LOGICON 2+2

BASIC MANUAL

LOGICON INC.
1075 CAMINO DEL RIO, SOUTH
SAN DIEGO, CALIFORNIA

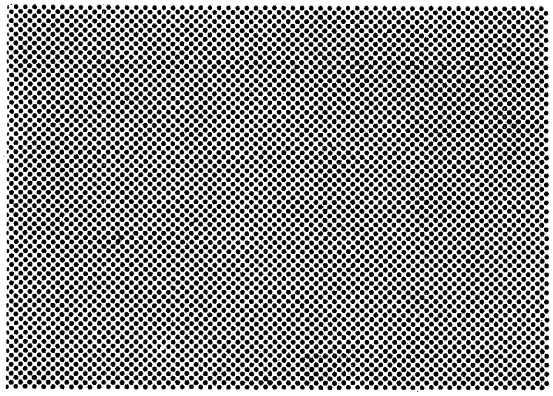
15 December 1970

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF TABLES



Preface

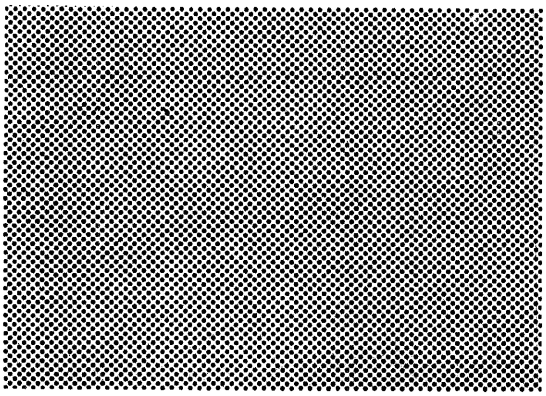
This user's manual is intended as a reference guide in the use of the BASIC[R] function of the LOGICON 2+2 System.

BASIC is a conversational, problem-solving language developed by Dartmouth College, Hanover, New Hampshire and is copyrighted by the Trustees of Dartmouth College. The development of BASIC was supported by the National Science Foundation under the terms of a grant to Dartmouth College. Under this grant, Dartmouth College developed, under the direction of Professors John G. Kemeny and Thomas E. Kurtz, the BASIC language. We would like to thank Dartmouth College for the privilege of using their BASIC Manual, 4th Edition as a reference in writing this manual.

BASIC is used to solve both simple and complex mathematical problems from your teletype console, and it is particularly suited to time-sharing. With BASIC, you type your computational procedure as a series of numbered statements, utilizing common English syntax and familiar mathematical notation. If BASIC is new to you, you need spend only an hour or so learning the elementary commands necessary for solving most business or scientific problems. With experience, you may add the advanced commands needed to perform more intricate manipulations and to express your problems more efficiently and concisely.

Once you have entered your statements via your console, simply type RUN to initiate execution of your routine and receive your results instantaneously.

[R]Registered: Trustees of Dartmouth College



I . . .

Intrduction

A program is a set of directions used to tell a computer how to solve and provide the answer(s) to some problem. In general, a program provides numerical answers to computational type problems.

Any program must meet two requirements before it can be carried out. The first is that it must be presented in a language that is understood by the computer. The computer language discussed in this manual is BASIC (Beginner's All-Purpose Symbolic Instruction Code). The second requirement for all programs is that they must be completely and precisely stated. This requirement is crucial when dealing with a digital computer, which has no ability to infer what you mean -- it does what you tell it to do, not what you meant to tell it.

As an introduction to the BASIC programming language, we will solve a system of two simultaneous linear equations in two variables.

$$ax + by = c$$

$$dx + ey = f$$

If ae - bd is not equal to 0, this system can be solved by:

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

If ae - bd = 0, there is either no solution or many, but there is no unique solution. 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)
```

NOTE

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

Each line of the program begins with a line number and serves to identify each line as a statement. A program is made up of such statements, most of which are instructions to the computer. Line numbers serve to specify the order in which these statements are to be performed. Before the program is run, BASIC sorts out and edits the program, putting the statements into the order specified by their line numbers. This means that the program statements can be typed in any order, as long as each statement is prefixed with a line number indicating its proper sequence in the order of execution. Each statement starts after its line number with an English word which denotes the type of statement. Spaces have no significance in BASIC, except in messages which are printed out, as in line number 65 above. Thus, spaces may be used, or unused, at will to modify a program and make it more readable.

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

```
10  READ A, B, C, D
```

The first statement, 10, is a READ statement and must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing a program, it will cause the variables listed after the READ to be given values according to the next available numbers in the DATA statements. In this example, we read A in statement 10 and assign it the value 1 from statement 70 and, similarly,

with B and 2, and with D and 4. At this point, the available data in statement 70 has been exhausted; however, there is more in statement 80, and so we pick up the value 2 from 80 to be assigned to E.

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

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

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

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

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

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

```
30  READ C, F
```

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

The computer is now ready to solve the system:

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

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

In statements 37 and 42, we compute the value of X and Y according to the formulas provided, using parentheses to indicate that CE - BF is divided by G.

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

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

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

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

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

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

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

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

The particular choice of line numbers is arbitrary as long as the statements are numbered in the order the machine is to follow. We would normally number the statements 10, 20, 30, ..., 130, so that we can later insert additional statements. 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. In regards

to DATA statements, we need only put the numbers in the order that we want them read (the first for A, the second for B, the third for D, the fourth for E, the fifth for C, the sixth for F, the seventh for the next C, etc.). In place of the three statements numbered 70, 80, and 85, we could have 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 teletype.

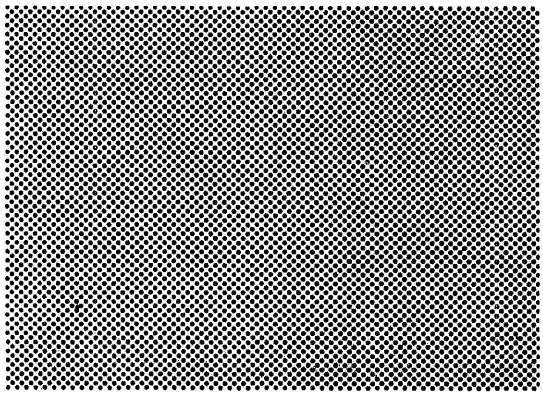
```
10  READ A, B, D, E
15  LET G = A * E - B * D
20  IF G = 0 THEN 65
30  READ C, F
37  LET X = (C * E - B * F) / G
42  LET Y = (A * F - C * D) / G
55  PRINT X, Y
60  GO TO 30
65  PRINT "NO UNIQUE SOLUTION"
70  DATA 1, 2, 4
80  DATA 2, -7, 5
85  DATA 1, 3, 4, -7
90  END
RUN

 4                    -5.50000
 0.666667              0.166667
-3.66667               3.83333
OUT OF DATA IN 30
```

NOTE

Typeouts from BASIC or from the Monitor are indicated in this text by underscoring.

After typing the program, we type RUN followed by a carriage return which directs the computer to compile execute the program.



II...

Language Coventions

STATEMENT NUMBER

All indirect statements in the program begin with a number. These numbers identify the statements in the program and specify the order in which the statements are to be executed. Before the program is compiled, the statements are ordered by ascending line number.

NOTE

Statement numbers must be decimal integers between 0 and 32767.

LINE LENGTH

The maximum number of characters that may be typed in a line is 72. Pressing the Line Feed key, while a statement is being typed, continues the statement on the next line. A statement may be continued for several lines provided that the maximum limit of 256 characters, including line feeds and carriage returns, is not exceeded. At the end of each entire statement, a Carriage Return must be typed.

SPACES

Spaces have no significance in BASIC except when they are included in strings. Spaces are used to improve the readability of the printed copy.

NUMBERS

A number may be positive or negative, contain up to nine digits, and is expressed in decimal form. For example, all of the following are

numbers in BASIC: 2, -3,675, 123456789, -.987654321, and 483.4156. The following are not numbers in BASIC: 14/3, $\sqrt{7}$, and .00123456789. The first two are formulas, but not numbers, and the last one has more than nine digits.

We gain additional 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 system in any of several forms: .123456789E-2 or 123456789E-11 or 1234.56789E-6. We may write ten million as 1E7 and 1965 as 1.965E3. We do not write E7 as a number, but must write 1E7 to indicate that it is 1 that is multiplied by 10^7.

IDENTIFIERS

Identifiers are the names by which variables and arrays are referenced in BASIC. Their form is:

- Any letter A to Z
- Any letter followed by any digit from 0 to 9

ARITHMETIC EXPRESSIONS

Arithmetic expressions are formed by combining numbers and/or variables with arithmetic operators as in ordinary mathematical formulas. There are five arithmetic operators in BASIC (Table 2-1).

TABLE 2-1. ARITHMETIC OPERATORS

Symbol	Meaning	Example
↑	Exponentiation	C↑2
*	Multiplication	3*B (=3xB)
/	Division	1/4(=1 ÷ 4)
+	Addition	8+F1
-	Subtraction	C8-5

Parentheses often are required in BASIC arithmetic expressions where they might not be needed in ordinary mathematical notation. For example, if you type $\frac{A+B}{C}$ as A+B/C in BASIC, the expression will be

interpreted as $A + \frac{B}{C}$. This is because BASIC performs division before addition, unless parentheses are used to denote otherwise. Thus, $\frac{A+B}{C}$ must be typed as (A+B)/C.

The order in which BASIC performs arithmetic operations is as follows:

1. Whatever is enclosed in parentheses will be computed first. When sets of parentheses appear within other sets of parentheses, the innermost set is evaluated first, then the next set, and so on.

2. Exponentiation.

3. Multiplication and Division. If * and / appear in the same expression, BASIC calculates from left to right; that is, 3/B↑2*C is equivalent to

$$\left(\frac{3}{B^2}\right) \times C$$

4. Addition and Subtraction. Similarly as above, if these two operators appear in the same expression, BASIC calculates from left to right.

MATHEMATICAL FUNCTIONS

A number of standard mathematical functions are available in BASIC. Each one has the same form: The name of the function followed by the argument enclosed in parentheses. Some of these functions are listed in the chart on the following page (See Table 2-2).

These functions may be included in any statement; for example, all of the following are acceptable:

```
80 LET B4 = Z8-EXP(X1+LOG(5/X1))
90 LET T = SQR(SIN(R)↑2+COS(Q) + 2
100 LET W4 = LOG(N*X-SIN(PI/N))
```

TABLE 2-2. STANDARD MATH FUNCTIONS

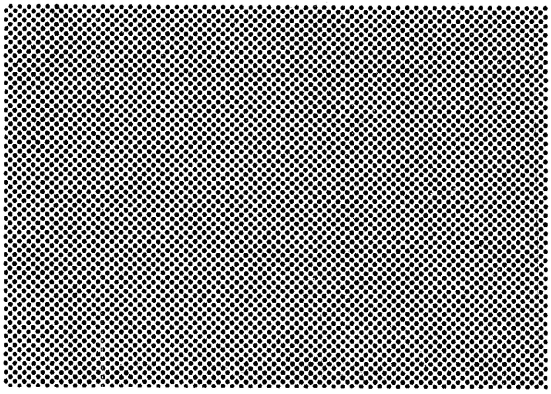
Function	Meaning
SIN(X)	Sine of X (X in radians)
COS (X)	Cosine of X (X in radians)
TAN (X)	Tangent of X (X in radians)
ATN (X)	Arctangent (in radians, over the range $-\pi/2$ to $+\pi/2$) of X
EXP (X)	Natural exponential of X, e^X.
ABS (X)	Absolute value of X
LGT (X)	Logarithm of X (base 10)
LOG (X)	Natural logarithm of X
SQR (X)	Square root of X

RELATIONAL EXPRESSIONS

A relational expression is one that compares one value to another (where the values may be represented by variables or arithmetic expressions), using the following relational operators:

SYMBOL	MEANING
<	Less than
<=	Less than or equal to
=	Equal to
>=	Greater than or equal to
>	Greater than
<>	Not equal to

(Relational expressions commonly occur in an IF statement where the THEN of the statement will be executed only if the specified relation is true.)



III...

Elements of the Language

The BASIC Language is composed of the following distinct statements:

DATA

Provides values to be assigned to variables by a READ statement.

General Form
DATA d_1, d_2, d_3

The values listed in DATA statements can be either numbers or literal strings, not expressions, and must be separated by commas.

The location of DATA statements in a program is arbitrary, although the usual procedure is to place them in a group at the end of the program. The only requirement is that the statements be numbered in the order in which the data is to be read.

DEF

Allows the user to define a function.

General Form
DEF $name_f$(parameters) = expression of function Where $name_f$ is FN followed by a letter (unique ID). Parameters are enclosed in ( ) and separated by commas.

Restrictions:

- Value of function must be computed in a single statement.
- Dummy parameters are local.

In addition to the standard BASIC library functions, the user may define any other function that he expects to use a number of times in a program. The indirect command DEF is used for this purpose. The names of programmer defined functions must contain three letters, the first two of which must be FN. Function names must be unique and they cannot be redefined within a program.

The form of the DEF statement is shown below; the programmer defines a function that will calculate the sine of an angle in degrees, using a library function.

```
10 DEF FNS(X) = SIN(X*3.14159/180)
```

An argument used in defining a function (X in the above example) is called a parameter. A programmer defined function can have either no parameters or any number of parameters (separated by commas and enclosed in parentheses). Parameters are "dummy" arguments; that is, when a defined function is used, certain specified values will temporarily replace the parameters where they appear in the function definition. For example,

```
>10 DEF FND(A,B) = 4*A*B+A 12
>20 Y = FND(2,1)
>30 PRINT Y
>RUN
>12
```

When the defined function was used in line 20, 2 and 1 replaced A and B respectively in the function definition in line 10. Thus, Y was set to $(4*2*1)+2^2$, or 12.

Parameters can have any variable name, including the names of variables used in the same program; in other words, the parameters are local to the function definition.

DIM

Dimension limits of 1, 2 or 3 dimensional arrays.

General Form
DIM name(d_1, d_2, d_3), name2...

If an array is to have a subscript different than 10, the size of the array must be specified by the DIM command. This command instructs BASIC to reserve a specified amount of space for array elements. For example,

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

will reserve 15 locations for elements A(1) to A(15). The DIM statement does not define any array elements; it simply allows a certain number of values to be accepted as input to the array.

Any number of arrays can be dimensioned in a single DIM statement as follows:

```
60 DIM K(20), L(3, 3), A5(12)
```

The user may save storage space by dimensioning arrays with subscripts less than 10, even though such dimensioning is not required. Thus, DIM E(3, 5) will reserve space for exactly 15 elements, whereas without the DIM statement, 100 (10X10) spaces would be reserved for the array E.

Subscripts always start from 1.

END

Terminates program execution.

General Form
END

The END statement identifies the physical end of the main program and results in transfer of control to the executive at object time. If subroutines have been identified in the main program by a GOSUB statement, the subroutines must follow the END statement.

FOR-TO-STEP

Specifies initial, increment value, and test value for loop iterations.

General Form
FOR var name = initial value TO final value STEP increment or FOR var name = initial value TO final value where the step size is implied to be 1.

NEXT

The increment, test, and iterate statement(s) of a loop.

General Form
NEXT var name

NOTE

If omitted - no test and increment occurs.
Loop body is executed once and falls through.

Since loops are so important and are used so often in programming, BASIC provides the two indirect commands FOR and NEXT to simplify loop specification. For example, suppose we want to write a program that will print out a table of the first 100 positive integers and their square roots.

```
10 FOR N = 1 TO 100
20 PRINT N, SQR(N)
30 NEXT N
```

Statement 10 specifies that N is initialized to the value 1 and that N should not be set to a value greater than 100[1]. The modification, an increase of 1 each time through the loop, is implied in this statement. The body of the loop is statement 20. The NEXT command in statement 30 instructs BASIC to return to the FOR statement for the next value of N. When the body of the loop has been executed for every specified value of N, BASIC will go to the statement following the NEXT.

NOTE

The value of N after exit from the loop is the final value assigned to N, 100.

N could have been increased to 100 in steps of any size other than the implied 1. To do this, we must specify the step size in a STEP clause. For example, suppose we want to print the square roots of the first 50 even integers. The program would be written as the one above with statement 10 replaced by:

```
10 FOR N = 2 TO 100 STEP 2
```

The specified step size may be negative. For example, if we want to print the square roots of the first 100 integers in descending order, statement 10 would be:

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

In the following example, statement 10 specifies that K should not be set to a value greater than 7. The final value assigned to K is 7.

```
>10 FOR K = 5 TO 7.
>20 PRINT K
>30 NEXT K
>RUN
 5
 6
 7
>
```

It is often useful to have loops within loops. The order in which BASIC must execute these nested loops is illustrated in the following skeleton examples:

<u>ALLOWED</u>

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

<u>NOT ALLOWED</u>

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

<u>ALLOWED</u>

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

Nested FOR loops of any complexity are allowed, but crossed FOR loops are not allowed.

GOSUB

A return jump to a closed subroutine. The subroutine is identified by line number.

General Form
GOSUB line number

When a part of a program is repeated several times in different places, it can be programmed more efficiently as a subroutine. Subroutine statements are written only once but can be used many times from any place in the main program. All subroutines must follow the end of the main program and be separated from the main program by an END statement. The command is used to transfer control to a subroutine. Its form is GOSUB followed by the line number of the first statement of the subroutine. The GOSUB command is similar to GO TO followed by a line number, in that it transfers unconditionally to another part of the program. GOSUB differs in that it does not go beyond the end of the subroutine, which must be indicated by a RETURN command. After a subroutine has been executed, return is to the statement following the one in which the GOSUB command was given.

The following example of a small subroutine shows two sections of the main program in which the GOSUB command is used.

```
10S = 3
20 GOSUB 400
30 PRINT H, P, X
.
.
.
100S = 7
110 GOSUB 400
120 Z = 3*H+P/X
.
.
.
390 END
400 H = S*SQR(2)
405 P = 2*S+H
410 IF P < = 10 THEN X = 1 ELSE X = 2
```

```
420 RETURN
.
.
.
```

When this program is run, line 20 instructs BASIC to transfer to the subroutine beginning at line 400. When the RETURN command at the end of the subroutine is reached, a return is made to line 30 (the line following the GOSUB command). Similarly, when the subroutine is called later from line 110, the return is to line 120.

A GO TO or an IF statement within a subroutine can cause transfer out of the subroutine before the RETURN command is reached. In addition, a subroutine can contain a GOSUB statement, which calls another subroutine.

Example:

```
.
.
.
40 X = SIN(Y+Z)
50 GOSUB 200
60 PRINT X
.
.
190 END
200 Q = X+R/S
210 IF Q < .5 THEN 240
220 PRINT "Q=";Q
230 GOSUB 500
240 RETURN
.
.
.
500 V = Q+R/S
510 PRINT "V=";V
520 RETURN
.
.
.
```

The subroutine beginning at line 200 contains both an IF... THEN... statement and a GOSUB command, which calls another subroutine. As specified in line 210, if Q < .5, a return is made (to line 60). When

Q > =. 5, the program continues with the next statements, in order, until it reaches the GOSUB 500 command. A transfer is then made to the subroutine beginning at line 500. Note the effect of the RETURN commands in this program; line 520 causes a return to line 240, which in turn causes a return to line 60 (the statement following the GOSUB 200 command).

Subroutines must be isolated from the main program; this is not done automatically by BASIC. The sequence of steps in the program must be designed so that the statements of the subroutine are executed only after a GOSUB command.

The indirect command END is used to isolate subroutines. This command causes execution of the program to terminate. All subroutines must be placed at the end of the main program and separated from the main program by an END statement as illustrated by the following:

```
10 ! MAIN PROGRAM BEGINS
.
.
.
100 GOSUB 700
.
.
.
690 END ! MAIN PROGRAM END
700 ! SUBROUTINE BEGINS
.
.
.
790 RETURN ! SUBROUTINE ENDS
```

GO TO

Unconditional transfer of control.

General Form
GO TO line number

The GO TO statements permit the programmer to alter the sequence in which program statements are executed. The statement transfers control to the specified statement number.

IF THEN ELSE

Conditional transfer of control.

General Form
IF condition THEN line number or IF condition THEN line number ELSE line number

The IF-THEN statement permits a conditional branch in a program. When the condition following the IF is true, control will be transferred to the designated line number. The next statement is executed when the condition is false.

The word ELSE, followed by a line number, can be added to the IF-THEN sequence. The form allows control to be transferred to the line number following the ELSE, if the condition is false.

Examples:

```
>70 IF X = .5 THEN 200 ELSE 300
```

(If X is .5, the program goes to line 200; otherwise, it goes to line 300.)

```
>100 IF X < 6 THEN 250
```

(If X is less than 6, the program transfers control to line 250. When X is greater than or equal to 6, the line following 100 is executed next.)

IF END DATA THEN

General Form
IF END DATA THEN line number

The IF END DATA statement determines if all the data has been read. If it has, control is transferred to the statement referenced by THEN. If all the data has not been read, all READ statements are initialized so that when one of them encounters the end of the data, it transfers to the statement designated by the IF END DATA statement.

INPUT

Accepts input typed in reply. Used to initialize variables in the list. Allows dynamic setting of values during program execution.

General Form
INPUT $Var_1, Var_2, \ldots Var_n$

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

```
40 INPUT X, Y
```

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

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

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

and the system types:

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

Without the comma 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 might take a long time to enter a large amount of data using INPUT; therefore, INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the running of the program such as with game-playing programs.

LET

Verb (optional) that identifies statement type as assignment.

General Form
LET var = expression or var = expression Where expression is a constant, variable name, or arithmetic expression.

This statement is not a statement of algebraic equality; rather, it is a command to the system to perform certain computations and to assign the answer to a certain variable. The word LET is optional on all assignment statements. For example, the following statements are identical in the BASIC language.

```
50 A = B + C/D
50 LET A = B + C/D
```

ON - GO TO

Provides for the computed transfer of control.

General Form
ON expression GO TO $line_1, line_2, \ldots$ where $line_1$, $line_2, \ldots$ is a sequence of line numbers to which the program transfers, depending on the value of the expression. If the value of the expression is 1, the program transfers to $line_1$; if the value of the expression is 2, the program transfers to $line_2$, and so on.

For example,

ON I*J GO TO 60, 70, 85

transfers to lines 60, 70, or 85 depending on whether the value of the expression I*J is 1, 2, or 3 respectively.

If the value of the expression is less than one or greater than the number of line numbers, an error message is printed. If the value of the expression is not an integer, the value will be truncated.

PRINT

The PRINT statement has the following different uses:

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

Each type is slightly different in form, but all start with PRINT after the line number.

Examples of type a:

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

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

Y, Z, B^2 - 4AC, and e^{A-B}. The system will compute the two formulas and print them for you, as long as you have already given values to A, B, and C. It can print up to five numbers per line in this format.

Examples of type b:

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

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

Examples of type c:

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

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

Examples of type d:

```
250 PRINT
```

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

Although the format of answers is automatically supplied for the beginner, the PRINT statement permits a greater flexibility for the more advanced programmer who wishes a different format for his output.

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

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

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

```
20 PRINT I,
```

you would have the numbers printed in the zones, reading:

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

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

Thus, the instruction:

```
50 PRINT X, Y
```

results in the printing of two numbers and the return to the next line, while:

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

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

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

```
250 PRINT
```

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

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

```
140 PRINT
150 NEXT M
```

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

ASSIGNING AND PRINTING STRING VALUES

A string value, like a numeric value, can be assigned to a variable with either an assignment statement, an INPUT statement, or a READ statement. Strings need be enclosed in quotation marks only in the event that commas are to be part of the string rather than a delimiter. Everything inside the quote marks is accepted except a Line Feed. A Line Feed indicates that the data is continued on the next line.

All forms of the PRINT command can be used to print strings. The effect of the comma is the same for printing string variables as for other print statements.

PICTURE FORMATTING

The user can specify his own format, for output, in addition to using the conventional BASIC forms of output. This feature, known as picture formatting, is useful in presenting calculated results in the form of tables and reports.

PRINT IN IMAGE STATEMENTS

The user may specify the exact format of his output by typing special characters in a string and using a PRINT IN IMAGE statement, as illustrated in the following example.

```
 10 INPUT A, B
 20 S = "E FORMAT #######, INTEGER %"
 30 PRINT IN IMAGE S:A, B
 RUN
?   200, 5.67
E   FORMAT .2E+03, INTEGER 6
```

In this example, S is a string variable that specifies the picture format to be used. The # signs in the string caused A to be printed in E format; the % signs caused the value of B to be rounded and printed as

in integer. All other characters in the string (including spaces) were printed as specified. The format symbols # and %, which are explained below, cannot be printed as part of the picture format because of their special significance.

The picture format string can include any of the specifications listed below. The numeric fields allow up to nine significant digits of a number to be printed, depending on the number of symbols used in the format string. If the specified format cannot be used for the number to be printed (for example, if an insufficient number of places is specified), the message, CANNOT FIT THIS FORM, is printed.

Integer Field

One or more % signs denote an integer field. One % sign must be typed for each digit of the number to be printed. Negative numbers require an additional % sign because of the preceding minus sign. A non-integer value will be rounded if an integer field is specified for it. For example,

```
A = 24, B = 174.78
PRINT IN IMAGE "%%%%%%%%":A, -A, B
 24 -24 175
```

Note the alternate form of the PRINT IN IMAGE statement illustrated above. Instead of a string variable whose value specifies the format, the picture format string itself is typed following IN IMAGE.

Integer fields are right justified; that is, if more % signs are specified than are necessary, leading spaces will be printed before the number. For example, the format "%%%" would cause 24 to be printed with one space before it, and 4 to be printed with two spaces before it.

Decimal Field

One or more % signs with an embedded decimal point denote a decimal field. The number to be printed is rounded to the specified number of decimal places. If the number is an integer or has fewer decimal places than the format specifies, trailing zeroes are printed. Negative numbers require an additional % sign because of the preceding minus sign.

Example:

```
10 X = 175.65, Y = 11
20 D = "%%%.%% %%%%%.%% %%.%"
30 PRINT IN IMAGE D:X, -X, Y
RUN
175.65 - 175.65 11.0
```

Decimal fields are right justified; that is, if more % signs before the decimal point are specified than are necessary, leading spaces are printed before the number.

NOTE

Whatever type of numeric field is specified in BASIC picture formatting, no more than nine significant digits of a number can be printed. If a number containing more than nine significant digits is printed with a field of more than nine symbols, the following occurs:

- Integer places, past the ninth significant digit, are filled with zeroes. For example, fourteen %'s will print the number 12345678901234 as 12345678901000.

- Decimal places, past the ninth significant digit, are replaced by blanks; for example, the field "%%%%%%%%.%%%%%" (in which eight %'s precede the decimal point and five follow it) will print the number 12345678.90123 and 12345678. followed by five blanks.

E Format Field

There are two forms for a field of E format:

1. A series of seven or more # signs.

2. One or more # signs, followed by a decimal point and a series of five or more # signs.

If the first form is used, the number printed begins with a decimal point. The second form allows the user to specify the number of digits before the decimal point. This is shown as follows:

```
10 C = 500
20 PRINT IN IMAGE "#######":C
30 PRINT IN IMAGE "##.#####":C
40 PRINT IN IMAGE "##.#####":-C
RUN
.5E+-3
50.E+01
-50.E+01
```

In the first form of the E format field, a minimum of seven # signs is needed:

1. The first # is for the leading space or minus sign of the mantissa (the number to the left of E).

2. The second # is for the decimal point of the mantissa.

3. The third # is for the minimum of one digit for the mantissa.

4. The fourth # is for the character E.

5. The fifth # is for the plus or minus sign of the exponent.

6. The sixth and seventh #'s are for the two digit integer exponent.

In the second form of the E format field, the # signs are used as follows:

1. A minimum of one # before the decimal point is for the mantissa.

2. Four #'s after the decimal point are for the exponential part.

3. The last # is for the leading space or minus sign of the mantissa.

NOTE

In the case of a positive number in E format, the leading space must be accounted for and always will be printed, while the integer and decimal fields allow this space to be suppressed.

Field of Strings

One or more % signs or # signs may be used to denote a string field. The number of symbols specified in the format determines how many characters of the string will be printed. For example, if A = "STRING", the format "%%%%%%%" of "######" may be used to print A. In the following example,

```
10 T = "CODE XY"
20 PRINT IN IMAGE "%%%%%%%":T
30 PRINT IN IMAGE "%%%%":T
RUN
CODE XY
CODE
```

the entire string is printed first; then, only four characters of the string are printed.

A string field is left justified; that is, if more % or # signs are specified than the number of characters in the string, trailing spaces are printed.

Descriptive Text in A Format

Any literal text may be included in the picture format string. Every character is printed exactly as it appears in the format, except for %, #, more than three $ or * symbols, [1] and decimal points. For example, the results of a program calculating the perimeter P and the area A of a triangle may be printed as follows:

```
110S = "PERIMETER IS %%.%, AREA IS %%%.%"
120 PRINT IN IMAGE S:P,A
```

[1] the meaning of these symbols is explained as follows:

Floating $ Field

This field is used to specify that a $ is to be printed immediately preceding an integer or decimal value (or a string). For example:

```
R = "$$$.$$ $$$.$$ $$$$"
PRINT IN IMAGE R:2.045,.7,300
$2.05   $.07   $300
```

These formats printed the specified values as the % formats would have, except that the last of the preceding spaces is replaced by a $. The $ always floats to the position before the first digit. If the $ field is specified so that there are no preceding spaces (that is, no room for the $), BASIC prints an error message. For example, 23.06 cannot be printed with the format "$$.$$".

The $ field must consist of four or more $ signs. For example, "$$$" is not a legal field, nor is "$$.$", since each of these contains only three $ signs. If these illegal fields were included in a format string, the characters would be taken as literal text and not a field designators. For example:

```
PRINT IN IMAGE "$%.%%":2.334
$2.33
```

Field designators

Text to be printed

The * Field

The * field is used to specify the * symbols are to appear before the number (or string) in place of the usual preceding spaces. For example,

```
S = "*** **.** ***.**"
PRINT IN IMAGE S:23,8.625,3.2
*23   *8.63   **3.20
```

These formats printed the specified values as the % formats would have, except that each preceding space is replaced by an *. If the * field is specified so that there are no preceding spaces (no room for an *), BASIC prints an error message. For example, 19.72 cannot be printed with the format "**.**".

The * field has the same restriction as the $ field. A minimum of four symbols is necessary. In the following example, "***" is interpreted as literal text rather than a field specification, and is printed as specified:

```
PRINT IN IMAGE "***##":"NOTE"
***NO
```

The * field is useful for check protection; that is, preceding *'s instead of spaces prevents anyone from adding to the beginning of the dollar amount on a check.

Image Repetition

Since the "picture" specified in an IMAGE format is the image of a line, a Carriage Return is supplied when the format is exhausted. Thus, if more values are to be printed than the number of fields specified, more than one line of the same image results.

Example 1:

```
PRINT IN IMAGE "%%":16.3,19
16
19
```

Example 2:

```
10 W = "%   %   %.%%"
RUN
1   2   3.00
4   5   6.00
7   8
```

NOTE

A picture format can also be specified by a string, formed by concatenation, that is:

```
G = "%%%"
F = "%%.%"
PRINT IN IMAGE F+G:16.3295
16.3295
```

PRINT IN FORM STATEMENTS

In addition to the line image type of picture format described above, BASIC provides a second type of format that uses IN FORM instead of IN IMAGE. The form of the output statements is similar; that is:

```
PRINT IN FORM S:A, B
PRINT ON 3 IN FORM S:X*Y, Z, W or
WRITE ON 3 IN FORM S:X*Y, Z, W
```

However, the format is field-oriented rather than line-oriented. The picture format string is not an image of the printed line, but specifies fields for whatever is to be printed, whether numbers, strings, descriptive text, or blanks.

Numeric, String, and Blank Fields

The symbols used to specify numeric and string fields are identical for IN FORM and IN IMAGE statements. One of the major differences between the two types of format statements is that when IN FORM is used, blanks typed between fields in the format string serve to separate the fields, but are not printed. For example, if M = 12 and N = 56.88, the statement

```
PRINT IN FORM "%% %%.%%":M, N
```

prints the values of M and N with no spaces between them. The blank in the preceding format serves only to separate the field for M from the field for N. To print blanks between numbers, use one or more B's to denote a field of blanks. Thus,

```
PRINT IN FORM "%% BBB %%.%%":M, N
```

prints the values of M and N with at least three spaces between them.

$ AND * FIELDS

These fields used with PRINT IN FORM yield the same results as when used with PRINT IN IMAGE, except that the sign of negative numbers is not printed. For example:

FIELD	PRINTS	IN IMAGE	IN FORM
$$$$	-16	$-16	$16
*****.**	-4.029	***-4.03	****4.03

CHARACTER AND FIELD REPLICATION

When IN FORM is used, the picture format can be written in a "shorthand" notation; that is, replication of characters and fields is permitted by using a multiplier. The following chart gives several examples of IN FORM character replication:

THE FORMAT	MAY BE TYPED AS
"%%%"	"3%"
"%%%%. %%%"	"4%. 3%"
"#######"	"7#"
"##. #####"	"2#. 5#"
"%% BBBB %%. %"	"2% 4B 2%. %"
"**********. **"	"10*. 2*"

The user also may specify the number of times a format field is to be used. The form of this field replication is,

N(format field)

where N is the number of times the format is to be used.

Example 1:

The format "2(3%. 2% B)"
is equivalent to "%%%. %% B %%%. %% B"

Example 2:

```
10 A = 543.66, B = 78.743, C = 345.788
20 G = "2(3%. 3% 4B) %%%"
30 PRINT IN FORM G:A, B, C
RUN
543.660   78.743   346
```

In this example, the field 3%. 3% 4B is used twice (to print A and B); then the field %%% is used to print C.

Example 3:

```
PRINT IN FORM "3(3%)":'6, 5, -1
16   5   -1
```

This statement specified three integer fields of three symbols each, with no blanks between the fields and, therefore, is equivalent to:

```
PRINT IN FORM "3% 3% 3%":16, 5, -1
```

Example 4:

The format: "20(4%. 2% B 4(3% B)/)"

illustrating two levels of field replication, may be used to print twenty lines, each with a decimal number and four integer numbers. A / generates a Carriage Return (see below).

NOTE

Up to four levels of field replication are allowed in a format.

Field for Descriptive Text

When IN FORM is used, any literal text that is to be printed must be enclosed in single quote marks to denote a text field.[1] For example,

```
10 D = "'X EQUALS' B.6%
20 X = P1/180
30 PRINT IN FORM D:X
RUN
X EQUALS .017453
```

Carriage Return in a Format

Unlike a format used in an IN IMAGE statement, no Carriage Return is given when the IN FORM format is exhausted. Thus, if fewer fields are specified than the number of values to be printed, the format is repeated on the same line as shown below.

[1] If the format string is enclosed in single rather than double quote marks, the literal text to be printed is enclosed in double quotes.

```
10 T = "% 2B %. % 2B"
20 PRINT IN FORM T:I FOR I = 1 TO 5
RUN
1   2.0   3   4.0   5
```

A slash (/) can be used in a format to generate a Carriage Return. Consecutive slashes may be used to generate blank lines. Note the results when the format above is modified to end with a / instead of 2B:

```
10 T = "% 2B %. %/"
20 PRINT IN FORM T:I FOR I = 1 TO 5
RUN
1   2.0
3   4.0
5
```

Remember, because an IMAGE format is the image of a line, a Carriage Return is always generated automatically when the format is exhausted.

The Single #

A single # may be used with PRINT IN FORM to specify what is known as "free field" format. Any number or string may be printed with this field. Up to eleven significant digits of a number can be printed. If the free field format is used to print a string, the entire string is printed. For example:

```
10 A = "STRING"
20 B = 68.9
30 C = 666
40 PRINT IN FORM "#":A, B, C, PI
RUN
STRING 68.9  666.  3.1415926535
PRINT IN FORM "#":123456789012345
.12345678901E+15
```

READ

Used, in conjunction with DATA, to assign values to variables.

General Form
READ Var_1, Var_2, ... Var_n

The READ command is always followed by a variable name or a list of variable names separated by commas. When BASIC executes a READ statement, the first variable listed in the statement is assigned the first value in the collection of DATA statements (the "data block"), the next variable is assigned the next value, and so on.

REM or !

Allows the inclusion of remarks in program or following statement (if ! used)

```
20 REM THIS IS A COMMENT
    or
30 ! THIS TOO IS A COMMENT
40 LET M = A/2 ! M IS SET TO A DIVIDED BY 2
```

Either an exclamation point (!) or the word REM is used to insert remarks or comments as indirect statements.

Remarks are listed along with the rest of the program, but are not printed out when the program is run. Any characters can be typed after ! or REM.

In addition, ! can be used to insert comments at the end of statements.

RESTORE

Resets pointer to beginning of data block such that the next READ executed is read from first word of data block.

General Form
RESTORE

Once all the data has been read from a data block, another READ request will cause an error message telling you that you are out of data. However, if you wish at any time during the program to reread all or part of the data block, you can do this with a RESTORE command. When this command is executed, the next READ command starts reading data from the beginning of the data block; that is, from the first value in the first DATA statement.

RETURN

Provides exit from a block of code identified as a subroutine.

General Form
50 RETURN

A subroutine can only be exited via a RETURN statement. Control is given to statement following the one in which the GOSUB command was given. A subroutine may contain several RETURN statements; however, the first one executed returns control to the calling program.

STRINGS

In sample programs throughout this manual, use has been made of the ability of the PRINT command to print straight text that has been enclosed either in a set of quotation marks or a set of apostrophes:

```
10 FOR N=1 TO 4
15 READ X
20 PRINT X; "SQUARED IS";X↑2
25 NEXT X
30 DATA 9.2,  12.04,  37,  7.34
35 END
```

The phrase "SQUARED IS", in line 20, is called a "string". Such strings can be made up of any mixture of alphabetic characters, numerals, and the special characters (including blank).

The use of strings is not confined to the PRINT statement. Strings can be considered in the same class as numeric values and can be handled in much the same way. String values can be assigned to variables. They can be entered via INPUT statements of DATA and READ statements. They can be elements of arrays. As an example:

```
100 STRING A, B, C
200 A="INCOME"
300 B="EXPENSES"
400 C="BALANCE"
500 PRINT A, B, C
600 END
```

Execution of this program results in the print out of the columnar headings:

```
INCOME          EXPENSES          BALANCE
```

Note that line 100 was needed to establish that the variables A, B, C are string variables. The absence of line 100 would have left A, B, C as REAL variables and would have caused an error.

The length of a string is the count of the characters appearing between the quotation marks or apostrophes. Strings can be of any length up to 255 characters. String variables are automatically set to accept strings of length up to 16 characters. If a string variable is to accept a string that is longer than 16 characters, suitable provision must be made in the string variable declaration:

```
150 STRING A:20, B, C:45
```

Line 150 prepares A to receive any string of up to 20 characters, B to receive any string of up to 16 characters, and C to receive any string of up to 45 characters. When a string value is assigned to a variable and the length of the string is longer than allowed for in the variable, truncation from the right takes place.

As an example of entering string values with an INPUT statement:

```
10 STRING A, B, C, D, E
20 INPUT A, B, C, D, E
30 PRINT A
40 PRINT B
50 PRINT C;D;E
60 END
```

When the INPUT statement is executed, BASIC looks for five values as input data. Suppose the user supplies (from his terminal or from data statements) the line:

```
"J. JONES", "82 MAIN ST.", "MYTOWN,", "CAL.", "92061"
```

Completion of execution of this program results in the printout:

```
J. JONES
82 MAIN ST.
MYTOWN, CAL. 92061
```

As another example, illustrating string values in DATA statements:

```
STRING C,D
16 READ C,D
18 READ A,B
20 X=A*B
22 PRINT C:A;D:X
24 DATA "PRINCIPLE=","INTEREST="
26 DATA 1200,.06
28 END
```

The printout resulting from the execution of this program is:

```
PRINCIPLE=1200      INTEREST=72
```

Within strings, any enclosed blanks are retained; in fact, a string can be made entirely of blanks:

```
100 STRING A,B,C
120 A="COLUMN 1"
140 B="          "
160 C="COLUMN 2"
180 PRINT A:B:C
185 END
```

When this program is executed, the colons in line 180 causes the printed phrase COLUMN 1 to be separated by the printed phrase COLUMN 2 by exactly the number of blank spaces between the quotation marks in the string value assigned to string variable B.

Strings are always handled just as strings even when they comprise totally valid numbers, e.g., "123.60". Thus, arithmetic is not performed on strings. However, the plus sign (+) can be used to concatenate strings:

```
100 STRING M,N,P
200 M="12345"
210 N="67890"
220 P=M+N
300 PRINT P
350 END
```

The printout from execution of this program is:

1234567890

Of course the same printout would result if line 300 had instead read PRINT M:N or even PRINT M+N.

And another example, the program:

```
10 STRING A, B
20 A="PART NO. "
30 B="AN703"
40 PRINT A+B+", NOT AVAILABLE"
```

results in the printout:

PART NO. AN703, NOT AVAILABLE

There are standard functions that the user can call for the manipulation of strings in BASIC. One of these converts strings, made up of characters expressing a number into numeric values, which can be used in arithmetic expressions and calculations. These functions are VAL(x), which convert the numeric string x to its REAL value respectively.

The following program illustrates the use of each of these functions on a string made up of numeric characters and a decimal point. Because this string has more than 16 characters, its declaration must include the announcement of its length:

```
110 STRING X:19
120 X="234.567891011121314"
130 A=VAL(X)
260 PRINT "A=":A
350 END
```

Execution of this program causes the following printout:

A=2. 3456789101E2

An operation, opposite to the one above, can be performed which converts a numeric expression to a string of numeric characters. This function is STR(x).

The following program illustrates the use of each of these functions on a numeric value expressed as a type DOUBLE number:

```
 70 DEFAULT STRING
 80 DOUBLE Y
 90 Y=234.567891011121314
100 A=STR(Y)
160 PRINT "A=":A
210 END
```

Execution of this program results in the following printout:

```
A=2.3456789101E2
```

It was necessary to include line 120 in order for the contents of C not to be truncated to the standard maximum of 16 characters. Note that even though line 120 reserved space for 30 characters in string C, string C was assigned a string having only 21 characters.

Bear in mind that the contents of string variable A above is a string even though it appears to be a number. This string value, of course, cannot be used in arithmetic expressions.

There are three string functions in BASIC that permit extraction of portions of string values and assigning these portions to other string variables. The LEFT function extracts the number of characters specified by the second argument, starting with the leftmost character, from the string given or referenced in the first argument:

```
 50 STRING A,T
100 A="JONES:$500"
150 T=LEFT(A,5)
200 PRINT "T=":T
250 END
```

Execution of this program results in the following printout:

```
T=JONES
```

The RIGHT function is the same but the character count begins with the rightmost character:

```
100 PRINT RIGHT ("JONES:$500",4)
150 END
```

Execution of this program causes the printout:

$500

Central portions of a string may be extracted with the SUBSTR function by specifying in the argument the position of the desired starting character, before specifying the number of characters to be extracted:

```
10 N=2
20 STRING A,B
30 A="ELEMONOPEE"
40 B=SUBSTR(A,4,N+1)
50 PRINT "B=":B
60 END
```

Execution of this program causes N+1 characters to be extracted from the string starting with the fourth character:

B=MEN

To count the number of characters in a string, the LENGTH function is used:

```
15 STRING X
20 X="ROTTERDAM"
25 Y=LENGTH(X)
30 PRINT "Y=":Y
35 END
```

Execution of this program prints out:

Y=9

To determine where a substring starts within another string, the INDEX function is used:

```
10 STRING X
12 X="GOBBLEDYGOOK"
20 A=INDEX(X,"B")
30 B=INDEX(X,"GOO")
35 C=INDEX(X,"M")
39 PRINT A;B;C
40 END
```

The printout from execution of this program is:

```
 3        9        0
```

A string of spaces can be generated by use of the SPACE function:

```
STRING A,B,C,D,X
20 A="SMITH"
30 B="HARRY",C="VIOLET",D="JIMMY"
40 X=SPACE(10)
50 PRINT A:SPACE(10-LENGTH(A)):B
60 PRINT X:C
70 PRINT X:D
80 END
```

Execution of this program prints out:

```
SMITH          HARRY
               VIOLET
               JIMMY
```

(The colons in this program could alternatively be plus signs.)

Strings can be compared with each other using relational operators in conditional expressions.

```
 5 STRING A
10 INPUT A,B
20 IF A="OVERDRAFT" THEN B=-1
30 PRINT "BALANCE=";B
40 END
```

Strings need not be compared only for one-to-one match or mismatch. They can also be compared to see if one is "greater than" or "lesser than" the other. This can be done because BASIC assigns each character a hierarchical position. Numerals are ordered in accordance with their value. Alphabetic characters have higher positions than numerals and are in alphabetic order. Thus:

$$3 < 8 < C < R$$

As an example of the application of this capability:

```
100 DEFAULT STRING
200 INPUT X
250 Y=LEFT(X, 1)
270 PRINT X;
280 IF Y < "N" PRINT "FIRST HALF" ELSE PRINT
    "2ND HALF"
300 END
```

If strings to be compared are of different lengths, the shorter is compared with the same number of characters first appearing in the longer word. If they match, the shorter is considered the lesser.

Because pairs of either quotation marks (") or of apostrophes (') can be used to delimit strings, quotations can be included within a string. Thus the line:

```
70 PRINT "THE PROGRAM IS CALLED 'PAYROLL'"
```

will result in the following printout:

```
THE PROGRAM IS CALLED 'PAYROLL'
```

The roles of the quotation marks and apostrophes in line 70 could be interchanged. It is only necessary that pairs be matched.

The table below is a summary listing of BASIC string functions.

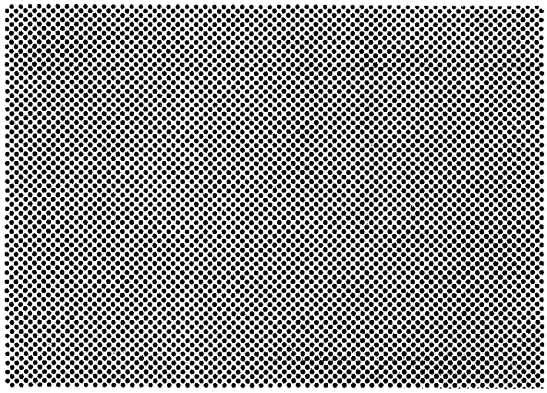
In the following example:

```
20 STRING A(12):10, B(3, 5):11
```

reserves space for a 12 element string array A, each element of which can contain up to 10 characters, and a 15 element array B with maximum string length of 11 characters per element.

TABLE 3-1. SUMMARY OF STRING FUNCTIONS

Function	Result
VAL(x)	REAL value of numeral string x.
STR(x)	Numeral string from the REAL value of numeric expression x.
LEFT (x,n)	A string of the first n characters of string x.
RIGHT (x,n)	A string of the last n characters of string x.
SUBSTR(x,n,m)	A string of m characters from string x starting at the nth character.
LENGTH(x)	The number of characters in string x.
INDEX(x,g)	The starting position of string g within string x.
SPACE(x)	A string of x spaces (blanks).



IV

Basic Editing Capabilites

This section describes the editing features that are included in the BASIC language. These features include the ability to change, add, and delete a statement. In addition, the capability exists for copying parts of a statement, backspacing, etc.

Delete and Edit command formats for BASIC differ from commands with the same name available through the Editor, in the manner of designating a statement or line of text. Designation of a statement is done by specifying the statement number.

DELETING STATEMENTS

The DELETE command allows the user to delete, from his source input, a specified statement or statements. One or more statements can be deleted with one command. Multiple statement numbers must be separated by commas, and the command is completed with a carriage return. On completion of the command, the response at the terminal is, "nDELETED", where n is the statement number.

Example 1:

```
>DELETE 10
10 DELETED
```

The preceding command would cause statement 10 to be deleted from the source record or file.

Example 2:

```
>DELETE 10, 20, 25)
10, 20, 25 DELETED
```

The preceding command causes statements 10, 20, and 25 to be deleted from the source record or file.

EDITING STATEMENTS

The EDIT command allows the user to select a specified source statement for editing. On completion of the command, the response will be the selected line being typed on the terminal. Editing of the line can proceed on receipt of a signature character, > , at the terminal.

Example 1:

```
>EDIT 20)
20 LET A(i,j) = (B(i,k) + C(10, 20))/F
>
```

The ability to back space, delete the previous word, delete the previous character, etc., is provided; Intraline Editing. When the edit is complete, the edited line replaces the original line in the program.

#1

```
>10 A(i,j) = B(K,L)) →BASIC STATEMENT
>15 C(10, 20) = D)   →BASIC STATEMENT
>DELETE 15)          →EDIT COMMAND TO DELETE
                        STATEMENT 15
15 DELETED           →TERMINAL RESPONSE
>Z^c =               →COPY PREVIOUS STATEMENT
                        TO =
```

→THIS INDICATES THAT Z^c IS NOT VALID BECAUSE THERE IS NO PREVIOUS LINE

#2

```
>10 A(i,j) = C↑2 + D↑3)     →BASIC STATEMENT
>20 B(i,j) = A(i,j) + B(i,j)) →BASIC STATEMENT
```

```
>EDIT 10)                          → EDIT COMMAND TO EDIT
                                      STATEMENT 10
10 A(i, j) = C↑2 + D↑3)           → TERMINAL RESPONSE
>Z^C +                              COPY PREVIOUS STATEMENT
                                      TO +
>10 A(i, j) = C↑2 +               → RESPONSE AT TERMINAL
                    D↑2)          → EDITED PORTION INPUT BY
                                      USER
```

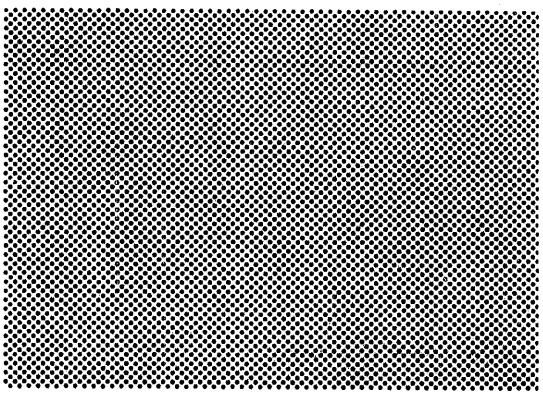
#3

```
>10 C = D↑2 + E↑2)                → BASIC STATEMENT
>15 G = C↑2)                      → BASIC STATEMENT
>Q^C                              → INTRALINE EDIT COMMAND
                                      TO DELETE PREVIOUS LINE
```

```
(BELL)                            → THE TERMINAL RESPONSE
                                      RINGS A BELL TO INDICATE
                                      THAT THE Q^C IS NOT VALID,
                                      AND STATEMENT 15 SHOULD
                                      BE DELETED BY A DELETE
                                      COMMAND
```

#4

```
>10 C = D↑2 + E↑2)                → BASIC STATEMENT
>15 G = C↑2 Q^C                   → BASIC STATEMENT, DELETE
                                      VIA Q^C
>LIST                             → LIST SOURCE COMMAND
 10 C = D↑2 + E↑2                 → RESPONSE TO LIST
```



V...

File and Data Base Management Statements

GENERAL

BASIC provides an interface to a more general 2+2 file and data-base manager subsystem. This interface consists of a simple set of declarative and operational statements useful in building, manipulating, and accessing data bases used in storing and retrieving information.

With respect to the data-base manager, a file may be considered to be a named space consisting of an ordered sequence of elements. All internal structuring defined for a file, and all access to the contents of a file, are controlled by the data-base manager.

The structure, access, and control of a file is determined by the "schema" definition associated with a file. A "schema" allows for the definition of file to include logical "areas" within the file. "Records" within "areas" of a file are also defined. A "record" is further defined into elements consisting of a name, type, mode, and size. "Schema", "areas", and "records", once defined are referred to by name. Rather than be concerned with the form or structure of the data-base manager, itself, the BASIC programmer need only concern himself with the manipulation of files and data bases in terms of the statements described below.

NOTE

For a complete description of the data-base manager, the BASIC programmer is advised to obtain the 2+2 system documentation of the data-base manager for a detailed description of "schema", "area", "record", etc.

INDIRECT STATEMENTS

The statements are grouped as follows:

1. Directory Content Statements

   The statements CREATE, RENAME, ERASE, and DESTROY transmit information to the file manager used to build, modify and update files. CLOSE file indicates deactivation of a previously active file.

2. General Input/Output Statements

   The RETRIEVE statement causes transmission of a set of quantities from a file to core storage.

   The statements, INSERT, APPEND, REPLACE, and REMOVE cause transmission of a set of quantities from core storage to a file in a manner appropriate to the specified statement.

File Designation

Files may be designated by a file name constant or file name variable. Within the BASIC system, the file name ("Δ") blank is used to designate the user terminal.

Directory Content Statements

CREATE a File. The CREATE statement is used to enter a new file name in the user's directory. The file is initially set to contain no information. Optionally, access privileges and additional user access may be specified.

General Form
CREATE file name where: file name is a file designator which may be a file name constant or file name variable.

Examples:

```
CREATE "A"
CREATE "BETA"
CREATE "PAYABLES"
```

The creator of a file is assigned full access privileges to that file. Those privileges include READ, WRITE, and EXECUTE access.

To assign privileges to other users who wish to access a file, the CREATE statement may be expanded in the following form:

CREATE file name/USER ID/ACCESS/USER ID/ACCESS...

Where file name is file designator:

> USER ID is a unique user identification string (not currently defined as to form or length) delimited by slashes (division signs).
>
> ACCESS is any combination of the following words separated by commas, -READ, -WRITE, EXECUTE, APPEND, or PASSWORD.

Any number of users may be authorized access to a file. However, access privileges must be specified for each authorized user. Public access is specified by a null user ID (i.e., //). Consider,

CREATE "PAYROOL"/AB234/READ/J. SHMOE/APPEND, EXECUTE

Creates the file named MILLISIN. It also makes the file public with both READ and EXECUTE privileges.

RENAME a File. The RENAME statement allows a user to rename any file file currently in a user's directory. Optionally, additional users and access privileges may be specified.

General Form
RENAME file name, file name where: file name is a file designator which may be a file name constant or file name variable.

Example:

RENAME "NEWTRANS", "OLDTRANS"

This renames the file currently in the user's directory as NEWTRANS to a file named OLDTRANS. The name NEWTRANS is removed from the user's directory, replaced by the name OLDTRANS.

Additional access privileges may be specified while renaming a file by attaching user ID's and access. For example,

```
RENAME "ABC", "DEF"/USER27/READ,WRITE/USER33/EXECUTE
```

File ABC is renamed as file DEF. In addition, USER27 is authorized to READ and WRITE the file, and USER 33 is granted execute only access.

ERASE a File. The ERASE statement allows a user to erase or clear the information currently contained in one or more files.

General Form
ERASE file name, filename, ... where: file name is a file designator which may be a file name constant or file name variable.

Examples:

```
ERASE "R"
ERASE "MONTH04", "MONTH05", "MONTH06"
```

DESTROY a File. The DESTROY statement provides the method for destroying both the information contained in a file and removing the entry containing the name of the file in the user's directory. That is, both the contents and the name of the file are destroyed and unrecoverable.

General Form
DESTROY file name, file name, ... where: file name is a file designator which may be a file name constant or file name variable.

Examples:

```
DESTROY "ACCOUNTS"
DESTROY "SMALLEST", "AVERAGE", "LARGEST"
```

CLOSE a File. The CLOSE statement is used to signal the system that processing of an active file has been completed. The specified file is deactivated. Subsequent statements referencing the file will reactivate the file and its contents. This statement need only be executed if a given BASIC program is concerned about simultaneously activating more files than the maximum number of allowable active files defined by the 2+2 system.

General Form
CLOSE file name, filename, ... where: file name is a file designator which may be a file constant or file name variable.

Examples:

```
CLOSE "XYZ"
CLOSE "FILE1", "FILE3", "FILE5"
```

General Input/Output Statements

Input. The RETRIEVE statement inputs quantities from a file to be processed by the computer program.

General Form
RETRIEVE file name (schema(area(record))) list where: file name is a file designator. schema is the schema name applied to file name. area is an area name within the schema being accessed record is a record name within the area of a schema. list is a list specification.

The RETRIEVE statement reads one record from the file name using schema/area/record name to isolate the desired record. The contents of the variables named in the accessed record replace the contents of those same variables in the BASIC program and may be used for computation.

The list specification is optional and need only be used when it is desirable to access only a few of the items defined in a record definition.

There must be a one-to-one correspondence between the variable names in the BASIC program, and the variable names used in a schema schema/area/record name definition.

The parenthesized arguments - area name and record name are optional so long as no ambiguity exists within the schema.

Examples:

RETRIEVE "MASTFILE" (SCH1(AREA2(REC3)))

RETRIEVE "MASTFILE"(SCH1(AREA2(REC3)))A, B
In this example, assume the definition of REC3 defined the variables A, B, C, D, &E. Only variable A, B are accessed via this statement.

RETRIEVE "MASTFIL"(SCH1)
This statement would be equivalent to example 1 if the schema, SCH1, defined only 1 area and 1 record.

Output. The statements, INSERT, APPEND, REPLACE, and REMOVE, output quantities contained in BASIC program variables to a file managed by the data-base manager.

General Form
APPEND INSERT NEXT INSERT PRIOR file name (schema(area(record))) list REMOVE REPLACE where: file name is a file designator schema is the schema name applied to file name. area is an area name within the schema being accessed. record is a record name within the area in the schema. list is a list specification.

The output statements write one record from core storage to file name according to the schema/area/record names specified. The contents of variables in core storage (BASIC program) defined by the schema are written appropriately into the specified file using the desired schema.

The APPEND statement adds a record at the end of the file according to the specified schema.

The INSERT NEXT statement inserts a record after the current record position of the file according to the specified schema.

The INSERT PRIOR statement inserts a record just before the current record position of the file according to the specified schema.

The REMOVE statement removes or deletes the record at the current record position of the file. In this case, the schema definition is used only to isolate and remove the desired record from the file.

The REPLACE statement replaces the record at the current record position of the file according to the specified schema.

The list specification is optional and need only be used when it is desirable to write out fewer items in a record than specified in the schema definition of a given record.

There must be a one-to-one correspondence between the variables names in the BASIC program, and the variable names used in a schema/area/record name definition.

The parenthesized arguments - area name and record name are optional so long as no ambiguity exists within the schema.

Examples:

APPEND "ORDERS"(ORDSCH(NEWAREA(NEWREC)))

APPEND "ORDERS"(ORDSCH)
This statement is equivalent to the first example if the "ORDERS" file consists of one area and one record type.

INSERT NEXT "DATEFILE"(DATSCH(HOLIDAY)))
This statement writes one record in the "DATEFILE", according to the schema DATSCH. Area HOLIDAY is written, and the statement assumes only one record type, hence no record name appears.

INSERT PRIOR "SIGMAS" (STATSCH(SQUARES(SUM))) A, B
This statement inserts a record prior to the current record position of the file SIGMAS. The schema STASCH, with area SQUARES, containing record SUM is written. However, since a list is specified, only variables A & B of record SUM are written - even is the record definition defined more elements than A & B.

REMOVE "CATALOG" (CATSCH((NAMREC)))
The parenthesized quantities in the example show a record deleted from the file "CATALOG". This example assumes the schema CATSCH defines only one area. Within the one area, the current record NAMREC is deleted.

The following examples illustrate possible combinations of optional parameters.

REPLACE "ENGFIL" (SCHEMA1(AREA2(REC3)))
REPLACE "ENGFIL" (SCHEMA1(AREA2(REC))) A, B, C
REPLACE "ENGFIL" (SCHEMA1(AREA2))
REPLACE "ENGFIL" (SCHEMA1((REC3)))
REPLACE "ENGFIL" (SCHEMA1)

DIRECT STATEMENTS

CREATE a File

The CREATE statement is used to enter a new file name in the user's directory. The file is initially set to contain no information. Optionally, access privileges and additional user access may be specified.

General Form
CREATE file name where, file name is any legal user file name not exceeding eight characters.

Examples:

```
CREATE Z
CREATE ALPHA
CREATE MASTFILE
```

The creator of a file is assigned full access privileges to that file. Those privileges include (currently) READ, WRITE, EXECUTE, and APPEND access.

To assign privileges to other users who wish to access a file, the CREATE statement may be expanded in the following form:

CREATE file name/user ID/access/user ID/access . . .

where file name is any legal file name of 1 to 8 characters

user ID is a unique identification string delimited by slashes.

access is any combination of the following words separated by commas: READ, WRITE, EXECUTE, APPEND, or PASSWORD.

Any number of users may be authorized access to a file. However, access privileges must be specified for each authorized user. Public access is specified by a null user ID (i. e., 11). Consider,

```
CREATE PAYROLL/AB234/READ/J. SHMOE/APPEND, EXECUTE
```

The file PAYROLL is created. User AB234 is given READ access, and J. SHMOE is authorized to EXECUTE and APPEND to the file. The statement,

CREATE MILLISIN//READ, EXECUTE

creates the file named MILLISIN. It also makes the file public with both READ and EXECUTE privileges.

RENAME a File

The RENAME statement allows a user to rename any file currently in a user's directory. Optionally, additional users and access privileges may be specified.

General Form
RENAME file name, file name where file name is any legal file name of 1 to 8 characters.

Example:

RENAME NEWTRANS, OLDTRANS

This renames the file currently in the user's directory as NEWTRANS to a file named OLDTRANS. The name NEWTRANS is removed from the user's directory, and is replaced by the name OLDTRANS.

Additional access privileges may be specified while renaming a file by obtaining user ID's and access.

Example:

RENAME ABC, DEF/USER27/READ, WRITE/USER33/EXECUTE

File ABC is renamed DEF. In addition, USER27 is authorized to READ and WRITE the file, and USER33 is granted execute only access.

ERASE a File

The ERASE statement allows a user to erase or clear the information currently contained in one or more files.

General Form
ERASE file name, file name, . . . where file name is any legal file name of 1 to 8 characters.

Examples:

```
ERASE Q
ERASE WEEK32, WEEK33, WEEK34
```

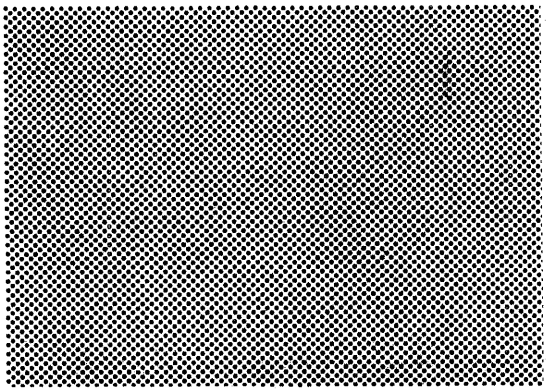
DESTROY a File

The DESTROY statement provides the method for destroying both the information contained in a file and removing the entry containing the name of the file in the user's directory. That is, both the contents and the name of the file are destroyed.

General Form
DESTROY file name, file name, . . . where file name is any legal file name of 1 to 8 characters.

Examples:

```
DESTROY FIRST
DESTROY SECOND, THIRD, FOURTH
```



VI . . .

Diagnostics and Debugging aids

GENERAL

Occasionally, a new program will be error free and give the correct answer the first run -- but more commonly, errors will be present and have to be corrected. Errors are of two types: 1) grammatical or form errors that prevent the running of the program, and 2) logical errors in the program that cause the computer to produce incorrect answers.

The BASIC language provides a set of diagnostic messages to aid the programmer in identifying grammatical or form errors in a program. Most diagnostic messages not only identify the type of error, but also indicate the line number where the error occurred. Section VII provides a list of the diagnostic messages and their complete meanings.

Logical errors are usually more difficult to detect, particularly when the program responds with answers that seem to be nearly correct. BASIC provides the following capabilities to assist the programmer in finding logical program errors.

DIRECT STATEMENTS

Direct statements are elements of the basic language that are entered without line numbers. These statements are executed when received by the system, and do not become a part of the program.

GOTO STATEMENT

General Form
GOTO LN Where: LN is a line number

The GOTO statement may be invoked only during program execution in debug mode. The debug is entered by executing a BREAK statement; the DEBUG statement; or via some run-time error diagnostics. The GOTO statement may be used after receiving the debug mode signature character (?), question mark, at the user's terminal.

The argument of the GOTO statement is the line number of the program at which it is desired that execution of the program resume.

```
? GOTO 1220
```

resumes execution of the program at line number 1220.

The GOTO statement may be executed any time program execution is suspended in debug mode.

LET Statement

General Form
LET VAR = {LITERAL / VAR}
Where: VAR is a variable or array element name LITERAL is a numeric or string constant

The LET statement may only be invoked during program execution in debug mode. The debug mode is entered by executing a BREAK statement; the DEBUG statement; or via some run-time error diagnostics. The LET statement may be used after receiving the debug mode signature character, (?), question mark at the user's terminal.

The arguments for the LET statement may be variables and literals. A variable is a variable name or an array element name. A literal is a numeric constant or a string constant. No expressions are evaluated. Consider

```
? LET A(2, 3, 4) = 27.5
```

when execution of the program is resumed, the array element A(2, 3, 4) will contain the value of the literal, 27.5.

```
? LET X = B(32)
```

when execution is resumed the variable X will contain the value contained in B(32).

The LET statement may be executed any time program execution is suspended in debug mode.

PRINT Statement

General Form
PRINT VAR, VAR, VAR, ...
Where: VAR is a simple variable or array name.

The PRINT statement may only be invoked during program execution in debug mode. The debug mode is entered by executing a BREAK statement, the DEBUG statement, or via some run-time error diagnostics. The PRINT statement may be used after receiving the debug mode signature character, (?), question mark, at the user's terminal.

Arguments for the PRINT statement may be variable names, array names, or elements of an array. No expressions are evaluated. For example,

```
? PRINT X, B, C(3, 5)
```

where X is a variable
B is an array dimensioned as B(5)
C(3, 5) is an element of the array C,

results in the following output at the user's terminal:

```
X = XXXX
B(1) = XXXX
B(2) = XXXX
  .
  .
  .
B(5) = XXXX
C(3, 5) = XXXX
```

The PRINT statement may be executed any time the program execution is suspended in debug mode.

DEBUG AIDS

The BASIC language provides five main debug aids. These are: controlled execution, trace prior to statement execution, break prior to statement execution, trace after a variable value change, and break after a variable value change.

CONTINUE Statement

The CONTINUE statement may be invoked only during execution of program compiled in debug mode. Debug mode is entered as described in PRINT, LET, and GOTO (Items 23 - 25).

CONTINUE sets the mode of execution program to normal or multiple step mode. It should be used to reset the STEP statement when single step execution is no longer desired.

The CONTINUE statement may be executed any time program execution is suspended in debug mode.

STEP Statement

The STEP statement may be invoked only during execution of a program compiled in debug mode. Debug mode is entered as described in PRINT, LET, and GOTO.

STEP sets the mode of execution of a program to single step mode. Statements compiled in debug mode are executed in single step. That is, after execution of each statement, the program is interrupted; the line number of the statement and the debug signature character (?) are typed at the user's terminal. The user may then execute any of the debug statements; alter MONITOR, BREAK and TRACE statements; reset CONTINUE mode; or resume execution of the program.

The STEP statement may be executed any time program execution is suspended in debug mode.

BREAK Statement

General Form
BREAK LN_1, LN_2-LN_3, VAR, (VAR, N_1-N_2), ...

The BREAK statement is similar in execution to the MONITOR statement. The BREAK statement allows the user to set break points within the executable program.

Whenever a line number or variable mentioned as an argument of a BREAK statement is executed, the item is printed at the user's terminal. The debug mode signature character (?) is printed, program execution is temporarily suspended, and BASIC enters debug mode. The user may then execute any of the debug statements or continue execution. Interpretation of the argument list is described under the MONITOR system (Item 17). BREAK may only be used with a program compiled in debug mode.

-BREAK Statement

General Form
-BREAK or -BREAK LN_1, LN_2-LN_3, VAR, (VAR, N_1-N_2),...

The -BREAK statement clears or directs BASIC to UN break the specified line numbers or variables. -BREAK with no argument "turns off" all previously referenced break functions. -BREAK with an argument list turns off only those variables and line numbers specified in the argument list.

The interpretation of arguments is described under the MONITOR statement (Item 17).

TRACE Statement

General Form
TRACE SN, SN-SN

The TRACE statement is used to logically trace execution of a program.

TRACE SN

Each statement number of the program is traced as executed. Each statement number is printed at the user's terminal.

-TRACE Statement

The -TRACE statement clears all references to any traces previously initiated. It is used to delete any traces currently in effect.

MONITOR Statement

General Form
MONITOR LN_1, LN_2-LN_3, VAR, (VAR, N_1-N_2), ...
Where: LN is a line number.
VAR is a simple variable or array name (i. e., ABC, DEF (3,4)).
N is an occurrence count.

The MONITOR statement applies only to statements compiled in debug mode. Line numbers and variables can be monitored during program execution. For example,

```
MONITOR 1000-1010, 257, XDOT, YDOT, ZDOT
```

causes the following:

- whenever any of the line numbers, 1000 thru 1010, inclusive, are executed, the line numbers are printed at the user's terminal.
- line number 257 is monitored whenever executed. The line number is printed at the user's terminal.
- the contents of the variables XDOT, YDOT, and ZDOT are printed at the user's terminal whenever they appear on the left hand side of an assignment statement. The value printed is that resulting after execution of the statement. If XDOT is referred to at line number 2110, the printout would be:

```
2110 XDOT = XXXXXX
```

An additional argument form for the MONITOR, statement can be illustrated with:

MONITOR (VELOCITY, 5-8)

This statement causes the variable VELOCITY to be monitored only for the 5th thru 8th time it occurs. This allows the user to monitor a variable only during significant portions of execution, thus reducing the amount of information output to the user's terminal. MONITOR may only be used with a program compiled in debug mode.

-MONITOR Statement

General Form
-MONITOR or -MONITOR LN_1, LN_2-LN_3, VAR, (VAR, N_1-N_2), ... Where: LN is a line number. VAR is a simple variable or array name. N is an occurrence count.

The -MONITOR statement clears or directs BASIC to UN monitor the specified line numbers or variables. -MONITOR with no argument "turns off" all previously referenced monitor functions. -MONITOR with an argument list turns off only those variables and line numbers specified in the argument list.

The interpretation of arguments is described under the MONITOR statement (Item 17).

PROGRAM CONTROL STATEMENTS

An interactive control language is an integral part of BASIC. These commands are used to build, compile, and execute a program.

COMPILE Statement

General Form
COMPILE or COMPILE file name

The COMPILE statement initiates compilation of either the current source program, or the file designated by file name. If the designated file is not BASIC source code, an error diagnostic is initiated.

The whole source program is compiled. Any errors during compilation are listed at the user's terminal.

EXECUTE Statement

General Form
EXECUTE or EXECUTE file name

The EXECUTE statement initiates execution of an object (or compiled) program. EXECUTE with no arguments executes the current object program in the user's working area. EXECUTE with file name executes the file designated by file name - if the designated file is in object form.

RUN Statement

General Form
RUN or RUN file name

The RUN statement is a combination of the COMPILE and EXECUTE statements. Either the current source program or the file designated by the file name, is compiled into object form and then executed. Execution begins only if no compilation errors are present.

SOURCE Statement

General Form
SOURCE file name

The SOURCE statement saves the current source program, as it exists, in a permanent file designated by "file name". The system responds with "NEW FILE" if the file name does not exist in the user's directory. The system responds with "OLD FILE" if this name already appears in the user's directory. The user responds to "NEW FILE" or "OLD FILE" by typing a carriage return (which creates a new file, or replaces an old file), or, aborting the statement by using the ESCAPE KEY.

OBJECT Statement

General Form
OBJECT file name

The OBJECT statement is identical to the SOURCE statement but saves the current object program, as it exists, in a permanent file designated by "file name". The system responds with "NEW FILE" if the file name does not exist in the user's directory. The system responds with "OLD FILE" if this name already appears in the user's directory. The user responds to "NEW FILE" or OLD FILE" by typing a carriage return (which creates a new file, or replaces an old file), or, aborting the statement using the ESCAPE KEY.

LOAD Statement

General Form
LOAD file name

The LOAD statement retrieves the permanently saved file designated by file name and places it in working storage for BASIC. The file may be in either source or object form. If in source form, the file may be listed, edited, compiled or executed. If the file is in object form, it may only be executed.

LIST Statement

General Form
LIST or LIST LN_1, LN_2-LN_3, LN_4, LN_5-LN_6, ...

The LIST statement causes the current source program to be listed as follows:

LIST — with no arguments lists the whole source program

LIST --- with arguments - lists the designated statements or range of statements. LIST, 30, 10-15, 40 lists lines 10 through 40 (inclusive), line 30, and line 40. No diagnostic or comment is made if designated line numbers are not present.

EDIT Statement

General Form
EDIT LN_1, LN_2-LN_3, LN_4, LN_5-LN_6, ...

The EDIT statement prepares the subsystem for editing or modifying one or more source statements in the current source program. For example:

EDIT 10, 20-25

Line 10 is printed at the user terminal. The user modifies or changes the statement. Following a carriage return, line 20 is printed and can be edited. Next, line 21, 22, ... thru line 25. Statements are edited one at a time until the argument list is exhausted. Missing line numbers are ignored.

DELETE Statement

General Form
DELETE LN_1, LN_2-LN_3 LN_4, LN_5-LN_6, ...

The DELETE statement causes portions of the current source program to be erased. Note that the execution of DELETE does not affect the permanently saved program unless the statement SOURCE file name is executed after the DELETE statement.

DELETE 35, 75-350, 900, 990

causes the deletion from the program of line number 35, line 75 thru 350, line 900, and line 990.

EXTRACT Statement

General Form
EXTRACT LN_1, LN_2-LN_3, LN_4, LN_5-LN_6, ...

The EXTRACT statement is the complement of the DELETE statement. EXTRACT deletes all of the current source program but the referenced line numbers. It is useful in taking portions of one program and preparing them for insertion in another program. Note that execution of EXTRACT does not affect the permanently saved program unless the statement SOURCE file name is executed after the EXTRACT statement.

EXTRACT 100-300, 500-600

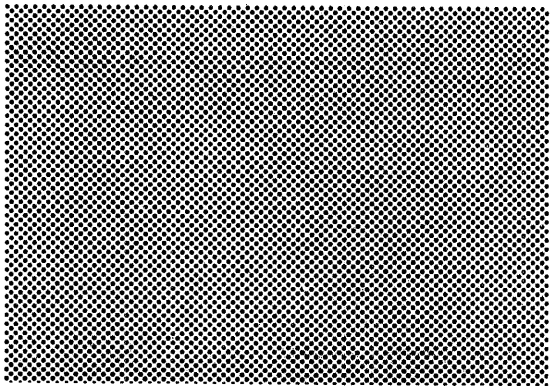
causes line numbers 100 thru 300, and line numbers 500 thru 600 to be "pulled out" of the current source program. All other line numbers are deleted.

TAPE Statement

TAPE is valid any time prior to compilation of any source statements. This causes source statements to be read from the paper tape, and processed through the initial phase of compilation.

QUIT Statement

QUIT provides an exit to the EXECUTIVE; BASIC can be continued by merely typing CONTINUE after performing EXECUTIVE functions. BASIC is terminated for the present user jobs, however, if the user should initiate a processing language (BASIC included) while EXECUTIVE is in control.



VII ...

Program Diagnostics

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

During execution, certain messages occur which do not stop execution, but inform you of irregular conditions existing in identified lines of your program. Other messages, however, point out serious errors which stop execution. Table 7-1 provides a list of compilation errors; Table 7-2 provides a list of execution errors.

TABLE 7-1. COMPILATION ERRORS

Message	Meaning
CUT PROGRAM OR DIMS	Either the program is too long, or the amount of space reserved by the DIM statements is too large, or a combination of these exists. This message can be eliminated by cutting the length of the program, reducing the size of the lists and tables, reducing the length of printed labels, or reducing the number of simple variables.
DIMENSION TOO LARGE AT (LINE #)	The size of a list or table is too large for the available storage at the line indicated.

TABLE 7-1. COMPILATION ERRORS (Cont)

Message	Meaning
EXPRESSION TO COMPLICATED IN (LINE #)	Too many operations have been attempted in a single expression. Probably too many parentheses have been used. Use two or more simpler expressions instead.
FOR'S NESTED TOO DEEPLY AT (LINE #)	Corresponding NEXT statement for preceding FOR statement must occur before another FOR statement can be used.
FOR WITHOUT NEXT IN (LINE #)	A NEXT statement is missing.
ILLEGAL CHARACTER IN (LINE #)	Use a valid character in place of an illegal character.
ILLEGAL CONSTANT IN (LINE #)	More than nine digits or incorrect form in a constant number, or a number out of bounds.
ILLEGAL FORMAT	The format of an instruction is wrong. Check especially IF-THEN's and FOR's.
ILLEGAL FORMULA IN (LINE #)	This may indicate missing parentheses, illegal variable names, missing multiply signs, illegal numbers, or many other errors.
ILLEGAL INSTRUCTION IN (LINE #)	Other than one of the fifteen legal BASIC instructions has been used in the line indicated.
ILLEGAL LINE NUMBER AFTER (LINE #)	Line number is of incorrect form, or contains more than five digits.

TABLE 7-1. COMPILATION ERRORS (Cont)

Message	Meaning
ILLEGAL LINE REFERENCE IN (LINE #)	There is some character other than a number in a transfer statement (such as a GO TO) where the line number should be.
ILLEGAL RELATION	A relational symbol other than one of the six permissible ones has been used in an IF-THEN statement.
ILLEGAL VARIABLE IN (LINE #)	An illegal variable name has been used.
INCORRECT NUMBER OF ARGUMENTS IN (LINE #)	The number of arguments when defined must equal the number of arguments when referenced.
INCORRECT NUMBER OF SUBSCRIPTS IN (LINE #)	Indicates a matrix with one subscript or a vector with two.
MISMATCHED STRING OPERATION IN (LINE #)	You have attempted to combine two strings algebraically, to compare a string and a number, or to assign a number to a string variable or vice versa.
NEXT WITHOUT FOR IN (LINE #)	A NEXT statement has been used without an accompanying FOR statement.
NO END INSTRUCTION	The program has no END statement.
NO NUMERIC DATA	There is at least one READ statement in the program calling for numeric data but no numeric data has been given.

TABLE 7-1. COMPILATION ERRORS (Cont)

Message	Meaning
NO STRING DATA	There is at least one READ statement in the program calling for string data, but no string data has been given.
SYSTEM ERROR IN (LINE #)	An error in BASIC; please report to your BPG.
*UNDEFINED LINE NUMBER (LINE#) IN (LINE #)	The line number appearing in a GOTO or IF-THEN statement does not appear as a line number in the program.
*UNDEFINED FUNCTION FN (LETTER) IN (LINE #)	A function such as FNF() has been used without appearing in a DEF statement. Check for typographical errors.
*These errors are not detected until run-time initialization	

TABLE 7-2. EXECUTION ERRORS

Message	Meaning
ABSOLUTE VALUE RAISED TO POWER IN (LINE #)	A computation of the form (-3)↑2.7 has been attempted. The system supplies (ABS(-3))↑2.7 and continues. Note: (-3)↑3 is correctly computed to give -27.
DIVISION BY ZERO IN (LINE #)	A division by zero has been attempted. The system assumes the answer is +∞ and continues running the program.
EXP TOO LARGE IN (LINE #)	The argument of an exponential function is supplied for the value of the exponential and the running is continued.

TABLE 7-2. EXECUTION ERRORS (Cont)

Message	Meaning
INPUT DATA NOT IN CORRECT FORMAT - RETYPE IT	Correct the input data.
LOG OF NEGATIVE NUMBER IN (LINE #)	The program has attempted to calculate the logarithm of a negative number. The system supplies the logarithm of the absolute value and continues.
LOG OF ZERO IN (LINE #)	The program has attempted to calculate the logarithm of 0. The system supplies $-\infty$ and continues running the program.
ON EVALUATED OUT OF RANGE IN (LINE #)	The integer part of the variable in the ON-GO TO statement is less than 1 or greater than the number of line numbers supplied by the statement.
OUT OF DATA IN (LINE #)	A READ statement for which there is no DATA has been encountered. This may mean a normal end of your program. Otherwise, it means you haven't supplied enough DATA. Execution stops.
RETURN BEFORE GOSUB IN (LINE #)	This occurs if a RETURN is encountered before a GOSUB. (The GOSUB does not require a lower statement number, but must be performed before a RETURN) Execution stops.
SQUARE ROOT OF NEGATIVE NUMBER IN (LINE #)	The program has attempted to extract the square root of a negative number. The system supplies the square root of the absolute value and continues running the program.

TABLE 7-2. EXECUTION ERRORS (Cont)

Message	Meaning
SUBSCRIPT ERROR IN (LINE #)	A subscript has been called for that lies outside the range specified in the DIM statement, or if no DIM statement applies, outside the range through 10. Execution stops.
ZERO TO A NEGATIVE POWER IN (LINE #)	A computation of the form 0↑(-1) has been attempted. The system +∞ and continues running the program.