SECTION II
Essentials of BASIC

The first section introduced the user to BASIC programming. This section describes the statements needed to write a simple BASIC program. It also describes the commands used to run a program, to edit a program, and to save and manipulate library programs.

The section begins with a description of expressions used in BASIC, and the constants, variables, functions and operators used in the formation of expressions.

Subsequent sections discuss particular features of more advanced BASIC.

The simple PRINT statement and RUN command used in Section I are used again in this section prior to the explanation of the full capabilities of PRINT and RUN.

Expressions

An expression combines constants, variables, or functions with operators in an ordered sequence. When evaluated, an expression must result in a value. An expression that, when evaluated, is converted to an integer, is called an integer expression. Constants, variables, and functions represent values; operators tell the computer the type of operation to perform on these values.

Some examples of expressions are:

(P + 5)/27	P is a variable that must have been previously assigned a value. 5 and 27 are constants. The slash is the divide operator. Parentheses group those portions of the expression evaluated first. If P = 49, it is an integer expression with the value 2.
(N-(R+5))-T	N, R, and T must all have been assigned values. + and - are the add and subtract operators. The innermost parentheses enclose the part evaluated first. If N=20, R=10, and T=5, the value of the integer expression is zero.

CONSTANTS

A constant is either numeric or it is a literal string.

Numeric Constants. A numeric constant is a positive or negative decimal number including zero. It may be written in any of the following three forms:

- As an integer — a series of digits with no decimal point.
- As a fixed point number — a series of digits with one decimal point preceding, following, or embedded within the series.
- As a floating point number — an integer or fixed point number followed by the letter E and an optionally signed integer.

Examples of Integers:

```
 1234
-70
 0
```

Examples of Fixed Point Numbers:

```
 1234.
 1234.56
-.0123
```

Floating Point Numbers. In the floating point notation, the number preceding E is a magnitude that is multiplied by some power of 10. The integer after E is the exponent, that is, it is the power of 10 by which the magnitude is multiplied.

The exponent of a floating point number is used to position the decimal point. Without this notation, describing a very large or very small number would be cumbersome:

```
1E+35 = 100000000000000000000000000000000000
1E-35 = .00000000000000000000000000000000001
```

Examples of Floating-Point Numbers:

1E+23	$= 1 \times 10^{23} = 100000000000000000000$
1.0E23	(same as above)
.001E26	(same as above)
1.02E+4	$= 1.02 \times 10^{4} = 10200.$
1.02E-4	= .000102

Within the computer, all these constants are represented as floating-point real numbers whose precision is 6 or 7 digits and whose size is between 10^{-77} and 10^{77}.

Literal Strings. A literal string consists of a sequence of characters in the ASCII character set enclosed within quotes. The quote itself is the only character excluded from the character string. By using an integer equivalent of the graphic character, even the quote may be included in a character string (see Strings, Section V).

Examples of Literal Strings:

```
"ABC"                    ""      (a null, empty, or zero length string)
"!!WHAT A DAY!!"         "  "    (a string with two blanks)
" X  Y  Z "
```

Blank spaces are significant within a string.

VARIABLES

A variable is a name to which a value is assigned. This value may be changed during program execution. A reference to the variable acts as a reference to its current value. Variables are either numeric or string.

Numeric variables are a single letter (from A to Z) or a letter immediately followed by a digit (from 0 to 9):

```
A       A0
P       P5
X       X9
```

A variable of this type always contains a numeric value that is represented in the computer by a real floating-point number. Other numeric representations can be specifically requested with the type statement (see Variable Types, Section IV). These types are integer, long floating-point, and complex.

A variable may also contain a string of characters. This type of variable is identified by a variable name consisting of a letter and $, or a letter, digit, and $:

```
A$      A0$
P$      P5$
```

The value of a string variable is always a string of characters, possibly null or zero length. String variables can be used without being declared with a DIM statement (see section V) only if the variable contains a single character.

If a variable names an array (see Arrays, Section III), it may be subscripted. When a variable is subscripted, the variable name is followed by one or two subscript values enclosed in parentheses. If there are two subscripts, they are separated by a comma. A subscript may be an integer constant or variable, or any expression that is evaluated to an integer value:

A(1)	A0(N,M)
P(1,1)	P5(Q5,N/2)
X(N+1)	X9(10,10)

A simple numeric variable and a subscripted numeric variable may have the same name with no implied relation between the two. The variable A is totally distinct from variable A(1,1).

Simple numeric variables can be used without being declared. Subscripted variables must be declared with a DIM statement (see Section III) if the array dimensions are greater than 10 rows, or 10 rows and 10 columns. The first subscript is always the row number, the second the column number. The subscript expressions must result in a value between 1 and the maximum number of rows and columns.

String arrays differ from numeric arrays in that they have only one dimension, and hence only one subscript. Also, the name of a string array and a simple string variable may not be the same (see String Arrays in Section V). Examples of subscripted string array names are:

A$(1)	A0$(N)

FUNCTIONS

A function names an operation that is performed using one or more parameter values to produce a single value result. A numeric function is identified by a three-letter name followed by one or more formal parameters enclosed in parentheses. If there is more than one, the parameters are separated by commas. The number and type of the parameters depends on the particular function. The formal parameters in the function definition are replaced by actual parameters when the function is used.

Since a function results in a single value, it can be used anywhere in an expression where a constant or variable can be used. To use a function, the function name followed by actual parameters in parentheses (known as a function call) is placed in an expression. The resulting value is used in the evaluation of the expression.

Examples of common functions:

SQR(x) where x is a numeric expression that results in a value $\geqslant 0$. When called, it returns the square root of x. For instance, if N = 2, SQR(N+2) = 2.

ABS(x) where x is any numeric expression. When called, it returns the absolute value of x. For instance, ABS(-33) = 33.

BASIC/3000 provides many built-in functions that perform common operations such as finding the sine, taking the square root, or finding the absolute value of a number. The available functions are listed in Appendix E. In addition, the user may define and name his own functions should he need to repeat a particular operation. How to write functions is described in Section VI, User-Defined Functions.

The functions described so far are numeric functions that result in a numeric value. Functions resulting in string values are also available. These are identified by a three-letter name followed by a $. String functions are described with user-defined functions in Section VI; available built-in string functions are listed in Appendix E.

OPERATORS

An operator performs a mathematical or logical operation on one or two values resulting in a single value. Generally, an operator is between two values, but there are unary operators that precede a single value. For instance, the minus sign in A - B is a binary operator that results in subtraction of the values; the minus sign in -A is a unary operator indicating that A is to be negated.

The combination of one or two operands with an operator forms an expression. The operands that appear in an expression can be constants, variables, functions, or other expressions.

Operators may be divided into types depending on the kind of operation performed. The main types are arithmetic, relational, and logical (or Boolean) operators.

The arithmetic operators are:

+	Add (or if unary, no operation)	A + B or +A
-	Subtract (or if unary, negative)	A - B or -A
*	Multiply	$A \times B$
/	Divide	$A \div B$
** or ^	Exponentiate (if ^ is used, it is changed internally to **)	A^B
MOD	Modulo; remainder from division	$A - B \times INT(A \div B)$ where INT(x) returns the largest integer $\leq x$. If A and B are positive, A MOD B is the remainder from $A \div B$.

In an expression, the arithmetic operators cause an arithmetic operation resulting in a single numeric value.

The relational operators are:

=	Equal	A = B
<	Less than	A < B
>	Greater than	A > B
<=	Less than or equal to	A ≤ B
>=	Greater than or equal to	A ≥ B
<> or #	Not equal (if # is used, it is changed internally to <>)	A ≠ B

When relational operators are evaluated in an expression they return the value 1 if the relation is found to be true, or the value 0 if the relation is false. For instance, A = B is evaluated as 1 if A and B are equal in value, as 0 if they are unequal.

Maximum and minimum operators are:

MIN	Select the lesser of two values	A MIN B
MAX	Select the greater of two values	A MAX B

These operators are evaluated as follows:

A MIN B = A if A is less than or equal to B; = B if B is less than A

A MAX B = A if A is greater than or equal to B; = B if B is greater than A

Logical or Boolean operators are:

AND	Logical "and"	A AND B
OR	Logical "or"	A OR B
NOT	Logical complement	NOT A

Like the relational operators, the evaluation of an expression using logical operators results in the value 1 if the expression is true, the value 0 if the expression is false.

Logical operators are evaluated as follows:

A AND B	= 1 (true) if A and B are both ≠ 0; = 0 (false) if A = 0 or B = 0
A OR B	= 1 (true) if A ≠ 0 or B ≠ 0; = 0 (false) if both A and B = 0
NOT A	= 1 (true) if A = 0; = 0 (false) if A ≠ 0

A string operator is available for combining two string expressions into one:

+	Concatenation	A$ + B$

The values of A$ and B$ are joined to form a single string; the characters in B$ immediately follow the last character in A$. If A$ contains "ABC" and B$ contains "DEF", then A$ + B$ = "ABCDEF" (see Strings, Section V).

EVALUATING EXPRESSIONS

An expression is evaluated by replacing each variable with its value, evaluating any function calls, and performing the operations indicated by the operators. The order in which operations is performed is determined by the hierarchy of operators:

**	(highest)
NOT	
* / MOD	
+ -	
+ (string concatenate)	
MIN MAX	
Relational (=, <, >, < =, > =, < >)	
AND	
OR	(lowest)

The operator at the highest level is performed first followed by any other operators in the hierarchy shown above. If operators are at the same level, the order is from left to right. Parentheses can be used to override this order. Operations enclosed in parentheses are performed before any operations outside the parentheses. When parentheses are nested, operations within the innermost pair are performed first.

For instance: 5 + 6*7 is evaluated as 5 + (6X7) = 47

7/14*2/5 is evaluated as ((7/14)X2)/5 = .2

If A=1, B=2, C=3, D=3.14, E=0

then:

A+B*C	is evaluated as A +(BXC) = 7
A*B+C	is evaluated as (AXB)+C = 5
A+B-C	is evaluated as (A+B)-C = 0
(A+B)*C	is evaluated as (A+B)XC = 9

A MIN B MAX C MIN D is evaluated as ((A MIN B)MAX C)MIN D = C = 3

When a unary operator immediately follows another operator of higher precedence, the unary operator assumes the same precedence as the preceding operator. For instance,

B**-B**C is evaluated as $(B^{-B})^{C}$ = 1/64 or .015625

In a relation, the relational operator determines whether the relation is equal to 1 (true) or 0 (false):

(A*B) < (A-C/3) is evaluated as 0 (false) since A*B=2 which is not less than A-C/3=0.

In a logical expression, other operators are evaluated first for values of zero (false) or non-zero (true). The logical operators determine whether the entire expression is equal to 0 (false) or 1 (true):

E AND A-C/3	is evaluated as 0 (false) since both terms in the expression are equal to zero (false).
A+B AND A*B	is evaluated as 1 (true) since both terms in the expression are different from zero (true).
A=B OR C=SIN(D)	is evaluated as 0 (false) since both expressions are false (0).
A OR E	is evaluated as 1 (true) since one term of the expression (A) is not equal to zero.
NOT E	is evaluated as 1 (true) since E=0.

If any ambiguity exists between the relational operator "=" and the assignment operator, the equal sign is treated as an assignment operator:

A=B=1	assigns 1 to both A and B.
A=1=B	assigns 1 to A if B equals 1, or 0 to A if B does not equal 1.

For rules governing the evaluation of relational expressions using strings, see Comparing Strings in Section V.

Statements

Statements essential to writing a program in BASIC are described here. Statements in general are described in Section I. It should be recalled that all statements must be preceded by a statement number and are terminated by pressing the return key. Statements are not executed until the program is executed with the RUN command.

Assignment Statement

This statement assigns a value to one or more variables. The value may be in the form of an expression, a constant, a string, or another variable of the same type.

Form

When the value of the expression is assigned to a single variable, the forms are:

variable = expression

LET variable = expression

When the same value is to be assigned to more than one variable, the forms are:

variable = variable = . . . = variable = expression

LET variable = variable = . . . = variable = expression

Several assignments can be made in one statement if they are separated by commas:

variable = expression, . . ., variable = expression

LET variable = expression, . . ., variable = expression

Note that the word *LET* is an optional part of the assignment statement.

Explanation

In this statement, the equal sign is an assignment operator. It does not indicate equality, but is a signal that the value on the right of assignment operator be assigned to the variable on the left. If any ambiguity exists between the relational operator "=" and the assignment operator, the equal sign is treated as an assignment operator.

When a variable to be assigned a value contains subscripts, these are evaluated first from left to right, then the expression is evaluated and the resulting value moved to the variable.

If a value is assigned to more than one variable, the assignment is made from right to left. For instance, in the statement A=B=C=2, first C is assigned the value 2, then B is assigned the current value of C, and finally A is assigned the value of B.

Examples

```
10 LET A=5.02
20 A=5.02
```

The variable A is assigned the value 5.02. Statements 10 and 20 have the same result.

```
30 X=Y7=Z=Z1=0
```

Each variable X, Y7, Z, and Z1 is set to zero. This is a simple method for initializing variables at the start of a program.

```
35 LET N=2
40 LET A[N]=N=9
```

First N is assigned the value 2 in line 35. In line 40 N is assigned the value 9, then the array element A(2) is assigned the value 9.

```
50 N=0
60 LET N=N+1
70 LET A[N]=N
```

Statements 50 through 70 set the array element A(1) to 1. By repeating statements 60 and 70, each array element can be set to the value of its subscript.

```
80 A=10.5,B=7.5
90 B$="ABC",C$=B$
```

Variable A is set to 10.5, then B is set to 7.5. The string variable B$ is assigned the value ABC, then C$ is assigned the value of B$ (or ABC).

```
100 C$=B$="ABC"
```

This statement has the same result as statement 90.

```
110 LET A=10.5,B=7.5,B$=C$="ABC"
```

Statement 110 has the same effect as the two statements 80 and 90.

REM Statement

This statement allows the insertion of a line of remarks in the listing of the program. The remarks do not affect program execution.

Form

REM any characters

Like other statements, REM must be preceded by a statement number. Unlike other statements, it cannot be continued on the next line.

Explanation

The remarks introduced by REM are saved as part of the BASIC program, and printed when the program is listed or punched. They are, however, ignored when the program is executed.

Remarks are easier to read if REM is followed by spaces, or a punctuation mark as in the examples.

Examples

```
10 REM: THIS IS AN EXAMPLE
20 REM  OF REM STATEMENTS.
30 REM -- ANY CHARACTERS MAY FOLLOW REM: "//**!!&&&,ETC.
40 REM...REM STATEMENTS ARE NOT EXECUTED
```

GOTO Statement

GOTO overrides the normal sequential order of statement execution by transferring control to a specified statement. The statement to which control transfers must be an existing statement in the current program.

Form

GOTO statement label

GOTO integer expression OF *statement label, statement label, . . .*

GOTO may have a single *statement label*, or may be multi-branched with more than one *statement label.*

If the multi-branch GOTO is used, the value of the *integer expression* determines the label in the list to which control transfers.

Explanation

If the GOTO transfers to a statement that cannot be executed (such as REM or DIM), control passes to the next sequential statement after that statement. GOTO cannot transfer into or out of a function definition (see Section VI). If it should transfer to the DEF statement, control passes to the line following the function definition.

The labels in a multi-branch GOTO are selected by numbering them sequentially starting with 1, such that the first label is selected if the value of the expression is 1, the second label if the expression equals 2, and so forth. If the value of the expression is less than 1 or greater than the number of labels in the list, then the GOTO is ignored and control transfers to the statement immediately following GOTO.

Examples

The example below shows a simple GOTO in line 200 and a multi-branch GOTO in line 600.

```
100 LET I=0
200 GOTO 600
300 PRINT I
400 REM   THE VALUE OF I IS ZERO
500 LET I=I+1
600 GOTO I+1 OF 300,500,800
700 REM   THE FINAL VALUE OF I IS 2
800 PRINT I

>RUN
 0
 2
```

When run, the program prints the initial value of I and the final value of I.

GOSUB/RETURN Statements

GOSUB transfers control to the beginning of a simple subroutine. A subroutine consists of a collection of statements that may be performed from more than one location in the program. In a simple subroutine, there is no explicit indication in the program as to which statements constitute the subroutine. A RETURN statement in the subroutine returns control to the statement following the GOSUB statement.

Form

GOSUB statement label

GOSUB integer expression OF *statement label, statement label, . . .*

RETURN

GOSUB may have a single *statement label,* or may be multi-branched with more than one *statement label.* In a multi-branch GOSUB, the particular label to which control transfers is determined by the value of the *integer expression.* The RETURN statement consists simply of the word RETURN.

Explanation

A single-branch GOSUB transfers control to the statement indicated by the label. A multi-branch GOSUB transfers to the statement label determined by the value of the integer expression. As in a multi-branch GOTO, if the value of the expression is less than 1 or greater than the length of the list, no transfer takes place. A GOSUB must not transfer into or out of a function definition (see Section VI).

When the sequence of control within the subroutine reaches a RETURN statement, control returns to the statement following the GOSUB statement.

Within a subroutine, another subroutine can be called. This is known as nesting. When a RETURN is executed, control transfers back to the statement following the last GOSUB executed. Up to ten GOSUB statements can occur without an intervening RETURN; more than this causes a terminating error.

Examples

In the first example, line 20 contains a simple GOSUB statement; the subroutine is in lines 50 through 70, with RETURN in line 70.

```
 10 LET B=90
 20 GOSUB 50
 30 PRINT "SINE OF B IS ";A
 40 GOTO 80
 50 REM: THIS IS THE START OF THE SUBROUTINE
 60 LET A=SIN(B)
 70 RETURN
 80 REM: PROGRAM CONTINUES WITH NEXT STATEMENT
>RUN
SINE OF B IS  .893992
```

The GOSUB statement can follow the subroutine to which it transfers as in the example below.

```
 10 LET B=90
 20 GOTO 100
 30 REM: THIS IS START OF SUBROUTINE
 40 LET A=SIN(B)
 50 RETURN
 60 REM: OTHER STATEMENTS CAN APPEAR HERE
 70 REM: THEY WILL NOT BE EXECUTED
 80 A=24,B=50
 90 PRINT A;B
100 GOSUB 30
110 PRINT A
120 REM: A SHOULD EQUAL .893992
130 PRINT B
140 REM: B SHOULD EQUAL 90
>RUN
 .893992
 90
```

This example shows a multi-branch GOSUB in line 20. The third subroutine executed has a nested subroutine. An IF. . .THEN statement is used in the example; should its function not be clear, see Conditional Statements below in this section.

```
 10 A=0
 20 GOSUB A+1 OF 100,150,200
 30 LET A=A+1
 40 IF A<3 THEN GOTO 20
 50 GOTO 300
 60 REM:  STATEMENT 50 BRANCHES AROUND ALL THE SUBROUTINES
100 REM:  FIRST SUBROUTINE IN MULTIBRANCH GOSUB
110 LET X=SQR(A+25)
120 PRINT "X = ";X
130 RETURN
150 REM:  SECOND SUBROUTINE IN MULTIBRANCH GOSUB
160 LET Y=COS(X)
170 PRINT "Y = COSINE X = ";Y
180 RETURN
200 REM:  THIRD SUBROUTINE IN MULTIBRANCH GOSUB
210 REM:  IT CONTAINS A NESTED SUBROUTINE
220 LET Y=Y+X
225 PRINT "Y + X = ";Y
230 GOSUB 260
240 RETURN
250 REM:  STATEMENT 240 RETURNS CONTROL TO STATEMENT 30
260 REM:  FIRST STATEMENT IN NESTED SUBROUTINE
270 B=SIN(Y)
280 PRINT "SINE Y = ";B
290 RETURN
295 REM:  STATEMENT 290 RETURNS CONTROL TO STATEMENT 240
300 REM:  PROGRAM CONTINUES WITH NEXT STATEMENT
>RUN
X =  5
Y = COSINE X =  .283663
Y + X =  5.28366
SINE Y = -.841213
```

END/STOP Statements

The END and STOP statements are used to terminate execution of a program. Either may be used, neither is required. An END is assumed following the last line entered in the current program.

Form

END

STOP

The END statement consists of the word END; the STOP statement of the word STOP.

Explanation

Both END and STOP terminate the program run. END has a different function from STOP only when programs are segmented (see Section X, Segmentation). When END is executed in a program segment that has been called by another program with INVOKE, control returns to the statement after INVOKE.

Whenever STOP is used, the program terminates. STOP in a program called with INVOKE terminates all program execution, including any suspended programs.

Examples

These three programs are effectively the same:

```
 10 LET A=2,B=3
 20 C=A**-A**B
 30 PRINT C
>RUN
 .015625
```

```
 10 LET A=2,B=3
 20 C=A**-A**B
 30 PRINT C
 40 END
>RUN
 .015625
```

```
 10 LET A=2,B=3
 20 C=A**-A**B
 30 PRINT C
 40 STOP
>RUN
 .015625
```

When sequence is direct and the last statement in the current program is the last statement to be executed, END or STOP are optional. They have a use, however, when sequence is not direct and the last statement in the program is not the last statement to be executed:

```
100 LET A=2
120 GOSUB 140
130 END
140 LET B=A+1
150 X=A**(B**A)
160 PRINT X
170 RETURN
>RUN
 512
```

The subroutine at line 140 follows the END statement.

```
 10 LET A=2
 20 X=A**2+A
 30 PRINT X
 40 IF X<100 THEN GOTO 80
 50 PRINT "X= ";X
 60 PRINT "A= ";A
 70 STOP
 80 A=A+1
 90 GOTO 20
>RUN
 6
 12
 20
 30
 42
 56
 72
 90
 110
X=  110
A=  10
```

The STOP statement at line 70 is skipped until the value of X is equal to or exceeds 100.

Looping Statements

The looping statements FOR and NEXT allow repetition of a group of statements. The FOR statement precedes the statements to be repeated, and the NEXT statement directly follows them. The number of times the statements are repeated is determined by the value of a simple numeric variable specified in the FOR statement.

Form

FOR variable = expression TO expression

FOR variable = expression TO expression STEP expression

The *variable* is initially set to the value resulting from the *expression* after the equal sign. When the value of the *variable* passes the value of the *expression* following TO, the looping stops. If STEP is specified, the *variable* is incremented by the value resulting from the STEP *expression* each time the group of statements is repeated. This value can be positive or negative, but should not be zero. If a STEP *expression* is not specified, the variable is incremented by 1.

The NEXT statement terminates the loop:

NEXT variable

The *variable* following NEXT must be the same as the *variable* after the corresponding FOR.

Explanation

When FOR is executed, the variable is assigned an initial value resulting from the expression after the equal sign, and the final value and any step value are evaluated. Then the following steps occur:

1. The value of the FOR variable is compared to the final value; if it exceeds the final value (or is less when the STEP value is negative), control skips to the statement following NEXT.
2. All statements between the FOR statement and the NEXT statement are executed.
3. The FOR variable is incremented by 1, or if specified, by the STEP value.
4. Return to step 1.

NOTE: Unless specified with a variable type statement, the values of the variables used to index a FOR loop are assigned as real by default. Round-off errors can increase or decrease the number of steps when non-integer step sizes are used.

The user should not execute the statements in a FOR loop except through a FOR statement. Transferring control into the middle of a loop can produce undesirable results.

FOR loops can be nested if one FOR loop is completely contained within another. They must not overlap.

Examples

Each time the FOR statement executes, the user inputs a value for R and the area of a circle with that radius is computed and printed:

```
  10 FOR A=1 TO 5
  20   INPUT R
  30   PRINT "AREA OF CIRCLE WITH RADIUS ";R;" IS ";3.14*R**2
  40 NEXT A
>RUN
?1
AREA OF CIRCLE WITH RADIUS  1     IS  3.14
?2
AREA OF CIRCLE WITH RADIUS  2     IS  12.56
?4
AREA OF CIRCLE WITH RADIUS  4     IS  50.24
?8
AREA OF CIRCLE WITH RADIUS  8     IS  200.96
?16
AREA OF CIRCLE WITH RADIUS  16    IS  803.84
```

The FOR loop executes six times, decreasing the value of X by 1 each time:

```
  10 FOR X=0 TO -5 STEP -1
  20   PRINT X-5
  30 NEXT X
>RUN
-5
-6
-7
-8
-9
-10
```

The first X elements of the array P(N) are assigned values. When N = X, the loop terminates. In this case, the value of X is input as 3:

```
 10 INPUT X
 20 FOR N=1 TO X
 30   LET P[N]=N+1
 40   PRINT P[N]
 50 NEXT N
>RUN
?3
 2
 3
 4
```

The examples below show legal and illegal nesting. A diagnostic is printed when an attempt is made to run the second example:

```
10 REM..THIS EXAMPLE IS LEGAL
20 FOR A=1 TO 10
30   FOR B=1 TO 5
40     LET X[A,B]=0
50   NEXT B
60 NEXT A
```

```
 10 REM.. THIS EXAMPLE IS ILLEGAL
 20 FOR A=1 TO 10
 30   FOR B=1 TO 5
 40     LET X[A,B]=0
 50   NEXT A
 60 NEXT B
>RUN
'FOR' - 'NEXT' VARIABLES DON'T MATCH IN LINE 50
```

Conditional Statements

Conditional statements are used to test for specific conditions and specify program action depending on the test result. The condition tested is a numeric expression that is considered true if the value is not zero, false if the value is zero. Conditional statements are always introduced by an IF statement; an ELSE statement may follow the IF statement. Both IF and ELSE statements may be followed by a series of statements enclosed by DO and DOEND.

Form

IF expression THEN statement label

IF expression THEN statement

IF expression THEN DO
statement
.
.
.
DOEND

An IF. . .THEN statement can be followed by an ELSE statement to specify action in case the value of the *expression* is false. Like IF, ELSE can be followed by a *statement*, a *statement label*, or a series of statements enclosed by DO. . .DOEND.

ELSE statement label

ELSE statement

ELSE DO
statement
.
.
.
DOEND

ELSE statements never appear in a program unless preceded by an IF. . .THEN statement. An ELSE statement must immediately follow an IF. . .THEN statement or the DOEND statement corresponding to an IF. . .THEN DO statement; no intervening statements (including REM) are permitted. DO. . .DOEND statements may follow only an IF. . .THEN or an ELSE statement.

The four diagrams below show all possible combinations of conditional statements. Items enclosed by [ ] are optional; one of the items enclosed by { } must be chosen. Statements immediately following THEN and ELSE are not labeled; all other statements must be labeled.

- *label IF expression THEN* $\left\{\begin{matrix} label \\ statement \end{matrix}\right\}$

  $\left[\text{label ELSE } \left\{\begin{matrix} label \\ statement \end{matrix}\right\}\right]$

- *label IF expression THEN DO*

  label statement

  ⋮

  label DOEND

  $\left[\text{label ELSE } \left\{\begin{matrix} label \\ statement \end{matrix}\right\}\right]$

- *label IF expression THEN* $\left\{\begin{matrix} label \\ statement \end{matrix}\right\}$

  label ELSE DO

  label statement

  ⋮

  label DOEND

- *label IF expression THEN DO*

  label statement

  ⋮

  label DOEND

  label ELSE DO

  label statement

  ⋮

  label DOEND

Explanation

The expression following IF is evaluated, and if true the program transfers control to the label following THEN or executes the statement following THEN. If DO follows THEN, the program executes the series of labeled statements terminated by DOEND. The program then continues. If the expression is false, control transfers immediately to the next statement or to the statement following DOEND if THEN DO was specified.

When an ELSE statement follows the IF. . .THEN statement, it determines the specific action should the IF expression be false. When the expression is true, the ELSE statement or the group of ELSE statements enclosed by DO. . .DOEND is skipped, and the program continues with the next statement after ELSE or DOEND.

A FOR statement can be specified in a DO. . .DOEND group; if so, the corresponding NEXT must be within the same DO. . .DOEND group. (See FOR. . .NEXT statement description in this section.)

IF statements are nested when an IF statement occurs within the DO. . .DOEND group of another IF statement. In such a case, each ELSE is matched with the closest preceding IF that is not itself part of another DO. . .DOEND group.

Examples

The various types of IF statement are illustrated with the following examples:

```
10 IF A=B THEN 30
20 LET A=B
30 PRINT A,B
```

If A equals B, the program skips to line 30, otherwise, it sets A equal B in line 20 and continues. In either case, line 30 is executed.

```
10 IF A=B THEN PRINT B
20 ELSE PRINT A,B
```

If A equals B, the value of B is printed, otherwise, both values are printed. The program then continues.

```
10 IF A=B THEN 100
20 ELSE 200
```

Program control transfers to line 100 if A equals B, to line 200 if not.

```
10 IF A=B THEN GOTO 100
20 ELSE GOTO 200
```

These two statements are identical in effect to the preceding two statements.

```
 10 IF A<100 THEN A=A+5
 20 ELSE DO
 30   LET X=A
 40   GOTO 100
 50 DOEND
 60 GOTO 10
100 PRINT X
```

If A is less than 100, it is increased by 5 and control skips to line 60 where control is returned to line 10. When A is equal to or greater than 100, X is set equal to A and control skips to line 100.

```
  5 INPUT A
 10 IF A<100 THEN DO
 20   A=A+1
 30   GOTO 200
 40 DOEND
 50 ELSE DO
 60   X=A
 70   A=0
 80   GOSUB 850
 90 DOEND
100 PRINT "A>=100"
120 END
200 PRINT "A=";A
210 END
850 PRINT "X=";X
851 PRINT "A=";A
852 RETURN
```

If A is less than 100, it is increased by 1 and control goes to line 200. If A is equal to or greater than 100, X is set equal to A, A is set to zero and the subroutine at line 850 is executed. The subroutine returns control to line 100.

If a value less than 100 is input for A, line 200 is executed and the program ends:

```
>RUN
?75
A= 76
```

If a value greater than 100 is input for A, the subroutine is executed, then line 100 is executed and the program terminates:

```
>RUN
?150
X= 150
A= 0
A>=100
```

The examples below illustrate nested IF. . .THEN statements.

```
 10 INPUT A,B,C
 20 IF (A+10)=(B+5) THEN DO
 30   A=B
 40   IF A>C THEN A=C
 50   ELSE C=B
 60 DOEND
 70 PRINT A,B,C
>RUN
?5,10,15
 10              10              10
```

With the particular values input, the first IF is true and the second IF is false. As a result both A and C are set equal to B.

```
 10 INPUT A,B,C
 20 IF A>B THEN DO
 30   IF B>C THEN DO
 40     IF C=10 THEN DO
 50       C=C+1
 60       GOTO 200
 70     DOEND
 80     ELSE GOTO 220
 90   DOEND
100   ELSE DO
110     IF C=10 THEN B=C+A
120     ELSE C=B-A
130     GOTO 180
140   DOEND
150 DOEND
160 PRINT "A<=B,A=";A
170 GOTO 230
180 PRINT "A>B,B<=C,B=";B
190 GOTO 230
200 PRINT "A>B>C,C=10"
210 GOTO 230
220 PRINT "A>B>C,C<>10,C=";C
230 END
>RUN
?10,15,20
A<B,A= 10

>RUN
?15,5,10
A>B,B<C,B= 25

>RUN
?20,15,5
A>B>C,C<>10,C= 5
```

So that nested IF statements may be easier to follow, the LIST command indents them as shown in these examples.

PRINT Statement

PRINT causes data to be output at the terminal. The data to be output is specified in a print list following PRINT.

Form

PRINT

PRINT print list

The *print list* consists of items separated by commas or semicolons. The list may be followed by a comma or a semicolon. If the list is omitted, PRINT causes a skip to the next line. Items in the list may be numeric or string expressions, special print functions for tabbing or spacing, or FOR loops to provide repeated output. The form of the FOR loop is:

(FOR statement, print list)

where the *print list* contains any items allowed in the PRINT statement list including other FOR loops. The FOR statement is described earlier in this section under the heading Looping Statements.

Explanation

The contents of the print list is printed. If there is more than one item in the print list, commas or semicolons must separate the items. The choice of a comma or semicolon affects the output format.

The output line is divided into consecutive fields, each of 15 characters except possibly the last. For example, on a terminal with default print length of 72 characters, there will be four fields of 15 characters and one of 12 characters. When a comma separates items, each item is printed starting at the beginning of a field. When a semicolon separates items, each item is printed immediately following the preceding item. In either case, if there is not enough room left in the line to print the entire item, printing of the item begins on the next line. The length of the print line can be changed by using the MARGIN statement (Section VIII).

The separator between items can be omitted if one or both of the items is a quoted string. In this case, a semicolon is inserted automatically.

A carriage return and linefeed are output after PRINT has executed, unless the output list is terminated by a comma or semicolon. In this case, the next PRINT statement begins on the same line.

If an expression appears in the print list, it is evaluated and the result is printed. Any variable must have been assigned a value before it is printed. Each character between quotes in a string constant is printed, excluding quotes.

If a FOR loop is included in the print list, each item in the print list associated with the FOR statement is printed once for each time the FOR loop is executed.

Numeric values are left justified in a field whose width is determined by the magnitude of the number. The smallest field is six characters. Numeric output format is discussed in detail below.

For the printing of data according to a customized format, see the PRINT USING and PRINT # USING statements described in Section IX.

Examples

When items are separated by commas, they are printed in consecutive fields per line; separated by semicolons, they directly follow one another. In the example below, the items are numeric, so each item is assigned a minimum of six characters.

```
 10 LET A=B=C=D=E=15
 20 LET A1=B1=C1=D1=E1=20
 30 PRINT A,B,C1,C
 40 PRINT A;B;C1;C;D;E;A1;D1;E1
 50 PRINT A,B;C,D
>RUN
 15             15             20             15
 15    15    20    15    15    15    20    20    20
 15             15    15       15
```

In the example below, a DIM statement is used to specify the number of characters in each string; if omitted, the strings are assumed to have only one character.

```
 10 DIM B$[3],C$[3]
 20 C$=B$="ABC"
 30 PRINT B$,C$
>RUN
ABC            ABC
```

In the example below, the first PRINT statement evaluates and then prints three expressions. The second PRINT skips a line. The third and fourth PRINT statements combine a string constant with a numeric expression. No fields are used in the print line for string constants unless a comma appears as separator. The fourth PRINT statement prints output on the same line as the third because the third statement is terminated by a comma.

```
 10 LET A=B=C=D=E=15
 20 LET A1=B1=C1=D1=E1=20
 30 PRINT A*B,B/C/D1+30,A+B
 40 PRINT
 50 PRINT "A*B =";A*B,
 60 PRINT "THE SUM OF A AND B IS";A+B
>RUN
 225            30.05          30

A*B = 225      THE SUM OF A AND B IS 30
```

A FOR statement can be specified in a print list with its own print list, all included within parentheses:

```
 10 FOR I=1 TO 3
 20   INPUT R
 30   A[I]=3.14*R**2
 40 NEXT I
 50 PRINT (FOR I=1 TO 3,A[I])
>RUN
?2
?3
?4
 12.56          28.26          50.24
```

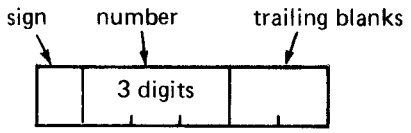
Note that NEXT is not needed when the FOR statement is included in a print list.

NUMERIC OUTPUT FORMATS

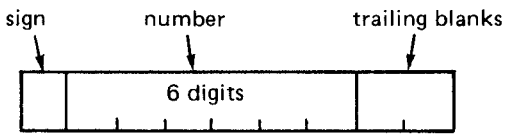
Numeric quantities are left justified in a field whose width is determined by the magnitude of the item. The width includes a position at the left of the number for a possible sign and at least one position to the right containing blanks. The width is always a multiple of three; the minimum width is six characters.

Integers

An integer with a magnitude less than 1000 requires a field width of six characters:



An integer with a magnitude between 1000 and 999999 inclusive requires a field width of nine characters:



Examples of integers:

The integers below are less than 1000 and greater than -1000:

```
  10 PRINT 1;999;30;-300;+295
>RUN
 1      999    30    -300    295
```

These integers are between 1000 and 999999 or between -1000 and -999999:

```
  10 PRINT 1000;+32751;-999999;45678
>RUN
 1000      32751    -999999    45678
```

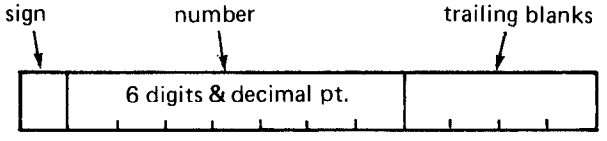
These integers are mixed in magnitude, but none are greater than 999999 or less than -999999:

```
  10 PRINT 1;1000;999;+32751;20;-999999;-300;45678;+296;5000
>RUN
 1      1000      999    32751     20    -999999  -300    45678   296
 5000
```

If an integer has a negative sign it is printed; a positive sign is not printed.

Fixed-Point Numbers

A fixed point number requires a field width of 12 positions. If the magnitude of the number is greater than or equal to .09999995 and less than 999999.5, or is less than .1 but can be printed with six significant digits, the number is printed as a fixed-point number with a sign. Trailing zeros are not printed, but a trailing decimal point is printed to show the number is not exact. The number is left-justified in the field with trailing blanks. The sign is printed only if it is negative.

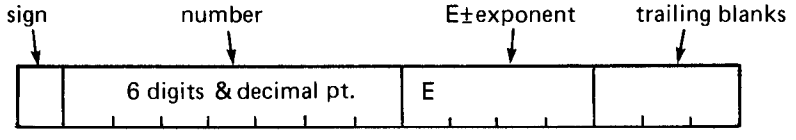


Examples of fixed-point numbers:

```
 10 PRINT 999999.4;.09999996;.000044
>RUN
 999999.      .1            .000044
```

Floating-Point Numbers

Any number, integer or fixed-point, with a magnitude greater than the magnitude of the numbers presented above, is printed as a floating-point number using a total field width of 15 positions:



Examples of floating-point numbers:

```
 10 PRINT 2345678;.0000044
>RUN
 2.34568E+06     4.40000E-06

 10 PRINT 23456789;.00000044
>RUN
 2.34568E+07     4.40000E-07

 10 PRINT .00003943;.0000257895
>RUN
 3.94300E-05     2.57895E-05
```

PRINT FUNCTIONS

These print functions may be included in a PRINT statement print list. A comma after any print function is treated as a semicolon.

TAB Function

The form of the tabulation function is:

TAB(integer expression)

The print position is moved to the column specified by the *integer expression*. Print positions are numbered from 0 to 71. If the print position must be moved to the left because the integer expression is less than the current position, nothing is done. If the expression is greater than 71, the print position is moved to the beginning of the next line.

SPA Function

The form of the spacing function is:

SPA(integer expression)

Blanks are printed for the number of spaces indicated by the *integer expression.* Nothing occurs when the expression is zero or negative. If the number of spaces will not fit on the current line, or the expression exceeds 71, a carriage return and line feed is generated.

The limit of 71 on TAB and SPA expressions does not apply to PRINT USING (see Section IX).

LIN Function

The form of the line skip function is:

LIN(integer expression)

The terminal performs a carriage return and as many line feeds as are specified in the expression. If the value is negative, the absolute value of the expression is used for the number of line feeds; no carriage return is generated. Normally, a carriage return and one line feed is performed at the end of a PRINT statement unless there is a trailing comma or semicolon.

CTL Function

The form of the carriage control function is:

CTL(integer expression)

All items preceding the CTL function in a PRINT statement are printed immediately, using the integer expression as the carriage control code. This function is effective only for the particular print statement in which it occurs and has no effect on any other statement. This function is useful when the output device is a line printer. The carriage control codes are listed below.

Carriage Control Codes

Decimal Code	Carriage Action
32	Single-space
43	Carriage return, no line feed
48	Double-space
49	Page eject (form feed)
64	Post-spacing
65	Pre-spacing
66	Single-space, with auto page eject (60 lines/pg)
67	Single-space, without auto page eject (66 lines/pg)
128+nn	Space nn lines (no automatic page eject). nn=1 thru 63 (i.e., codes 129 thru 191).
192	Page eject (*ftc #1)
193	Skip to bottom of form (*ftc #2)
194	Single-spacing, with auto page eject (*ftc #3)
195	Single-space on next odd-numbered line, with auto page eject (*ftc #4)
196	Triple-space, with auto page eject (*ftc #5)
197	Space 1/2 page, with auto page eject (*ftc #6)
198	Space 1/4 page, with auto page eject (*ftc #7)
199	Space 1/6 page, with auto page eject (*ftc #8)
256	Post-spacing
257	Pre-spacing
258	Single-space, with auto page eject (60 lines/pg)
259	Single-space, without auto page eject (66 lines/pg)
	*Format Tape Channel number

Examples of Print Functions

The TAB, SPA, LIN and CTL functions are illustrated below:

```
 10 PRINT TAB(8);" TITLE:PRINT HEADING";SPA(10);"SUMMARY REPORT";
 20 PRINT LIN(3);" DETAIL LINES"
>RUN
        TITLE:PRINT HEADING          SUMMARY REPORT


 DETAIL LINES
```

The LIN function can generally be used to provide double or triple spacing, to suppress spacing, or to provide a line feed. For instance,

Double Space	LIN(2)
Suppress Spacing	LIN(0)
Line Feed only	LIN(*-integer expression*)

```
 10 PRINT "ABC";LIN(-1);"DEF";LIN(2);"GHI"
>RUN
ABC
   DEF

GHI
```

Some frequently used carriage control characters are:

Double Space	CTL(48)
Page Eject	CTL(49)
Suppress Spacing	CTL(43)

The decimal numbers associated with the carriage control characters are used as the integer expression in the CTL function. To illustrate:

```
 10 LET P=1,X=500
 20 PRINT CTL(49),"PAGE NO";P
 30 PRINT CTL(48),"DETAIL LINE"
 40 PRINT TAB(15),X,CTL(43);
 50 PRINT TAB(10),"X=";
>RUN
```

After ejecting to the top of a new page, the print items are output as:

```
 PAGE NO 1


 DETAIL LINE
          X=    500
```

In the following example, the CTL function causes a double space between "LINE 1" and "LINE 2", but has no effect on statement 20:

```
:BASIC
>HP32101B.00.08(4WD)   BASIC   (C)HEWLETT-PACKARD CO 1976
>10 PRINT "LINE 1",CTL(48),"LINE 2"
>20 PRINT "LINE 3"
>RUN
LINE 1

LINE 2,
LINE 3

>EXIT

 END OF SUBSYSTEM
```

The effect of the CTL function in the next example is immediate at it's location within the PRINT statement 200. It has no effect at the end of that statement where a normal linefeed and carriage control occurs.

```
:BASIC
>HP32101B.00.08(4WD)   BASIC   (C)HEWLETT-PACKARD CO 1976
>100 FOR I=1 TO 2
>200 PRINT "ABCD",CTL(130),"EFGH"
>300 NEXT I
>RUN
ABCD

EFGH
ABCD

EFGH

>EXIT

 END OF SUBSYSTEM
```

READ/DATA/RESTORE Statements

Together, the READ, DATA, and RESTORE statements provide a means to input data to a BASIC/3000 program. The READ statement reads data specified in DATA statements into variables specified in the READ statement. RESTORE allows the same data to be read again.

Form

READ item list

The items in the *item list* are either variables or FOR loops. Items are separated by commas. A FOR loop has the form:

(FOR statement, item list)

where the *item list* contains variables or FOR loops separated by commas.

DATA constant, constant,. . .

The *constants* are either numeric or string. Constants in the DATA statement are assigned to variables in the READ statement according to their order: the first constant to the first variable, the second to the second and so forth.

RESTORE

RESTORE label

The *label* identifies a DATA statement.

Explanation

When a READ statement is executed, each variable is assigned a new value from the constant list in a DATA statement. RESTORE allows the first constant to be assigned again when READ is next executed or, if a label is specified, the first constant in the specified DATA statement.

More than one DATA statement can be specified. All the constants in the combined DATA statements comprise a data list. The list starts with the DATA statement having the lowest statement label and continues to the statement with the highest label. DATA statements can be anywhere in the program; they need not precede the READ statement, nor need they be consecutive.

If a variable is numeric, the next item in the data list must be numeric; if a variable is a string, the next item in the data list must be a string constant. It is possible to determine the type of the next item with the TYP function (see Section VIII).

If the READ statement contains a FOR statement, the items following the FOR statement within parentheses are assigned values once for each time the FOR statement is executed. The FOR variable can be used in the item list, as can further FOR statements.

A pointer is kept in the data list showing which constant is the next to be assigned to a variable. This pointer starts at the first DATA statement and is advanced consecutively through the data list as constants are assigned. The RESTORE statement can be used to access data constants in a non-serial manner by specifying a particular DATA statement to which the pointer is to be moved.

When the RESTORE statement specifies a label, the pointer is moved to the first constant in the specified statement. If the statement is not a DATA statement, the pointer is moved to the first following DATA statement. When no label is specified, the pointer is restored to the first constant of the first DATA statement in the program.

Examples

The data in statement 10 is read in statement 20 and printed in statement 30:

```
  10 DATA 3,5,7
  20 READ A,B,C
  30 PRINT A,B,C
>RUN
 3             5             7
```

Note the use of RESTORE in this example. It permits the second READ to read the same data into a second set of variables:

```
   5 DIM A$[3],B$[3]
  10 DATA 3,5,7
  20 READ A,B,C
  30 READ A$,B$
  40 DATA "ABC","DEF"
  50 RESTORE
  60 READ D,E,F
  70 PRINT A$+B$,A;B;C; D;E; F
>RUN
ABCDEF          3     5     7     3     5     7
```

In the following examples, the data from three DATA statements is read into an 8-element array variable and a simple variable. The same data is then restored and read into three simple variables.

```
 10 DATA 3,5,7
 20 DATA 9,11,13
 30 DATA 15,17,19
 40 READ (FOR I=1 TO 8,C[I]),D
 50 PRINT (FOR I=1 TO 8,C[I]),D
>RUN
 3             5             7             9             11
 13            15            17            19
```

```
 10 DATA 3,5,7
 20 DATA 9,11,13
 30 DATA 15,17,19
 40 READ (FOR I=1 TO 8,C[I]),D
 50 PRINT (FOR I=1 TO 8,C[I]),D
 60 RESTORE
 70 READ A
 80 RESTORE 20
 90 READ B
100 RESTORE 30
110 READ C
120 PRINT A,B,C
>RUN
 3             5             7             9             11
 13            15            17            19
 3             9             15
```

INPUT Statement

The INPUT statement allows the user to input data to his program from the terminal. INPUT has options that allow the user to save excess input and to print prompting strings before input. FOR loops may be included in the item list associated with INPUT.

Form

INPUT

INPUT item list

The items in the *item list* may be variables, string constants, or FOR loops. Items are separated by commas. FOR loops have the form:

(FOR statement, item list)

where the *item list* contains variables or FOR loops separated by commas.

A colon (:) may precede or follow the INPUT *item list*. When a colon follows the list, excess input is saved in a buffer; when a colon precedes the list, input is assigned from the buffer before it is requested from the user at the terminal.

An INPUT statement with no *item list* clears the input buffer; INPUT followed only by a colon fills the buffer.

Explanation

When an INPUT statement is executed, a question mark (?) is printed at the terminal and the program waits for the user to type his input. The input is in the form of constants separated by commas. If an insufficient number of constants is typed, the program responds with two question marks (??). This requests the user to input more constants. The type of data item, numeric or string, must match the type of variable it is destined for.

Like the READ and PRINT statements, the INPUT statement can include any number of FOR loops. Each time a FOR statement is executed, the user inputs a constant to match the variables in the item list associated with the FOR statement.

Numeric Constants. Numeric constants always begin with the first non-blank character preceding the comma or the end of the line.

String Constants. A string may be unquoted, in which case it begins with the first non-blank character and ends with the last non-blank character in the line. It may not contain quotation marks. A string may also be quoted, in which case it is delimited on each side by quotes and is followed either by a comma or the end of the line.

The INPUT statement can be requested to print a string constant instead of a question mark by placing the string constant immediately before a variable. When the value for the variable is needed, the string is printed instead of the usual question mark. Any number of these request strings can be included in the variable list.

Examples

```
  10 DIM C$[25]
  20 INPUT A,B,C$
  30 X=A*B**2
  40 PRINT C$;X
>RUN
?2,5,"X=A TIMES B SQUARED,  X="
X=A TIMES B SQUARED,  X= 50

  10 INPUT "INPUT VALUE OF RADIUS  ",R
  20 X=3.14*R**2
  30 PRINT "AREA OF X =",X
>RUN
INPUT VALUE OF RADIUS  25
AREA OF X =     1962.5
```

Note that a series of strings on one line separated by commas will be recognized as a single string constant unless each (except the last) is enclosed in quotes. See the following example:

```
 10 DIM A$[10],B$[10],C$[10]
 20 INPUT "THREE NAMES?",A$,B$,C$
 30 PRINT A$,B$,C$
>RUN
 THREE NAMES?PAUL,PETE,DIX
 ??"MARY","JOHN"
 PAUL,PETE,      MARY            JOHN

>RUN
 THREE NAMES?PAUL
 ??PETE
 ??DIX
 PAUL            PETE            DIX

>RUN
 THREE NAMES?"PAUL","PETE",DIX
 PAUL            PETE            DIX
```

This example illustrates the various prompts for input:

```
  10 INPUT A,"NUMBER?",B,C
  20 PRINT A,B,C
>RUN
?15
NUMBER?63.5
???
 15                63.5              7
```

If all input values are entered at one time, only the first prompt is used:

```
  10 INPUT A,"NUMBER?",B,C
  20 PRINT A,B,C
>RUN
?15,63.5,7
 15             63.5           7
```

The examples below illustrate FOR loops in the INPUT item list:

```
  10 INPUT (FOR I=1 TO 5 STEP 2,A[I])
  20 PRINT (FOR I=1 TO 5 STEP 2,A[I])
>RUN
?1,3,5
 1              3              5


  10 INPUT N,(FOR K=1 TO N,"WHAT'S NEXT?",B[K])
  20 PRINT (FOR K=1 TO N,B[K])
>RUN
?3
WHAT'S NEXT?1
WHAT'S NEXT?2
WHAT'S NEXT?3
 1              2              3


  10 INPUT N,(FOR S1=1 TO N,(FOR I=1 TO N,C[S1,I]))
  20 PRINT (FOR S1=1 TO N,(FOR I=1 TO N,C[S1,I]))
>RUN
?2
??1,2,3,4
 1              2              3              4
```

The example below illustrates the use of the colon (:) to save input in the buffer, and to assign input from the buffer. A colon following the input list saves the buffer; a colon preceding the input list assigns values from the buffer.

In this example, four input values are placed in the buffer. However, following line 20 the buffer is cleared because there is no colon after E. Another value must be input for F.

```
  10 INPUT A,B:
  20 INPUT :E
  30 INPUT :F
  40 PRINT A,B,E,F
>RUN
?1,2,3,4
??9
 1             2             3             9
```

By putting a colon after E as well as before it, the entire buffer is saved:

```
  10 INPUT A,B:
  20 INPUT :E:
  30 INPUT :F
  40 PRINT A,B,E,F
>RUN
?1,2,3,4
 1             2             3             4
```

BUF FUNCTION

The BUF function is used in conjunction with INPUT to determine the type of the next data item in the buffer. The form is:

BUF(X)

The parameter X has no meaning; any expression can replace X as the actual parameter. The results of executing BUF(X) are:

Value of BUF(X)	Next Item in Buffer
1	real
2	string
4	no data in buffer
5	integer
6	long
7	complex

BUF(X) will not return the value 3.

Example

```
 10 INPUT :
 20 IF BUF(0)=4 THEN GOTO 190
 30 IF BUF(0)=5 THEN DO
 40   INPUT :A:
 50   PRINT "INTEGER A=";A
 60   GOTO 20
 70 DOEND
 80 IF BUF(0)=1 THEN DO
 90   INPUT :B:
100   PRINT "REAL NO =";B
110   GOTO 20
120 DOEND
130 IF BUF(0)=2 THEN DO
140   INPUT :C$:
150   PRINT "STRING C=";C$
160   GOTO 20
170 DOEND
180 GOTO 20
190 PRINT "END OF BUFFER"
```

When run, the user can input any number of constants and they will be kept in the input buffer. This example assumes that no long or complex numbers will be input.

```
>RUN
?1.3,"X",576,35.2,66.6,75,"A","C"
REAL NO = 1.3
STRING C=X
INTEGER A= 576
REAL NO = 35.2
REAL NO = 66.6
INTEGER A= 75
STRING C=A
STRING C=C
END OF BUFFER
```

ENTER Statement

The ENTER statement provides the program with more control over the input operation. The statement can limit the amount of time allowed to input data from the input device (e.g., terminal), provide the program with the actual input time, indicate whether the data is of the correct type, and return logical device number of the user's terminal.

Form

There are three forms of the ENTER statement:

ENTER # terminal variable

ENTER time limit expression, actual time variable, input variable

ENTER # terminal variable, time limit expression, actual time variable, input variable

The *terminal variable* after # is used to return the logical device number of the terminal; the *time limit expression* specifies the time allowed for input; the *actual time variable* is assigned the actual time used; and the *input variable* is assigned the value typed in.

Explanation

The first form sets the terminal variable equal to the user's terminal logical device number.

The time limit expression specifies the length of time, in seconds, that the user is allowed to enter his input. The value must be in the range 1 to 255. If it is greater, 255 is used; if it is less, 1 is used.

The actual time variable is set to the approximate time, in seconds, that the user takes to respond. If an improper input is typed, this value is negated. If the user fails to respond within the allotted time, this variable is set to -256.

Only one value can be typed in for each ENTER statement and it is assigned to the input variable. A string should not be entered enclosed in quotes, but it may contain quotes. A string that is too long is truncated on the right.

The ENTER statement differs from the INPUT statement in that a "?" is not printed on the user terminal and the system returns to the program if the user does not respond within a specified time limit (there is no time limit on INPUT). Also, the program does not generate a linefeed after the user types in a carriage return.

Examples

```
  10 DIM C$[25]
  20 ENTER #A
  30 PRINT "TERMINAL NO.=";A
  40 PRINT "YOU HAVE 1 MINUTE TO TYPE 25 CHARACTERS FOR C$"
  50 ENTER 60,B,C$
  60 PRINT LIN(1);"ACTUAL TIME=";B
  70 PRINT C$
  80 PRINT LIN(1);"TYPE VALUE FOR C"
  90 ENTER #A,60,B,C
 100 PRINT LIN(1);"ACTUAL TIME=";B
 110 PRINT C
>RUN
TERMINAL NO.= 17
YOU HAVE 1 MINUTE TO TYPE 25 CHARACTERS FOR C$
EMBEDDED "QUOTES" O.K.
ACTUAL TIME= 13.41
EMBEDDED "QUOTES" O.K.

TYPE VALUE FOR C
25.7E-8
ACTUAL TIME= 6.62
 2.57000E-07
```

The system enters the logical terminal number in the variable A as a result of line 20; A can then be referenced as in line 30. Since ENTER does not provide a prompt character, it is useful to print some form of prompt particularly because there is a time limit on the input.

Note that the system does not provide a linefeed after input. It is therefore essential, if any output is to be printed after the input line, to provide a linefeed (use LIN function) within the PRINT statement. Without this linefeed, a subsequent output line overprints the input line.

A common use of ENTER is to test students:

```
  10 PRINT "WHAT IS .25 TIMES 75?"
  20 ENTER 30,T,X
  30 IF X=.25*75 THEN GOTO 70
  40 PRINT LIN(1),"SORRY,THE CORRECT ANSWER IS";.25*75
  50 PRINT "TRY THE NEXT PROBLEM"
  60 GOTO 80
  70 PRINT LIN(1);"CORRECT,YOU ANSWERED IN";T;"SECONDS"
  80 REM..THE NEXT PROBLEM COULD START HERE
>RUN
WHAT IS .25 TIMES 75?
18.75
CORRECT,YOU ANSWERED IN 3.35        SECONDS
```

> BASIC

When a BASIC/3000 program is waiting for input at the terminal as a result of an INPUT or ENTER statement, the user can interrupt input and request a new level of the BASIC/3000 Interpreter by typing > BASIC.

The computer returns a greater than sign (>) to prompt for other BASIC statements or commands.

The previous program is suspended until the user types EXIT. EXIT in this case returns control to the INPUT or ENTER statement in the previous program. The computer types two question marks (??) to signal that it is waiting for further input.

Example

```
  10 PRINT "WHAT IS THE SQUARE ROOT OF 94?"
  20 INPUT I
>RUN
WHAT IS THE SQUARE ROOT OF 94?
?>BASIC
BASIC 01.0
>10 PRINT SQR(94)
>RUN
 9.69536

>EXIT


??9.69536

>
```

The user responds to the INPUT prompt signal with > BASIC. He can then enter and run another program. EXIT returns control to the original program. He now enters the value he got as a result of the program run in > BASIC.

When BASIC/3000 is entered with > BASIC, it cannot be entered again in the same way. That is, there is no nesting of this feature.

Commands

So far we have used a set of commands (LIST, RUN, SCRATCH) for simple program manipulation. Both LIST and RUN have parameters and functions other than were illustrated. The full capability of commands used to run a program, to edit a program, and to save a program in the library are described here. The commands are:

RUN

The Editing Commands:

LIST

SCRATCH

DELETE

RENUMBER

LENGTH

Library Commands:

NAME

SAVE

GET

APPEND

PURGE

CATALOG

Commands in general are described in Section I. It should be recalled here that commands do not have labels; they are entered directly after the > prompt character and are executed immediately. Unlike statements, commands may not contain embedded blanks except between parameters. Some commands may be abbreviated.

Certain conventions are used in the command description:

UPPER-CASE	Key words that must be spelled correctly
lower-case	Words defined by the user
[ ]	Enclose optional items
{ }	Enclose required items
|	Separates alternatives, one of which must be chosen
. . .	Indicate the preceding item may be repeated

In the command descriptions, certain keywords are used:

programname	a BASIC/3000 program file
filename	a non-BASIC/3000 file
asciifile	an MPE/3000 ASCII file

Key word parameters may be in any order.

RUN

The RUN command executes a BASIC/3000 program; the form is

RUN [programname] [,label] [,OUT=asciifile] [,NOWARN] [,FREQ] [,NOECHO] [,MR]

If *programname* is specified, the named program is retrieved from the user's library and made the current program. Any program previously in the user's work area is scratched. The current program then is executed. Any traces and breakpoints are deleted. (Traces and breakpoints are described in Section VII, Debugging.)

If *label* is specified, execution starts at the first executable statement at or after the label number. The starting statement must not be within a function definition. If the label specifies a DEF statement, execution begins at the first executable statement following the function definition.

OUT=*asciifile* diverts all printed output and trace information to the specified ASCII file.

NOWARN suppresses warning messages.

FREQ causes a table to be printed following program execution that summarizes the usage of all statements in all programs that are part of the run. There may be more than one program in a run when segmentation is used (see Section X, Segmentation).

NOECHO suppresses printing of program input when the input and list files are not on the same device.

MR allows the execution of a program that locks multiple files, provided that the user has MR capability (see Section VIII, Dynamic Locking).

Examples of RUN

The program below is the current program:

```
 10 DATA 3,5,7
 20 DATA 9,11,13
 30 DATA 15,17,19
 40 READ (FOR I=1 TO 8,C[I]),D
 50 PRINT (FOR I=1 TO 8,C[I]),D
 60 RESTORE
 70 READ A
 80 RESTORE 20
 90 READ B
100 RESTORE 30
110 READ C
120 PRINT A,B,C
```

First the entire program is run, then it is run starting at line 60:

```
>RUN
 3              5              7              9              11
 13             15             17             19
 3              9              15

>RUN,60
 3              9              15
```

Next the same program is run with a frequency table:

```
>RUN,FREQ
 3                5                7                9                11
 13               15               17               19
 3                9                15


FREQUENCY TABLE

TOTAL STATEMENTS = 12
TOTAL TIME = .297 SECONDS

       FREQUENCY      EXECUTION TIME
LABEL  COUNT PCT     AVE    TOTAL   PCT
  10       1   8     .001    .001     0
  20       1   8     .000    .000     0
  30       1   8     .001    .001     0
  40       1   8     .022    .022     7
  50       1   8     .160    .160    54
  60       1   8     .001    .001     0
  70       1   8     .002    .002     1
  80       1   8     .000    .000     0
  90       1   8     .002    .002     1
 100       1   8     .001    .001     0
 110       1   8     .002    .002     1
 120       1   8     .000    .000     0
SYSTEM OVERHEAD              .105    35
```

Editing Commands

The editing commands always affect the current program, that is, the program that is currently being entered at the terminal.

LIST

The LIST command lists all or part of the current program; the form is

LIST [first [- last]] [, OUT=asciifile] [, RECSIZE=number] [,NONAME]

where *first* and *last* specify the range of statements to be listed, and *asciifile* specifies the ASCII file to which the list is diverted. If RECSIZE is specified, *number* specifies the number of characters per record for the list file. If *NONAME* is specified, the program name is not listed; this is useful when listing programs to be read back with the XEQ command. The default parameters are the normal list file and a record size of 72 characters per record. If neither *first* nor *last* is specified, the entire program is listed. If only *first* is specified, just that statement is listed.

Examples

```
>LIST
```

The entire current program is listed at the terminal.

```
>LIST 1-100,OUT=FASTFILE,RECSIZE=130
```

Statements 1 through 100 of the current program are listed on the file FASTFILE with a record size of 130.

Note that a listing can be stopped by pressing the CTRL Y key. The user is returned to BASIC control.

SCRATCH

The SCRATCH command deletes the entire current program and its name; the form is

SCRATCH | SCR

SCRATCH also clears traces and breakpoints. (Traces and breakpoints are described in Section VII, Debugging).

Example

```
>SCR
```

The current program is deleted, and a new current program can be entered in the user's work area.

DELETE

The DELETE command deletes one or more specified statements; the form is

{*DELETE | DEL*} *first* [- *last*] [, *first* [- *last*]] . . .

where *first* and *last* are statement labels; the statements referenced by the parameters are deleted from the program. Each *first-last* pair specifies a range of statements which are to be deleted. If a *first* is given without a *last*, only the one statement is deleted.

Example

```
>DEL 45,75,400-700
```

Statements 45, 75, and all statements from 400 through 700 inclusive are deleted from the user's current program.

RENUMBER

The RENUMBER command allows the user to renumber any of the statements in the current program; the form is

{ *RENUM* | *RENUMBER* } [*newfirst* [, *delta* [, *oldfirst* [- *oldlast*]]]]

oldfirst and *oldlast* specify the range of original statements to be renumbered (defaults are 1—15999). If only *oldfirst* is specified, the default for *oldlast* is 15999. The first of these statements is assigned the number *newfirst* (default is 10) and each of the remainder is assigned a statement number *delta* greater than its predecessor (default for *delta* is 10). Any statement in the program which references a renumbered statement is changed as required for consistency.

Examples

```
>RENUMBER
```

The statements in the current program are renumbered in increments of 10 starting with statement number 10.

```
>RENUM 5,5,1-890
```

The old statement numbers 1 through 890 are renumbered starting with 5 and increasing by 5.

LENGTH

The LENGTH command reports the size of the current program; the form is

LENGTH | *LEN*

The length of the current program (in 16-bit words) is printed

Example

```
>LENGTH
```

The length of the current program is printed.

Examples Using Editing Commands

After the user enters text at a terminal, mistakes can be corrected by pressing the CNTL H (or H^C) key or the backspace key.

```
>10 INPUG\T A,B,C,D,E
>20 REM..INPUT 5 VALUES
>30 LET S=(A@\+B+C+D+E)?\/5
>40 REM..S=AVERAGE OF 5 INPUT VALUES
>50 PRINT S
```

LIST correctly lists the program:

```
>LIST
  10 INPUT A,B,C,D,E
  20 REM..INPUT 5 VALUES
  30 LET S=(A+B+C+D+E)/5
  40 REM..S=AVERAGE OF 5 INPUT VALUES
  50 PRINT S
```

LENGTH gives the length in computer words:

```
>LENGTH
53 WORDS.
```

The remark lines are deleted and the program is listed:

```
>DELETE 20,40
>LIST
  10 INPUT A,B,C,D,E
  30 LET S=(A+B+C+D+E)/5
  50 PRINT S
```

Next, the program is renumbered and listed again:

```
>RENUMBER
>LIST
  10 INPUT A,B,C,D,E
  20 LET S=(A+B+C+D+E)/5
  30 PRINT S
```

The program is scratched. When LIST is now specified, there is no current program; the computer returns a ">" to prompt for further entries:

```
>SCRATCH
>LIST
>
```

Library Commands

When a current program is complete, and if it is to be used again, it should be saved in the user's library. A copy of the current program identified by a name is kept in the library when the program is saved. The current program is not affected; it remains the current program until log off, or until it is scratched with the SCRATCH command.

When a program is saved, it must be given a name either with the NAME or SAVE command. The program name is used to get, to append, or to purge a program in the user's group library. The name must be unique among names in a particular user's group library, but it may be duplicated in other groups. A catalog of the programs and files contained in the user's library may be requested with the CATALOG command.

NAME

The NAME command assigns a name to the current program; the form is

NAME programname

The *programname* specified is assigned to the current program. The *programname* can be any combination of eight alphabetic and numeric characters, beginning with an alphabetic character.

Example

```
>NAME PROGX
```

The current program is assigned the name PROGX.

SAVE

The SAVE command stores a copy of the current program in the user's library; the form is

SAVE [programname] [!] [,FAST] [,RUNONLY] [,MR]

If *programname* is specified, that name is given to the saved copy, but not to the current program. If *programname* is omitted, the name of the current program is assumed; in this case, the program must have been named before it can be saved. If there is no file with the same name in the user's library, a new file is created and a copy of the current program is stored in it. If a file with the same name already exists in the library, the SAVE command is rejected unless the exclamation mark is specified, in which case the original file is purged and a new file created.

FAST causes the program to be saved in pseudo-compiled form so that it can be RUN more quickly. It also ensures that the program is valid (matching FOR-NEXT pairs, etc.).

A program saved for RUNONLY is assumed to be free of errors and ready for execution. When a RUNONLY program is brought into the user's work area with GET, certain commands are illegal until a SCRATCH or another GET. For instance, a RUNONLY program cannot be listed or modified. The only commands legal when a RUNONLY program is current are:

ABORT
CATALOG
CREATE
DUMP
EXIT
GET
KEY
PURGE
RESUME or GO
RUN
SCRATCH
SPOOL
SYSTEM
TAPE
XEQ

MR saves a program with MR status, if the user has MR capability (see Section VIII, Dynamic Locking). Otherwise, the following message appears on the terminal:

COMMAND EXCEEDS USER CAPABILITY

Examples

```
>SAVE PROGX
```

The name PROGX is assigned to the copy of the current program that is saved in the user's library.

```
>NAME PROGX
>SAVE
```

The current program is given the name PROGX, and then a copy is saved in the user's library.

```
>SAVE PROGX!,FAST,RUNONLY
```

A copy of the current program is assigned the name PROGX and stored in the user's library; any other program with the name PROGX is purged from the library. The program is saved in pseudo-compiled form and, if retrieved as the current program, only commands legal with RUNONLY can be used.

GET

The GET command loads a specified BASIC/3000 program into the user's working space; the form is

GET programname

where *programname* is the name of a program to replace the current program. GET deletes all traces and breakpoints.

Example

```
>GET SEARCH
```

SEARCH is a program saved in the user's library. It is now also available in the user's work area replacing any previous program in that area.

PURGE

The PURGE command removes a file or program from the user's library; the form is

PURGE {*basicfile* | *programname* | *filename*}

The file or program specified is deleted from the user's library; it is not recoverable once it has been purged.

Example

```
>PURGE PROGX
```

PROGX is a file or program in the user's library. It is no longer available to the user and its name may be assigned to another file or program.

APPEND

The APPEND command appends a specified program to the user's current program; the form is

APPEND programname

The program specified is appended to the end of the current program. The last sequence number of the current program must be smaller than the first sequence number of the appended program. Programs which have been saved in pseudo-compiled form (see SAVE command) and RUNONLY programs cannot be appended.

Example

```
>APPEND PROGX
```

PROGX is a program saved in the user's library. It is appended to the program currently in the user's work area.

CATALOG

The CATALOG command provides a list of programs or files specified by the user. The list includes the program or file name, the type, the number of logical records, and if desired, the record width.

The form is:

{ CAT | CATALOG } [fileset] [,ALL] [,RECSIZE] [,OUT=asciifile] [,START=filename]

where:

fileset	one or more files or programs referenced by file name, group name, and/or account name. When *fileset* is omitted, all the files in the user's log-on group are listed. (See the next page for a full description of *fileset*.)
ALL	all ASCII and Binary files are included in the list; if ALL is omitted, only BASIC files and programs are listed.
RECSIZE	requests the record width for each file. If RECSIZE is omitted, record width is not listed.

OUT=*asciifile*	the file listing is diverted to the specified ASCII file; if OUT is omitted, the list is on the list device (e.g., the terminal).
START=*filename*	the listing starts with the specified file name.

For each file listed, the file name, the type (BF for BASIC file, SP for saved program, FP for fast saved program, A for ASCII, B for Binary) and the number of records in the file are listed. The record width is listed if RECSIZE is specified; the width is in bytes for ASCII files, in words otherwise. The listing is printed in as many columns as will fit across the width of the list device.

Output can be stopped with CTRL Y, as with the LIST command.

The fileset parameter has three fields that allow the user to request descriptions of one file alone, or various sets of files. The filename field indicates a specific file or all files within the units designated by the other fields. The group field denotes the group to which the files belong. The account field denotes the account to which the group belongs, or it may specify all accounts in the system. To specify all files, groups, or accounts, the user enters the character @ in the appropriate field. The three fields are separated by periods.

The table below shows the possible combination of entries in *fileset*:

File Field	Group Field	Account Field	Entry Example	Meaning
filename	groupname	accountname	FILE.GROUP.ACCT	The file named, in the group and account designated.
filename	groupname		FILE.GROUP	The file named, in the group designated under the log-on account.
filename			FILE	The file name, under the log-on group.
@	groupname	accountname	@.GROUP.ACCT	All files in the group named, under the designated account.
@	groupname		@.GROUP	All files in the group named, under the log-on account.
@			@	All files in the log-on group. This is the default case.
@	@	accountname	@.@.ACCT	All files in all groups under the account named.
@	@		@.@	All files in all groups under the log-on account.
@	@	@	@.@.@	All files in the system.

@ means all

Examples Using Library Commands

A program is input, named, and saved in the user's library. It is then scratched as the current program:

```
>100 INPUT A,B,C,D,E
>120 LET S=(A+B+C+D+E)/5
>130 PRINT S
>NAME AVERAGE
>SAVE
>SCRATCH
```

A second program is entered, named, and saved. The first program is then appended to this program to make a third program. It too is named and saved:

```
>10 INPUT R
>20 P=3.14
>30 A=P*R**2
>40 PRINT A
>NAME AREA
>SAVE
>APPEND AVERAGE
>SAVE CALC
```

Any of these programs may now be brought back as the current program with GET. To illustrate, each is retrieved and then listed:

```
>GET AVERAGE
>LIST
AVERAGE
 100 INPUT A,B,C,D,E
 120 LET S=(A+B+C+D+E)/5
 130 PRINT S
>GET AREA
>LIST
AREA
  10 INPUT R
  20 P=3.14
  30 A=P*R**2
  40 PRINT A
>GET CALC
>LIST
CALC
  10 INPUT R
  20 P=3.14
  30 A=P*R**2
  40 PRINT A
 100 INPUT A,B,C,D,E
 120 LET S=(A+B+C+D+E)/5
 130 PRINT S
```

To determine whether a particular program is in the user's library, he can type CATALOG followed by the program name. If there are not too many files in the current log-on group, he can simply type CATALOG to get a list of all the files currently saved.

In this example, the user requests a catalog of the program CALC. He then types RUN CALC and the program will be retrieved from the library and run:

```
>CATALOG CALC

ACCOUNT=LANG        GROUP=BASIC

  NAME     RECORDS    NAME     RECORDS    NAME     RECORDS
CALC       SP      2
>


>RUN CALC
CALC
?60
 11304
?34,56,43,61,54,73
 49.6
```

If there is no further need for the saved programs, each may be purged as follows:

```
>PURGE CALC
>PURGE AREA
>PURGE AVERAGE
```

The program CALC remains the current program as a result of the RUN CALC command until it is scratched or is replaced by another program in the user's library, or until the user exits from BASIC.

Saved programs remain in the library after log-off and can only be removed with the PURGE command.