LINE s	- STOP statement executed in line s.
BREAK	- a break has been processed.
DONE	- END statement executed.
WAIT FOR READY --	- A transfer is started.
PROGRAM INTERRUPTED IN EXECUTION OF DEFFN. SAVE AND RE-COMPILE BEFORE FURTHER EXECUTION	- Caused by STOP, Run Error or Escape.

7.5 ASCII Character Set

Graphic	Octal Value	Decimal Value	ASC Abbreviation	Comments
	0	0	NUL	Null
	1	1	SOH	Start of heading
	2	2	STX	Start of text
	3	3	ETX	End of text
	4	4	EOT	End of transmission
	5	5	ENQ	Enquiry
	6	6	ACK	Acknowledge
	7	7	BEL	Bell
	10	8	BS	Backspace
	11	9	HT	Horizontal tabulation
	12	10	LF	Line feed
	13	11	VT	Vertical tabulation
	14	12	FF	Form feed
	15	13	CR	Carriage return
	16	14	SO	Shift out
	17	15	SI	Shift in
	20	16	DLE	Data link escape
	21	17	DC1	Device control 1
	22	18	DC2	Device control 2
	23	19	DC3	Device control 3
	24	20	DC4	Device control 4
	25	21	NAK	Negative acknowledge
	26	22	SYN	Synchronous idle
	27	23	ETB	End of transmission block
	30	24	CAN	Cancel
	31	25	EM	End of medium
	32	26	SUB	Substitute
	33	27	ESC	Escape
	34	28	FS	File separator
	35	29	GS	Group separator
	36	30	RS	Record separator
	37	31	US	Unit separator
	40	32	SP	Space
!	41	33	!	Exclamation point
"	42	34	"	Quotation mark
#	43	35	#	Number sign
$	44	36	$	Dollar sign
%	45	37	%	Percent sign
&	46	38	&	Ampersand
'	47	39	'	Apostrophe
(	50	40	(	Opening parenthesis
)	51	41	)	Closing parenthesis
*	52	42	*	Asterisk
+	53	43	+	Plus
,	54	44	,	Comma
-	55	45	-	Hyphen (Minus)
.	56	46	.	Period (Decimal)

Graphic	Octal Value	Decimal Value	Abbreviation	Comments
/	57	47	/	Slant
0	60	48	0	Zero
1	61	49	1	One
2	62	50	2	Two
3	63	51	3	Three
4	64	52	4	Four
5	65	53	5	Five
6	66	54	6	Six
7	67	55	7	Seven
8	70	56	8	Eight
9	71	57	9	Nine
:	72	58	:	Colon
;	73	59	;	Semi-colon
<	74	60	<	Less than
=	75	61	=	Equals
>	76	62	>	Greater than
?	77	63	?	Question mark
@	100	64	@	Commercial at
A	101	65	A	Uppercase A
B	102	66	B	Uppercase B
C	103	67	C	Uppercase C
D	104	68	D	Uppercase D
E	105	69	E	Uppercase E
F	106	70	F	Uppercase F
G	107	71	G	Uppercase G
H	110	72	H	Uppercase H
I	111	73	I	Uppercase I
J	112	74	J	Uppercase J
K	113	75	K	Uppercase K
L	114	76	L	Uppercase L
M	115	77	M	Uppercase M
N	116	78	N	Uppercase N
O	117	79	O	Uppercase O
P	120	80	P	Uppercase P
Q	121	81	Q	Uppercase Q
R	122	82	R	Uppercase R
S	123	83	S	Uppercase S
T	124	84	T	Uppercase T
U	125	85	U	Uppercase U
V	126	86	V	Uppercase V
W	127	87	W	Uppercase W
X	130	88	X	Uppercase X
Y	131	89	Y	Uppercase Y
Z	132	90	Z	Uppercase Z

Graphic	Octal Value	Decimal Value	ASC Abbreviation	Comments
[	133	91	[	Opening bracket
\	134	92	\	Reverse slant
]	135	93	]	Closing bracket
^	136	94	^	Circumflex, up-arrow
_ or ←	137	95	_, UND, BKR	Underscore, back arrow
`	140	96	`, GRA	Grave accent
a	141	97	a, LCA	Lowercase a
b	142	98	b, LCB	Lowercase b
c	143	99	c, LCC	Lowercase c
d	144	100	d, LCD	Lowercase d
e	145	101	e, LCE	Lowercase e
f	146	102	f, LCF	Lowercase f
g	147	103	g, LCG	Lowercase g
h	150	104	h, LCH	Lowercase h
i	151	105	i, LCI	Lowercase i
j	152	106	j, LCJ	Lowercase j
k	153	107	k, LCK	Lowercase k
l	154	108	l, LCL	Lowercase l
m	155	109	m, LCM	Lowercase m
n	156	110	n, LCN	Lowercase n
o	157	111	o, LCO	Lowercase o
p	160	112	p, LCP	Lowercase p
q	161	113	q, LCQ	Lowercase q
r	162	114	r, LCR	Lowercase r
s	163	115	s, LCS	Lowercase s
t	164	116	t, LCT	Lowercase t
u	165	117	u, LCU	Lowercase u
v	166	118	v, LCV	Lowercase v
w	167	119	w, LCW	Lowercase w
x	170	120	x, LCX	Lowercase x
y	171	121	y, LCY	Lowercase y
z	172	122	z, LCZ	Lowercase z
{	173	123	{, LBR	Opening (left) brace
|	174	124	|, VLN	Vertical line
}	175	125	}, RBR	Closing (right) brace
~	176	126	~, TIL	Tilde
	177	127	DEL	Delete, rubout

7.6 Line Edit Commands

In TSS versions of BASIC the following control characters are available when typing a line on the terminal.

&A	Backspace one character (types ↑).
&C	Copy one character from old line.
&D	Copy rest of old line and terminate edit.
&E	Change insert/replace mode (types < or >).
&F	Copy rest of old line (without typing) and terminate edit.
&H	Copy rest of old line without terminating edit.
&I	Space to next tab stop.
&L	Terminate edit.
&M	Terminate edit (CR).
&OC	Copy old line up to but not including character C.
&PC	Skip characters in old line up to but not incluing character C (% is typed for each character skipped).
&Q	Backspace to the beginning of the new and old lines (types ← CR).
&R	Retype fast.
&S	Skip one character in old line (types %).
&T	Retype aligned.
&U	Copy up to next tab stop.
&VC	Take character C literally.
&W	Backspace one "word" (types \ ). Since &W also means "end of file", it must be preceded by &V to reach the BASIC system.
&XC	Skip characters in old line up to and including the character C (% is typed for each character skipped).
&Y	Append rest of old line to new line and edit the result.
&ZC	Copy old line up to and including character C.

The BASIC command LIST <line-number> will copy the actual line to old line. Accordingly line edit commands may be used to edit this line.

7.7 The LIB COMMAND

This command is availabe in TSS versions only.
Syntax:

LIB <Library-file name> (<Sub-file name>)

The LIB command is used to read BASIC programs from library files. The following example reads sub-file SUBR1 in the library-file LIB6.

LIB LIB6 (SUBR1)

7.8 The SIZE Command

This command prints on the terminal the size of your program and the statement number of the first and last line.

APPENDIX A

CALLING ASSEMBLY AND FORTRAN ROUTINES

A.1 Introduction

This section describes how the BASIC user may link his program to a subroutine written in MAC II assembly code or NORD FORTRAN IV.

This system gives access to ND's large subroutine libraries written in assembly or FORTRAN source language. Examples of such libraries are Scientific subroutine package (about 200 subroutines), Commercial subroutine package, Plot package, etc.

The possibility of accessing assembly subroutines enables the BASIC user to utilize all the features of the computer. He may for instance control devices such as analog multiplexers, digital input/output devices etc. from a BASIC program.

The combination of BASIC's many advantages as a high-level language and this new feature has really made NORD BASIC system a tool suitable for many applications.

It is assumed that the user is familiar with the following documentation:

MAC Users' Guide

NORD FORTRAN IV Reference Manual

Binary Relocatable Loader

MAC and FORTRAN both produce an object representation of the source code called Binary Relocatable Format (BRF). This format allows machine instructions and data to be loaded anywhere in core memory. The loading is done by a Binary Relocating Loader (BRL). A version of this loader is used in the BASIC system to allow BASIC to utilize programs in BRF format.

A.2 Description

The new commands for loading BRF subroutines are:

MLOAD < File name>

Load one subroutine. Terminated by)9END in MAC; END in FORTRAN.

ALOAD < File name >

Load several subroutines. Terminated by)9EOF in MAC; EOF in FORTRAN.

Note: File name includes standard I/O equipment.

The new BASIC statement for calling a BRF subroutine is:

CALL

The CALL statement must be followed by the subroutine name and parameters, if any.

General description:

<no.> CALL <Subr.name> (<1.param>,.....,<N.param.>)

This statement and the load command may of course be typed in any order.

A.2.1 Subroutine Name

The name may consist of one to five alphanumeric characters where the first one must be a letter. If more than five characters are used, only the leftmost five are recognized.

Note: One should not use more than five characters, because MAC uses only the rightmost five in a label.

A.2.2 Parameters

Actual parameters are specified in the calling program (CALLER) which is BASIC in this case. See example below.

Formal parameters are specified in the subroutine called (CALLEE). This specification is necessary only with FORTRAN subroutines. See example below.

The type and number of parameters must always correspond.

If the subroutine name in the CALL statement is terminated by carriage return or comment, then no actual parameters are passed to the BRF subroutine.

One may specify up to 30 parameters when calling a FORTRAN subroutine. The number of parameters to a MAC subroutine is restricted by BASIC itself. (Only one statement per line.)

The parameters must be enclosed by parentheses and separated by commas. A parameter must be written as an expression, i.e., a variable, a number, a function, or a combination of these.

The CALL statement is a call by reference. Hence the actual parameters are delivered to the CALLE as addresses from the CALLER. The CALLEE is then permitted to change the value of the actual parameters. This value is returned to BASIC if the parameter is a variable.

Following is an example which illustrates use of parameters:

Call a subroutine written in MAC or FORTRAN which multiplies two numbers and returns the result:

```
10 CALL MULT(X, Y, Z)
```

The variables X, Y, and Z are actual parameters now accessible by subroutine MULT. Assume that X is the multiplicand, Y the multiplier and Z the result of the operation.

Subroutine MULT written in FORTRAN:

```
SUBROUTINE MULT(A, B, C)
C=A*B
END
```

If an indexed variable is used as actual parameter in the BASIC CALL statement, a pointer to this element may be found using the A-register as described in Section A.3.2. Addresses of other elements may be found recognizing that subscripted variables in BASIC have a column zero and a row zero and elements are stored with entries of a column in consecutive core addressing.

There is no restriction on the names in the formal parameter list. However, it is important that the parameter types correspond to the representation in BASIC. A, B and C are all real variables in FORTRAN. Parameter types are discussed later. Multiplicand A corresponding to actual parameter X is multiplied by B corresponding to Y with the result being placed in C which corresponds to Z.

Subroutine MULT written in MAC:

```
)9BEG
)9ENT  MULT
MULT,   SWAP  SA  DB
        STA   SAVB

        LDF   I 0 ,B        % 1. PARAMETER (X)

        FMU   I 1 ,B        % 2. PARAMETER (Y)

        STF   I 2 ,B        % 3. PARAMETER (Z)

        LDA   SAVB
        COPY  SA  DB
        EXIT
SAVB.    0

)9END
)LINE
```

In the above example **0 ,B refers to** parameter X, 1 ,B to Y and 2 ,B to Z. **Note that it is the sequence of the** parameters which is **important,** and not the names.

..3 Usage

A.3.1 Calling a FORTRAN Subroutine

All variables in a BASIC program are **represented in** floating point format. In FORTRAN, however, **there are** several types of variables:

Logical
Integer
Real
Double
Complex

The formal parameter types are checked by BASIC at run time; thus only reals and integers are accepted. Before entering the subroutine, BASIC converts the actual parameters to integer, if specified. This process is reversed upon return.

Note: The value of such parameters must be within range of the integer representation (signed 15 bit number).

If the formal parameter is an integer array, no conversion takes place, but the parameter is accepted.

Subscripted variables in FORTRAN are indexed from one, whereas BASIC uses an index zero. This must be recognized especially when using two dimensional arrays.

With an array designator used in a FORTRAN subroutine, the element used as actual parameter will be identified by FORTRAN as the first FORTRAN array element.

When the last FORTRAN routine is loaded, the user must always remember to load the FORTRAN run time system with the command ALOAD.

Example with actual and formal parameters:

```
10 CALL EXECU (-5, A, SIN(X)/COS(Y), B(5), Z(1))

SUBROUTINE EXECU(I, A, SINCO, B, ARAY)
```

BASIC	FORTRAN	COMMENT
CALL	SUBROUTINE	Call and declaration
EXECU	EXECU	Subroutine name
-5	I	Converted to integer/real
A	A	Real variable
SIN(X)/COS(Y)	SINCO	Real variable
B(5)	B	Real array element
Z(1)	ARAY	Real array. Declared DIMENSION in FORTRAN. Index must be one to correspond with FORTRAN. Subroutine EXECU is now able to access (change) all the elements in the array ARAY (Z).

A.3.2 Calling a MAC Subroutine

When entering the subroutine, the A register points to a string of the actual parameter addresses (if any). The L register contains the return address. The value of the B register upon return from the subroutine must not differ from the one upon entry. The other central registers may be used freely.

An extraction of the compiled code generated from the following BASIC statement is listed below:

BASIC statement:

```
10 CALL MACR(A,B+C,"TEXT",4)
```

Extraction of object representation:

```
            000004          % Number of parameters
(A reg.) →  014560          % Address of variable A
            023412          % Address of the result B+C
            023430          % Start address of the string "TEXT"
            023507          % Address of the number 4
(L reg.) →                  % Return
```

The MAC routine must be programmed according to the facts mentioned above for successful access to the parameters.

Example:

```
)9BEG
)9ENT MACR

MACR,   SWAP SA DB
        STA  SAVB     % Save B reg.

        LDF I 0,B     % Value of A
        -
        -

        LDF I 1,B     % The sum B+C
        -
        -

        LDA I 2,B     % First two bytes
        -             % of the string
        -
        -

        LDF I 3,B     % The value 4
        -
        -
        -
        LDA  SAVB     % B reg
        COPY SA DB    % Entering value
        EXIT          % Return to BASIC
SAVB,   0
)9END
)LINE
```

As mentioned all values in BASIC are represented in floating point format (LDF - STF).

Strings are represented in the standard format: The ASCII values are packed two by two in consecutive order. (Corresponding to string assembly in MAC and A format in FORTRAN.)

A.4 New Error Messages

Compile time:

CE60 ARGUMENT SYNTAX ERROR

Run time:

RE50 SUBROUTINE REFERRED TO IN CALL STATEMENT NOT LOADED

RE51 FORTRAN PARAMETER DESCRIPTORS NOT FOUND
System error.

RE52 NOT ACCEPTABLE SUBROUTINE TYPE
Acceptable types: Subroutine, Real Function, Integer Function.

RE53 NOT CORRESPONDING NUMBER OF ACTUAL AND FORMAL PARAMETERS

RE54 NOT ACCEPTABLE FORMAL PARAMETER.
Acceptable types: Real Variable, Real Array, Integer Variable, Integer Array. (The last type is not converted.)

RE55 SUBROUTINE(S) CALLED FROM OTHER SUBROUTINE(S) NOT LOADED
This message is preceded by a list of the actual names.

The user may also receive error messages from the loader. (Documentation: Binary Relocatable Loader.) These errors will always reset the loader, i.e., everything is forgotten about previous loaded BRF routines.

Fatal FORTRAN run time errors will return to BASIC and print BREAK, preceded by the actual error message.

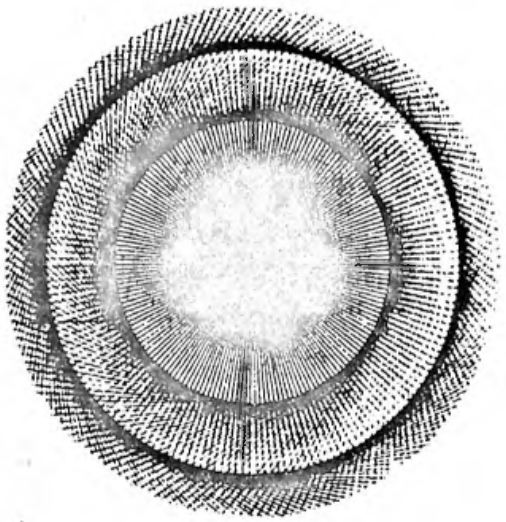
A.5 Program Examples

Example 1:

Plot (2 x 3) cm rectangles in step of 2 degrees from the same origin:

```
10 FOR    A=0 TO 360 STEP 2
20 CALL   RECT(0,0,3,2,A,3)
30 NEXT   A
40 END

ALOAD  T-R    (Load NORD BASIC PLOT PACKAGE)
ALOAD  T-R    (Load FORTRAN  RUN TIME SYSTEM)
RUN
```



The step may easily be changed by re-writing statement no. 10; for instance:

```
10 FOR A=0 TO 360 STEP 10
```

A/S NORSK DATA-ELEKTRONIKK
Lørenveien 57, Oslo 5 - Tlf. 21 73 71

COMMENT AND EVALUATION SHEET

ND-60.040.02 NORD BASIC Reference Manual

In order for this manual to develop to the point where it best suits your needs, we must have your comments, corrections, suggestions for additions, etc. Please write down your comments on this pre-addressed form and post it. Please be specific wherever possible.

FROM: _____

_____

_____

N O R D B A S I C

Reference Manual