Burroughs B 1700 SYSTEMS

BASIC
REFERENCE MANUAL

$3.00

Burroughs

B 1700 SYSTEMS

BASIC

REFERENCE MANUAL

Burroughs Corporation
Detroit, Michigan 48232

$3.00

Printed in U. S. America

6-73

1067535

Correspondence regarding this document should be forwarded using the Remarks Form at the back of the manual, or may be addressed directly to Systems Documentation, Technical Information Organization, TIC-Central, Burroughs Corporation, Burroughs Place, Detroit, Michigan 48232.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

INTRODUCTION

The purpose of this manual is to provide a description of the BASIC language as implemented on the B 1700 systems. BASIC, an acronym for Beginners All-purpose Symbolic Instruction Code, was initially developed under the direction of Professors Kemeny and Kurtz at Dartmouth College. B 1700 BASIC includes the capabilities of the original Dartmouth College BASIC plus extensions provided for compatability with the General Electric MARK II BASIC language.

BASIC is a problem-oriented language designed for a wide range of applications and may be easily applied to both business/commercial, as well as engineering/scientific processing tasks. The BASIC language is designed for use both by individuals who have little previous knowledge of computers, as well as individuals with considerable programming experience. A distinct advantage of BASIC is that its rules of form and grammar are learned quite easily.

A program written in B 1700 BASIC, called a source program, is accepted as input by the BASIC compiler. The compiler first verifies that each source statement is syntactically correct and then converts the source program into BASIC S-CODE. The S-CODE generated by the compiler can then be executed on the B 1700 using the BASIC INTERPRETER. The INTERPRETER causes the systems hardware to perform the operations specified by the S-CODE and thus the source program. For more detailed information regarding the function of S-CODE and its relation to the INTERPRETER and the hardware, refer to the B 1700 System Reference Manual (form 1057155).

The B 1700 BASIC compiler operates under the control of a Master Control Program (MCP). Similarly the S-CODE generated by the compiler is executed under control of the MCP.

B 1700 BASIC, at this time, does not provide the following capabilities which will be implemented at a later date:

a. String variables.
b. String expressions.
c. String variable assignment statement.

d. CHANGE statement.
e. PRINT USING statement.
f. Image statement.
g. Binary files.
h. CHAIN statement.
i. CALL statement.
j. Matrix operations.

SECTION 1

COMPONENTS OF BASIC

GENERAL

BASIC statements are composed of certain key words used together with the fundamental elements of the language. The fundamental elements of B 1700 BASIC are discussed in this section.

NUMERIC CONSTANTS

A numeric constant consists of a series of decimal digits, may be preceded by a sign (+ or -), and may, but need not, contain a decimal point. Unlike some programming languages, BASIC does not distinguish between numbers containing a decimal point (i.e., real numbers) and those written without a decimal point (i.e., integers). All numeric values, in BASIC, are stored internally in floating-point form and thus are handled as if they were real numbers.

A numeric constant may be expressed in scientific form through the use of the letter E followed by a signed or unsigned 1- or 2-digit integer which is the power of 10 that the number preceding the letter E is to be raised. As an example, 2.145E-4 represents the value .0002145. The decimal point in a number expressed in E notation may follow or precede any digit as long as the correct power of 10 is expressed.

In B 1700 BASIC, the range for numeric values is approximately 10** (-77) through 10**77. Zero is also allowed. The precision for numeric values is 9 significant digits.

STRING CONSTANTS

A string constant is enclosed in quotation marks and consists of any sequence of EBCDIC characters valid to the B 1700 processor. Since the quotation mark is used to define the beginning and the end of a string constant, the quotation mark may not be embedded in the string. Examples of string constants are: "A", "12", and "$*1-%#".

Blank characters are significant in BASIC only when contained in a string constant.

In B 1700 BASIC, a string constant may contain a maximum of 255 characters.

NUMERIC VARIABLES

A numeric variable is a symbolic name used to represent a numeric value. Unlike a constant, whose value remains fixed, a variable name represents a data item whose value may be changed during program execution.

A numeric variable name may be a single alphabetic letter or a single alphabetic letter followed by a single decimal digit. Valid numeric variable names are:

A, A0, A1, A2, ... Z, Z0, Z1, ... Z9.

In B 1700 BASIC, each numeric variable used in a program is initialized to zero during compilation.

SUBSCRIPTED VARIABLES

In addition to the previously described variables (referred to as simple variables) a variable may also be subscripted to refer to a particular element in an array. A subscripted variable in BASIC represents an element of a one- or two-dimensional array. The general form of a subscripted variable is as follows:

n(s1,s2)

where n is the array name and s1 and s2 are arithmetic expressions which determine the values of the subscripts. The second subscript, s2, is optional depending on the number of dimensions the array has.

A subscripted variable represents a specific element in a numeric array and consists of the name of the array followed by a set of subscripts indicating the position of the element within the array enclosed in parentheses. An array may have a maximum of two dimensions.

An array name must be a single alphabetic letter. The same letter, however, may be used in a program as both a simple variable name and an array name.

Unless an array is dimensioned in a DIM statement, BASIC assumes that each subscript of a subscripted variable can range from 0 through 10. When either subscript of a subscripted variable exceeds 10, the dimensions associated with the array name must be declared in a DIM statement. (See Section 4 for additional information on the DIM statement.)

When an array is used in a program, whether dimensioned in a DIM statement or not, the elements n(0) and n(0,0) are available in a one-dimensional and two-dimensional array, respectively.

A subscript may be any arithmetic expression. When an arithmetic expression is used as a subscript, it is evaluated and the resulting value is truncated to an integer before being used as a subscript.

In B 1700 BASIC, each array used in a program is initialized to zero during compilation.

EXPRESSIONS

An expression is any constant, variable, function reference, or a combination of these separated by operators or parentheses. There are two types of expressions:

a. Arithmetic.
b. Relational.

Arithmetic Expressions

An arithmetic expression is a rule for computing a numerical value and is any numeric constant, numeric variable (either simple or subscripted), arithmetic function reference, or combination of these separated by arithmetic operators or parentheses.

An arithmetic expression may contain the following arithmetic operators:

Operator	Meaning
↑ or ^ or **	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction
-	Negation

Negation is a unary operator; that is, it operates on only one variable. The other operators are binary operators requiring two variables (or constants). No arithmetic operator may be assumed to be present. Examples of invalid arithmetic expressions are:

```
A//B
(A+B) (C+D)
```

Parentheses may be used in an arithmetic expression to denote the order in which operations are to be performed. Parentheses have first precedence in determining the order of evaluation; when nested parentheses occur, evaluation proceeds from the innermost set to the outermost set.

The precedence order (or hierarchy of arithmetic operators) used in evaluating an arithmetic expression is as follows:

(highest)	Function reference
	Exponentiation
	Multiplication and division
(lowest)	Addition and subtraction

The order in which operators of the same level are performed is from left to right, except for exponentiation, where evaluation is from right to left, e.g., A**B**C is evaluated as A**(B**C).

References to both user-defined functions and mathematical functions may also appear in arithmetic expressions. User defined functions and mathematical functions are discussed in subsequent sections of this manual.

Relational Expressions

A relation expresses a condition between two arithmetic expressions that, when evaluated, is either true or false. A relational expression consists of any two arithmetic expressions separated by a relational operator. The relational operators and their meanings are:

Operator	Meaning
<	Less than
<=	Less than or equal to
=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to

Chains of relations are not permitted, e.g., A<B<C.

SYNTAX RULES

The following rules must be observed in the formation of a BASIC program:

a. Each source statement must be punched on a separate card.

b. A statement may not be continued onto a second card.

c. Each statement must begin with a unique line number. The line number starts in column 1 and may consist of as many as five decimal digits. The line number is used as both a statement label and a sequence number. The source statements comprising a program are required to be in ascending numerical sequence as each source statement being read by the compiler is sequence checked.

d. Spacing (blanks) has no significance within a source statement except for information contained in string constants. Spaces can be used to make a program more readable.

SECTION 2

ASSIGNMENT STATEMENTS

GENERAL

The assignment statement is used to assign a value to a simple or subscripted variable. The following assignment statements are discussed in this section:

a. Arithmetic assignment statement.

b. Multiple arithmetic assignment statement.

ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement is used to assign a value to a numeric variable. The arithmetic assignment statement has the following format:

n LET < variable > = < arithmetic expression >

where n is the line number of the LET statement, <variable> is either a simple or subscripted numeric variable, and <arithmetic expression> is a numeric constant, numeric variable (either simple or subscripted), or an arithmetic expression.

The word LET is optional.

Examples:

```
100 LET A = 4
15 LET P(2,3) = SQR(A**2 + B**2)
10 Z1 = ABS(B(1,1))
```

MULTIPLE ARITHMETIC ASSIGNMENT STATEMENT

The multiple arithmetic assignment statement is used to assign a value to more than one numeric variable. The multiple arithmetic assignment statement has the following format:

n LET <variable1> =<variable2>=<variable3>=...=<arithmetic expression>

where n is the line number of the multiple arithmetic assignment statement; <variable1> = <variable2> = <variable3> =... represents a sequence of simple or subscripted numeric variable names separated by the equal sign; and <arithmetic expression> is a numeric constant, numeric variable (either simple or subscripted), or an arithmetic expression.

This statement assigns the value of the <arithmetic expression> to each variable name on a left-to-right basis. For example, the multiple arithmetic assignment statement

```
10 LET X=I=A(I)=4
```

assigns the value 4 to X, I, and A(4) as though the following statements had been used:

```
10 LET X=4
20 LET I=4
30 LET A(4)=4
```

The word LET is optional in this statement.

Examples:

```
10 LET A=B=C=0
55 LET V(1) =W=X=Y=X= 2*A+B
```

SECTION 3

CONTROL STATEMENTS

GENERAL

Normally, the executable statements in a BASIC program are executed in line number sequence; that is, after one statement has been executed, the statement immediately following it is executed. Control statements are used to alter the normal flow of a program. They may transfer control to another part of the program, terminate program execution, or control iterative processes. The following control statements are discussed in this section:

a. GO TO statement.
b. IF statement.
c. ON statement.
d. FOR statement.
e. NEXT statement.
f. END statement.
g. STOP statement.

GO TO STATEMENT

At some point in a program it may be necessary to unconditionally transfer control to a line number out of the physical (line number) sequence of instructions. The GO TO statement is used to transfer control to the statement whose line number is specified. The GO TO statement has the following format:

n GO TO <line number>

where n is the line number of the GO TO statement and <line number> is the line number of the statement to which control will be transferred.

Example:

```
110 GO TO 200
```

IF STATEMENT

Execution of the IF statement causes a relational expression to be evaluated and the sequence of execution of the program statements to be altered only if the specified condition is satisfied. The IF statement has the following format:

$$n \text{ IF } \langle \text{relational expression} \rangle \left\{ \begin{matrix} \text{THEN} \\ \text{GO TO} \end{matrix} \right\} \langle \text{line number} \rangle$$

where n is the line number of the IF statement, <relational expression> is a relational expression as described in section 1, under the heading RELATIONAL EXPRESSIONS, and <line number> is the line number to which control will be transferred if the <relational expression> is satisfied. Either of the key words THEN or GO TO may be used in the syntax of this statement.

If the <relational expression> is satisfied, program control will be transferred to the statement specified by <line number>. If the <relational expression> is not satisfied, program control will be passed on to the next statement in line number sequence following n. As an example,

```
30 IF A<100 THEN 70
```

specifies that program control is to be transferred from line number 30 to line number 70 as long as A is less than 100. If A is equal to or greater than 100, control will pass to line 40 or whatever line number is the next line number in sequence following line 30.

ON STATEMENT

Execution of the ON statement allows program control to be transferred to one of several line numbers, depending upon the value of a specified arithmetic expression. The ON statement has the following format:

$$n \text{ ON } \langle \text{arithmetic expression} \rangle \left\{ \begin{matrix} \text{THEN} \\ \text{GO TO} \end{matrix} \right\} \langle \text{linel} \rangle, \langle \text{line2} \rangle, \langle \text{line3} \rangle, \ldots$$

where n is the line number of the ON statement; <arithmetic expression> is any numeric constant, numeric variable (either simple or subscripted) or arithmetic expression; and <linel>, <line 2>, <line3>,... represent the line numbers to which program control will be transferred, depending upon the value of the <arithmetic expression>.

If the value of the <arithmetic expression> is 1, program control will transfer to the statement whose line number is <linel>; a value of 2 will transfer control to the statement whose line number is <line2>, and so on. After evaluation, if the integer part of the value of the <arithmetic expression> is less

than 1 or greater than the number of line numbers listed in the computed GO TO statement, a run-time error message is printed and program execution is terminated.

Example:

```
10 K = 4
20 ON K GO TO 60, 40, 50, 70
```

Execution of the above two statements causes program control to be transferred to the statement at line number 70.

PROGRAM LOOPS

Often it is necessary to have the same portion of a program repeated several times, possibly with some slight modification each time that portion is executed. This repetitive process in a program is referred to as a loop. An example of a program loop is:

```
10 READ A,B
20 LET C = A+B/10
30 LET D = A*C
40 PRINT "C= "; C, "D= ";D
50 GO TO 10
60 DATA 10,20,30,40,50,60
70 END
```

In this example, line numbers 10 through 40 would be executed in the normal manner. When line number 50 is executed, program control would be unconditionally transferred back to line number 10 once more. Line number 10 through 40 would again be repeated and when line number 50 is executed, control is once more returned to line number 10. This repetitive action continues until all of the data items in line number 60 are exhausted. At that time, program execution would be terminated. Since the loop is one of the most useful tools in programming, two statements are provided in BASIC for controlling program loops.

FOR AND NEXT STATEMENTS

The FOR statement together with the NEXT statement is used to cause the same portion of a program to be repeated a specified number of times. The FOR statement may have either of the following formats:

1. n FOR <variable> = <expression1> TO <expression2>
2. n FOR <variable> = <expression1> TO <expression2> STEP <expression3>

where n is the line number of the FOR statement; <variable> is a simple numeric variable; and <expression1>, <expression2>, and <expression3> are

any numeric constant, numeric variable (either simple or subscripted), or arithmetic expression.

The <variable> following the word FOR must be an unsubscripted numeric variable and is called the index of the loop. In the general form <expression1>, <expression2>, and <expression3> are referred to as the initial, final, and incremental parameters, respectively, for the index <variable>. The value of <expression1> is the initial value assigned to the index <variable>; the value of <expression2> is the maximum value which the index <variable> may attain; and the value of <expression3> is the increment or decrement to be added to the index <variable> on each pass through the loop. If a STEP clause is not specified (option 1), the value of <expression3> is assumed to be 1.

A NEXT statement must follow the FOR statement. The NEXT statement is used to delimit the statements which are to be executed repeatedly (the range of the loop). The NEXT statement has the following format:

n NEXT <variable>

where n is the line number of the NEXT statement and <variable> is the same index <variable> used in the companion FOR statement.

Execution of a FOR statement results in the following action:

a. The values of <expression1>, <expression2>, and <expression3> are calculated. If a STEP clause is not specified, the value of <expression3> is assumed to be 1.
b. The value of <expression1> is assigned to a temporary simple variable supplied by the compiler.
c. The value of <expression3> is tested to determine whether the step-size is positive or negative. If the value of <expression3> is greater than or equal to zero, step d. is performed; otherwise, step e. is performed.
d. The value of the temporary variable is compared to the value of <expression2>. If the value of the temporary variable is greater than the value of <expression2>, the loop is completed and program control is transferred to the statement following the NEXT statement; otherwise, step f. is performed.
e. The value of the temporary variable is compared to the value of <expression2>. If the value of the temporary variable is less than the value of <expression2>, the loop is completed and program control is transferred to the statement following the NEXT statement; otherwise, step f. is performed.

f. The value of the temporary variable is assigned to the index <variable>.

g. All executable statements between the FOR and the NEXT statements, i.e., the range of the loop, are executed.

h. The value of <expression3> is added to the temporary variable, and steps c. through g. are repeated until the loop is satisfied.

When it is desired to increment the index <variable> by something other than 1, the STEP clause (format 2) is used in the FOR statement. The step-size need not be an integer. For example, 10 FOR I = 1 TO 3 STEP .1 is a valid FOR statement. When the STEP clause is used, <expression3> must always assume a non-zero value; otherwise, the range of the loop will be executed indefinitely.

The value of <expression1>, <expression2>, and <expression3> may each be either positive or negative. However, if the initial final and incremental values in the FOR statement form an impossible combination, the range of the loop will not be executed and program control will be transferred to the statement following the companion NEXT statement. The following FOR statement contains an impossible combination of control parameters: 10 FOR I = 3 TO -3.

A FOR...NEXT loop may appear within the range of another FOR...NEXT loop. When nested loops are used, the inner loop must completely reside within the range of the outer loop, that is, they must not cross as illustrated in the following:

Legal Nesting

```
10 FOR X=1 TO 5
...
50  FOR Y=1 TO 5
...
60  NEXT Y
...
90 NEXT X
```

Illegal Nesting

```
10 FOR X=1 TO 5
20 FOR Y=1 TO 5
...
80 NEXT X
90 NEXT Y
```

The same index <variable> may not be used in two loops, one of which is nested within the other.

FOR...NEXT loops may not be nested to more than 20 levels.

Example:

```
05 FOR I=1 TO 3
10   READ A,B
20   LET C=A+B/10
```

```
30   LET D=A*C
40   PRINT "C= ";C,"D= ";D
50 NEXT I
60 DATA 10,20,30,40,50,60
70 END
```

END STATEMENT

The END statement is used to indicate the end of a BASIC program. The END statement has the following format:

```
n END
```

where n is the line number of the END statement.

In B 1700 BASIC, the END statement is optional in a program. The END statement, if used, is treated in the same manner as the STOP statement.

STOP STATEMENT

The STOP statement is used to stop the execution of a program. The STOP statement may appear anywhere in the program. The STOP statement has the following format:

```
n STOP
```

where n is the line number of the STOP statement.

More than one STOP statement may be used within a BASIC program.

SECTION 4

SPECIFICATION STATEMENTS

GENERAL

Specification statements are non-executable statements used to supply compile-time information about program variables to the compiler. The DIM statement is the only specification statement in the BASIC language and must be used in a program to declare the size of an array whenever one or both dimensions exceed 10.

DIM STATEMENT

The DIM statement is used to specify the dimensions of a designated array. The DIM statement has the following format:

n DIM <array declarator list>

where n is the line number of the DIM statement and <array declarator list> is a list of one or more array declarations sep ated by commas. An array declaration consists of an array name followed by a left parenthesis, followed by either a single positive integer indicating a one-dimensional array or two positive integers separated by a comma indicating a two-dimensional array, followed by a right parenthesis. The integer or integers specify the maximum subscript value for each dimension of their respective array names.

An array name in B 1700 BASIC may be any single alphabetic letter. Valid array names are:

A,B,C,.....,Z

The DIM statement specifies the number of data elements which may be contained in an array. Unless an array name appears in a DIM statement, BASIC by default reserves space for elements 0 through 10 for one-dimensional arrays (11 spaces total), and elements in both rows and columns 0 through 10 for two-dimensional arrays (121 spaces total).

When an array is used in a program, whether dimensioned in a DIM statement or not, the elements <array name>(0) and <array name>(0,0) are available in a one-dimensional and two-dimensional array, respectively.

The maximum number of elements permitted in an array is 262143 (i.e., 2**18-1). An array may have a maximum of 2 dimensions.

Example:

```
10 DIM A(30), B(20,40), C(3), D(2,2)
```

SECTION 5

PROGRAM DOCUMENTATION

GENERAL

Explanatory remarks, included throughout a program for documentational purposes, are an important part of any program. In BASIC, comments preceded by an apostrophe may follow a statement, or entire lines in a program may be devoted to remarks through use of the REM statement.

REM STATEMENT

The REM (REMARKS) statement is used to insert explanatory remarks at any point within a BASIC program. The REM statement has the following format:

n REM <remark>

where n is the line number of the REM statement and <remark> consists of any valid EBCDIC characters.

The REM statement allows entire lines of documentation to be included in a BASIC program. When the first three characters following the line number of a statement are REM, the compiler ignores the entire statement. Although the contents of each REM statement are ignored, the line number of a REM statement may be referenced in a GO TO, IF-THEN, ON-GO TO, or GOSUB statement. This enables control to be transferred to a description of the routine which follows, thus enhancing the readability of a BASIC program.

COMMENTS

Comments may be included on the same line following a BASIC statement if the comment is separated from the statement by an apostrophe (') character. Anything in the columns following the ' character through the end of that line will be ignored by the compiler and treated as a comment.

Example:

```
010 REM  ** FINAL GRADE CALCULATION PROGRAM **
020 REM     G = EXAMINATION GRADE
030 REM     W = EXAMINATION WEIGHT OF EACH EXAM
040 REM     F = FINAL GRADE BASED ON FIVE EXAMS
050 REM
060 DIM W(5)
070 FOR E = 1 TO 5                    'READ EXAM WEIGHTS
080 READ W(E)                         'FOR FIVE EXAMS
090 NEXT E
100 FOR S = 1 TO 4                    'S = STUDENT NUMBER
110 LET F = 0                         'INITIALIZE FINAL GRADE TO 0
120     FOR E = 1 TO 5
130     READ G                        'READ EXAM(E)
140     LET F = F+W(E)*G              'ACCUMULATE FINAL GRADE
150     NEXT E
160 PRINT "STUDENT NUMBER ";S, "FINAL GRADE= ";F
170 NEXT X
180 STOP
190 DATA .10,.15,.20,.25,.30          'EXAM WEIGHTS
200 DATA 75,82,85,79,91               'STUDENT 1 GRADES
210 DATA 100,76,88,92,98              'STUDENT 2 GRADES
220 DATA 63,60,75,72,83               'STUDENT 3 GRADES
230 DATA 94,87,90,91,95               'STUDENT 4 GRADES
240 END
```

SECTION 6

INPUT/OUTPUT

GENERAL

In BASIC, data files are considered to be either internal or external to the program which processes those files. External files, discussed in section 7, are files which reside on disk. Internal files are data contained within the program in DATA statements or data entered into the program by means of an INPUT statement. The following input/output statements pertaining to internal files are discussed in this section:

a. READ and DATA
b. RESTORE
c. INPUT
d. PRINT

READ AND DATA STATEMENTS

The READ statement is used to assign values to the variables listed in the READ statement, using data elements from the associated DATA statement(s). The READ statement has the following format:

n READ <list of variable names>

where n is the line number of the READ statement, and <list of variable names> is a list of one or more numeric variables (either simple or subscripted) separated by commas.

The DATA statement is always used in conjunction with the READ statement and contains the data values which are to be assigned to the variables appearing in the READ statement list. The DATA statement has the following format:

n DATA <list of constants>

where n is the line number of the DATA statement and <list of constants> is a list of one or more numeric constants separated by commas.

Only numeric constants may appear in a DATA statement.

Prior to execution of a BASIC program, all DATA statements contained in a program are combined into one numeric data block according to the order in which the DATA statements appear in that program. Each time a READ statement

is executed, a constant is accessed from the data block for each variable name contained in the read statement list. If the data block is exhausted and a READ statement attempts to initialize another variable, program execution is terminated with an OUT OF DATA message. When more constants are supplied in DATA statements in a program than are required by the READ statement(s), the extra constants are ignored.

The READ statement must always be associated with a DATA statement.

Examples:

```
10 READ X,Y,Z
20 DATA 10,20,30
```

The above statements result in the same action as:

```
10 LET X = 10
15 LET Y = 20
20 LET Z = 30
```

RESTORE STATEMENT

The RESTORE statement is used to position the data pointer associated with the numeric data block to the beginning of that data block. The RESTORE statement has the following format:

```
n RESTORE
```

where n is the line number of the RESTORE statement.

All DATA statements contained in a program are combined into one numeric data block according to the order in which the DATA statements appear in that program. The RESTORE statement enables the data pointer associated with that data block to be reset from its current position to the first element in the block, so that subsequent READ statements can reread the data.

Example:

```
10 READ X,Y
20 RESTORE
30 READ Z
40 DATA 10,20,30
50 END
```

Execution of the above example causes the numeric constant 10 to be assigned to variables X and Z, and the numeric constant 20 to be assigned to variable Y.

INPUT STATEMENT

The INPUT statement is used to assign values from a card file labeled INPUT to the variables whose names are listed in the INPUT statement. The INPUT statement has the following format:

```
n INPUT <list of variable names>
```

where n is the line number of the INPUT statement and <list of variable names> is one or more numeric variables (either simple or subscripted) separated by commas.

The INPUT statement allows data values to be assigned to variable names during the time that the program is being executed. In a batch BASIC environment, the INPUT statement is used to obtain input from a card device or card reader during program execution. The data values to be entered into the program are obtained from a card file whose file-id is INPUT. In the absence of a DELIMIT statement referencing this file, individual data values are punched free-field, delimited by commas, in this card file. Individual punch cards in the INPUT card file must not contain line numbers, as any line numbers in the file will be treated as data elements.

Example:

```
10 INPUT A,B
20 END

?DATA INPUT
12.5,256
?END
```

Execution of the above example causes the numeric constants 12.5 and 256 from the card file labeled INPUT to be assigned to variables A and B, respectively.

PRINT STATEMENT

The PRINT statement is used to print output on the line printer during program execution. The PRINT statement has the following format:

```
n PRINT <list>
```

where n is the line number of the PRINT statement and <list> consists of one or more numeric constants, string constants, numeric variable names (either simple or subscripted), arithmetic expressions, or references to the TAB function separated by commas or semicolons.

The PRINT statement enables numeric and/or string data to be printed in either a zoned or packed format on the line printer during program execution. Zoned or packed output format is specified through use of commas or semicolons, respectively, as separators between data items in the PRINT statement list.

Zoned Format

In BASIC, the output line is considered to be divided into five print zones, each 15 spaces in width. Use of the comma to separate multiple items in the PRINT statement list causes the first item to be printed starting at the beginning of the first zone (print position 0), the second item at the next zone (print position 15), the third item at the next zone, etc. If more than five items are to be printed in the zoned format, the carriage is advanced to the first zone of the next output line when the fifth zone is used, and printing continues in this same manner until the PRINT statement list is exhausted. Upon completion of the PRINT statement list, the carriage is advanced to the beginning of the next output line in preparation for subsequent PRINT statements.

A comma encountered in a PRINT statement list causes the carriage to be advanced to the first position of the next print zone within the current output line or to the beginning of the first print zone on the next output line if the last zone of the current line has been used. Since each comma encountered in the PRINT statement list causes the carriage to be advanced to the beginning of the next print zone, successive commas may be used when skipping of print zones in the output line is desired. If the PRINT statement list ends in a comma, the carriage is merely advanced to the first position of the next print zone and not to the beginning of a new line unless the fifth zone of the current line has been used.

Examples:

```
10 LET A=B=4
20 PRINT "A+B=", A+B, "A*B=", 16
30 END
```

Execution of the above example causes the following output line to be printed on the line printer:

```
A+B=             8            A*B=            16
```

```
10 LET A=B=4
20 PRINT "A+B=", A+B,
```

```
30 LET C = 16
40 PRINT "A*B=", C
50 END
```

Execution of the above example causes the following output line to be printed on the line printer:

```
A+B=              8             A*B=             16
```

Packed Format

Use of the semicolon to separate multiple items in the PRINT statement list causes the output line to be printed in a more closely packed format. No spacing of the carriage is made in the output line other than those spaces automatically outputted when a numeric value is printed. If insufficient positions for another item remain in the output line after a semicolon is encountered in the PRINT statement list, the carriage is advanced to the next line.

Unlike zoned format where the output line is divided into specific print segments, packed format uses the length of the item to be printed to determine the length of a print segment. To determine what will be printed in packed format, it is necessary to understand how string constants and numeric values are printed by the PRINT statement.

String constants are printed by the PRINT statement just as they are declared in the program, with no leading or trailing spaces. Numeric values, however, are printed with a sign position preceding the number and a trailing space following the number. Negative values are preceded by a minus sign, and positive numbers are preceded with a blank sign position. The number of print positions occupied by a numeric value in the output line depends upon the magnitude of the number and whether or not it is an integer. Numeric values are printed in one of the following forms:

a. Integer form - a number containing 1 to 10 decimal digits printed without a decimal point.
b. Fractional form - a number containing 1 to 6 decimal digits printed with a decimal point. Trailing (right-most) zeros are not printed, and a number less than one is printed with a zero to the left of the decimal point.
c. Scientific form - a number expressed as a number greater than 1 and less than 10 printed in fractional form followed by a space, followed by the letter E, followed by a signed or unsigned 1- or 2-digit

integer, which is the power of ten that the number in fractional form preceding the letter E is to be raised.

Numeric values are printed by the PRINT statement in the above forms according to the following rules:

1. An integer whose absolute value is less than 1073741824 (i.e., 2**30) is printed in integer form.
2. An integer whose absolute value is greater than or equal to 1073741824 is rounded away from zero to six significant digits and printed in scientific form.
3. A number whose absolute value is less than 0.1 and which can be represented exactly by six or fewer digits is printed in fractional form.
4. A number whose absolute value is greater than or equal to 0.1 and less than 999999 is rounded away from zero to six significant digits and printed in fractional form.
5. A number whose absolute value is less than 0.1 and which cannot be represented exactly by six or fewer digits is rounded away from zero to six significant digits and printed in scientific form.
6. A number whose absolute value is greater than 999999 which is not an integer is rounded away from zero to six significant digits and printed in scientific form.

Example:

```
10 LET A=B=4
20 PRINT "A+B="; A+B; "A*B="; 16
30 END
```

Execution of the above example causes the following output line to be printed on the line printer:

```
A+B=  8  A*B= 16
```

If a PRINT statement list ends with a semicolon, the first element in the list of the next PRINT statement begins at the end of the current print segment.

Example:

```
10 LET A=B=4
20 PRINT "A+B="; A+B;
30 LET C=16
40 PRINT "A*B="; C
50 END
```

Execution of the above example causes the following output line to be printed on the line printer:

```
A+B=  8 A*B=  16
```

Vertical Spacing

Since completion of a PRINT statement advances the carriage to the beginning of the next print line, vertical spacing can be achieved through use of the PRINT statement consisting of only the word PRINT. If the previous PRINT statement did not end with a comma or semicolon, a blank line is left in the output.

Example:

```
100 PRINT "SALES FOR WEEK ENDING MARCH 8"
110 PRINT
120 PRINT "JONES", "SMITH", "MILLER"
130 END
```

Execution of the above example causes the following output to be printed on the line printer:

```
SALES FOR WEEK ENDING MARCY 8
JONES          SMITH          MILLER
```

Use of the PRINT statement consisting of only the word PRINT following a PRINT statement ending with a comma or a semicolon causes a partially filled output line to be printed on the line printer as illustrated by the following example.

Example:

```
10 FOR I = 1 TO 5
20 FOR J = 1 TO I
30 PRINT J;
40 NEXT J
50 PRINT
60 NEXT I
99 END
```

Execution of the above example causes the following output to be printed on the line printer:

```
1
1  2
1  2  3
1  2  3  4
1  2  3  4  5
```

TAB Function

The TAB function, TAB(X), may be used in the PRINT statement list to move the print mechanism to the position determined by the argument of the TAB. The letter X, shown as the argument of the TAB function, may be replaced by any numeric constant, numeric variable (either simple or subscripted), or arithmetic expression.

If a MARGIN statement referencing the line printer has not been used, the output line consists of print positions 0 through 74. Use of the comma in the PRINT statement list can be thought of as performing a tabulation to the next tab-stop; these stops being set at print positions 0, 15, 30, 45, and 60. The TAB function does not cause anything to be printed, but provides the ability to tab to any desired position within the output line.

The value of the argument, which may be any arithmetic expression, represents a print position in the output line. After the argument is evaluated, the integer part of its value is taken and treated modulo the current margin setting (75 in the absence of a MARGIN statement) to obtain a print position. The print mechanism is then moved to this position in the output line. If the tab position specified by the value of the argument is less than the current position of the print mechanism, then the TAB is ignored.

To achieve the desired tabulation, semicolons should be used to separate the items in a PRINT statement which is to be controlled by the TAB function.

Example:

```
10 PRINT A;TAB(20); B; TAB(40);C
```

Execution of the above PRINT statement causes an output line to be printed. This line contains the value of A starting in print position 0, the value of B starting in print position 20, and the value of C starting in print position 40.

SECTION 7

EXTERNAL FILES

GENERAL

One of the features of BASIC is the ability to READ or WRITE external data files. External data files reside on disk and can be read or written under program control. External files enable data to be stored separately from the program in which it is to be processed, thus permitting the construction of larger and more complex programs. In addition, external files enable a given set of data to be accessed by more than one program.

External data files in B 1700 BASIC are sequential EBCDIC files. These files are accessed in a sequential manner and contain data stored as a set of EBCDIC character codes. The statements pertaining to external files are discussed in this section. Through the remainder of this section, the term "files" should be understood to mean external data files.

FILES STATEMENT

The FILES statement is used to specify the maximum number of files to be open at any one time to the program during execution and, optionally, the file name of the files to be opened. The FILES statement has the following format:

n FILES <list of file names>

where n is the line number of the FILES statement, and <list of file names> is a list of 1 to 16 file names and/or asterisks separated by semicolons.

The FILES statement specifies the maximum number of files to be open to the program at any one time and reserves space in the program to access the indicated number of files. In B 1700 BASIC the FILES statement may contain from 1 to 16 file names, that is, during execution a maximum of 16 files may be open at any one time to a program.

An external file can be named in two different ways: It can have one name up to 10 characters long, or two names divided by a slash (/), each of which can be up to 10 characters long. External files may reside on a removable disk, as well as the system disk. When an external file resides on a removable disk, the disk cartridge or disk pack identifier (dp-id) must precede the file names

so that the system may locate the proper file. If a single name is used on a removable disk, it must be followed by a slash (/), with no spaces between the last character of the file-name and the slash. This trailing slash is required in order that the system can tell that the file is on a removable disk and has a single name. In summary, file names in B 1700 BASIC consist of one of the following forms:

a. identifier-1
b. identifier-2/identifier-3
c. dp-id/identifier-4/
d. dp-id/identifier-5/identifier-6

For additional information concerning the construction of file names refer to the B 1700 Software Operational Guide (Form 1057171).

Each file-name (or asterisk) entry in a FILES statement is referenced throughout a BASIC program by means of a number reflecting the position of the file-name entry within the FILES statement. In other words the first file-name entry is considered to be file one, the second file two; etc. An asterisk may be used instead of a file-name in the FILES statement. When this is done a file may be associated with the position occupied by the asterisk at a later point in the program through use of the FILE statement. The FILE statement is subsequently discussed in this section, under the heading of FILE STATEMENT.

If the desired list of file names cannot be contained in one FILES statement, then more than one FILES statement may be used in a program. A file-name may not appear more than once in the FILES statement(s) of a program.

Example:

```
10 FILES HOURS; ACCTPAC/PAYROLL/MASTERFILE
20 FILES ACCTPAC/PAYCHECKS/; *; TAXABLE/ITEMS
.
.
.
999 END
```

In the above file names, HOURS and TAXABLE/ITEMS are examples of a single-name file and a two-name file, respectively, both of which reside on the system disk. File names ACCTPAC/PAYCHECKS/ and ACCTPAC/PAYROLL/MASTERFILE are examples of a single-name file and a two-name file, respectively, both of which reside on a removable disk cartridge labeled ACCTPAC. In this case the disk cartridge identifier, ACCTPAC, must precede the file names so that the system may locate the files. In addition, a single-name file residing on a

removable disk must be followed by a slash (/), with no spaces between the last character of the file-name and the slash. This trailing slash is required in order that the system can tell that the file resides on a removable disk and has a single name.

File Designator

Each file-name (or asterisk) entry specified in a FILES statement in a given program is referenced in that program by means of a numeric argument called the file designator. The value of this numeric argument designates the position of a file-name (or asterisk) within the file name list of the FILES statement. When the file designator has a value of 1, the first file-name (or asterisk) entry appearing in the FILES statement is referenced. When the file designator has a value of 2, the second file-name (or asterisk) entry appearing in the FILES statement is referenced. When the file designator has a value of three, the third entry in the FILES statement is referenced, etc. If more than one FILES statement appears in a program, each file-name (or asterisk) entry is referenced in sequential order as if the file name list of each subsequent FILES statement were a continuation of the list from the prior FILES statement.

The file designator may be an integer constant, a numeric variable (either simple or subscripted), or an arithmetic expression. If a numeric variable or an arithmetic expression is used as a file designator, it will be evaluated and truncated to an integer before being used as the file designator.

If the value of the file designator is greater than the number of entries appearing in the FILES statement(s), program execution is terminated with an error message.

Example:

```
010 FILES FILE1; FILE2      'INPUT FILES
020 FILES FILE3; FILE4      'OUTPUT FILES
.
.
100 READ #1, A
.
.
190 LET F2=2
200 READ #F2,B
.
.
```

```
290 SCRATCH #F(I)+N
300 WRITE #F(I)+N,C
.
.
999 END
```

In the above example, the file READ statement at line 100 assigns a numeric value from the file named FILE1 to the variable A, because the value of the file designator in this file READ statement refers to the first file-name in the FILES statement. Likewise, the file READ statement at line 200 assigns a numeric value from the file named FILE2 to the variable B, because the value of the file designator refers to the second file-name in the FILES statement. At line number 300 the integral value of the file designator determines whether the value of variable C is written to FILE3 or FILE4.

When the value of the file designator is 0, the internal files discussed in section 6 are referenced. A file READ statement referencing a file designator of 0 will cause the system to search the program for data contained in DATA statements. A file INPUT statement referencing a file designator of 0 will cause the system to expect input from a card file labeled INPUT. A file output statement referencing a zero file designator will print output to the line printer.

FILE STATEMENT

The FILE statement is used to open or close a designated file. The FILE statement may have either of the following formats:

1. n FILE# <file designator>, "<file-name>"
2. n FILE# <file designator>, "*"

where n is the line number of the FILE statement, <file designator> is as described previously in this section, under the heading FILE DESIGNATOR, and <file-name> is the name of a file as described previously under the heading FILES STATEMENT. The punctuation between the <file designator> and the "<file-name>" or "*" may be either a comma or a colon.

In order for an external file to be accessed by a program the file must be open to the program. Those file names listed in the FILES statement are opened to the program and thus may be accessed. The FILE statement enables a file appearing in the file name list of a FILES statement to be closed and another file to be opened to the program in its place.

Example:

```
10 FILES FILEA;*;FILEC
20 READ #1, X
30 FILE #1, "FILE1"
40 READ #1, Y
50 FILE #2, "FILE2"
60 READ #2, Z
99 END
```

In the above example, execution of the FILE statement in line 30 causes FILEA to be closed and FILE1 to be opened to the program and associated with the file designator previously occupied by FILEA. In effect, the FILES statement in line 10 has now become

```
10 FILES FILE1; *; FILEC
```

Execution of the FILE statement in line 50 associates FILE2 with the file designator associated with the asterisk (*) in the FILES statement and opens this file to the program. At this point the FILES statement in line 10 has, in effect, become

```
10 FILES FILE1; FILE2; FILEC
```

Use of an asterisk (*) instead of a file-name in a FILE statement causes the designated file to be closed and the associated file designator to be invalidated to the program. Once a given file designator is invalidated, it remains invalidated to the program until a subsequent FILE statement associates a file-name with that file designator and thus validates it to the program.

Example:

```
10 FILES FILEX; FILEY
20 READ#2, A,B,C
30 FILE#2, "*"
40 READ#1, D,E
50 FILE#2, "FILEA"
60 READ#2, X,Y
70 END
```

In the above example, the FILE statement in line 30 causes FILEY to be closed and the associated file designator to be invalidated to the program. Following line 50 FILEA is associated with this file designator and may be accessed by the program.

As specified previously in this section in the discussion of the FILES statement, a maximum of 16 external files may be open at any one time to a program during execution. This does not imply that the maximum number of files that can be accessed by a program is limited to 16. Use of the FILE statement enables files to be closed and other file names to be associated with previously used file designators, thus permitting more than this number of files to be physically processed by a program.

FILE MODES

When an external data file is being processed by a program, it is either in the read mode or the write mode. In the read mode, only file READ or file INPUT statements may be executed on a file; likewise only file WRITE or file PRINT statements may be executed on a file in the write mode. A file is initially opened to a program in the read mode. In order to execute file WRITE statements, a file must first be placed in the write mode by execution of either a SCRATCH or an APPEND statement. If read statements are to be executed on a file while in the write mode, the file must first be placed in the read mode by means of either a RESTORE or a BACKSPACE statement. While in the write mode, a file may also be placed in the read mode by closing and re-opening the file through use of the FILE statement.

FILE READ STATEMENT

The file READ statement is used to assign values from the file specified by the file designator to the variables whose names are listed in the READ statement. The file READ statement has the following format:

n READ # <file designator>, <list of variable names>

where n is the line number of the file READ statement, <file designator> is as previously discussed in this section under the heading FILE DESIGNATOR, and <list of variable names> is a list of one or more numeric variables (either simple or subscripted) separated by commas. The punctuation between the <file designator> and the <list of variable names> may be a comma or a colon.

The file READ statement assumes that each line in the designated EBCDIC file begins with a line number. The line number, however, is not considered to be part of the data file. As elements are read from an EBCDIC file by the file READ statement, the first number present on each line is considered to be the line number of that line and is ignored. In addition, the file READ statement assumes that an EBCDIC file contains the standard delimiter, i.e., the comma.

If a file contains other than the standard delimiter, the delimiter contained in the file must be specified with a DELIMIT statement prior to the execution of file READ statements on that file.

The file READ statement is data item oriented. Execution of the file READ statement causes the number of data items required to satisfy the list of variable names associated with that READ statement to be read from the designated file. The first number on each line is considered to be a line number and is disregarded. Each time a file READ statement is executed, sufficient data items are read, beginning with the next available item, until the list of variable names is satisfied.

Example:

External file FILEA contains the following lines of data:

```
100 11,12,13,14
110 15,16,17
120 18,19
130 20
```

The following program accesses the above file:

```
10 FILES FILEA
20 READ#1, X,Y
30 READ#1, A,B,C
99 END
```

In this example the file READ statement at line 20 assigns the data values 11 and 12 to variables X and Y, respectively. After execution of line 20, the data pointer associated with FILEA points to data value 13. Execution of the file READ statement at line 30 assigns the data values 13, 14, and 15 to the variables A, B, and C, respectively, leaving the data pointer pointing to the data value 16 as the next available item in FILEA.

The file READ statement does not inform the user when the last data item has been accessed from a file, as is the case when the READ statement accesses the last data item from a DATA statement. When an external file is out of data, a value of zero (0) is assigned to each numeric variable encountered in all subsequent READ statements on that file. The IF END and IF MORE statements allow an end-of-data condition to be handled programmatically when external files are used.

Zero may be used as the file designator in a file READ statement. A zero file designator causes the file READ statement to read data items from DATA statements contained within the program, rather than from an external file.

FILE INPUT STATEMENT

The file INPUT statement is used to assign values from the file specified by the file designator to the variables whose names are listed in the INPUT statement. The file INPUT statement has the following format:

n INPUT # <file designator>, <list of variable names>

where n is the line number of the file INPUT statement, <file designator> is as described in this section, under the heading FILE DESIGNATOR, and <list of variable names> is a list of one or more numeric variables (either simple or subscripted) separated by commas. The punctuation between the <file designator> and the <list of variable names> may be a comma or a colon.

The file INPUT statement and the file READ statement serve the same purpose, but the manner in which elements are accessed from an EBCDIC file by these statements differs in two ways. The file READ statement considers the first number present on each line in the designated file to be a line number, but the file INPUT statement assumes that the specified file does not contain line numbers and treats any line numbers in the file as data items. In addition, the file READ statement is data item oriented, and the file INPUT statement is line oriented.

Execution of the file INPUT statement causes the number of elements required to satisfy the list of variable names associated with that INPUT statement to be accessed from the designated file. The first number on each line is not considered to be a line number and is accessed as a data element. If insufficient elements to satisfy the list of variable names reside on the current file line, elements are accessed from the next file line until each variable in the list has been assigned a value. When the list is satisfied, the remainder of the current line is ignored irrespective of whether or not more data elements are contained on that line, and the beginning of the following file line is considered to be the next available data element in that file.

Example:

External file FILEA contains the following lines of data:

```
100 11,12,13,14
110 15,16,17
120 18,19
130 20
```

The following program accesses the above file:

```
10 FILES FILEA
20 INPUT #1, X,Y
30 INPUT #1, A,B,C
99 END
```

In this example, the file INPUT statement at line 20 assigns the values 10011 and 12 to variables X and Y, respectively. After execution of statement 20, the data pointer associated with FILEA points to the first element on the next file line. Execution of the file INPUT statement at line 30 assigns the values 11015, 16, and 17 to the variables A, B, and C, respectively, advancing the data pointer associated with this file to the beginning of the next line.

The file INPUT statement assumes that the designated EBCDIC file contains the standard delimiter, i.e., the comma. If a file contains other than the standard delimiter, the delimiter contained in the file must be specified with a DELIMIT statement prior to the execution of file INPUT statements on that file.

The file INPUT statement does not inform the user when the last data item has been accessed from a file. When an EBCDIC file is out of data, a value of zero (0) is assigned to each numeric variable encountered in all subsequent INPUT statements on that file. The IF END or IF MORE statements allow an end-of-data condition to be handled programmatically when using external files.

Zero may be used as the value of the file designator in a file INPUT statement. When the file designator is zero, the file INPUT statement causes values to be obtained from a card file in the same manner as the INPUT statement. Refer to section 6, under the heading INPUT STATEMENT, for information concerning the INPUT statement.

FILE WRITE STATEMENT

The file WRITE statement is used to place the value of each listed item into the file specified by the file designator. The file WRITE statement has the following format:

n WRITE # <file designator>, <list>

where n is the line number of the file WRITE statement, <file designator> is as described under FILE DESIGNATOR and <list> consists of one or more numeric constants, string constants, numeric variable names (either simple or subscripted), arithmetic expressions, or references to the TAB function separated by commas or semicolons. The punctuation between the <file designator> and the <list> may be a comma or a colon.

The file WRITE statement writes the data items specified in the file WRITE statement list to the designated EBCDIC file. Prior to the execution of a file WRITE statement, the designated file must have been placed in the write mode by execution of a SCRATCH or an APPEND statement.

The file WRITE statement for EBCDIC files generates a line-numbered file. The line numbers begin with the number 00100, are incremented by 10, and are followed by a space. In the absence of a DELIMIT statement, the standard delimiter, i.e., the comma, is output immediately following each data item written to the file. If other than the standard delimiter is desired, then the delimiter to be used must be specified with a DELIMIT statement prior to the execution of any file WRITE statements on that file.

Each file WRITE statement for EBCDIC files generates one line of output unless the margin limit for the designated file is exceeded, or unless a comma or a semicolon follows the last item in the file WRITE statement list. When the margin limit is exceeded, output to the current file line is terminated, the next line number is generated, and output continues to the new file line. When the file WRITE statement list ends in a comma or a semicolon, subsequent file WRITE statements to that EBCDIC file continue on the current file line if space is available.

The zoned and packed format conventions discussed in section 6, under the PRINT statement apply when writing to an EBCDIC file. A comma following an item in the file WRITE statement list causes the next item to be placed in the file line at the beginning of the next 15-character zone; a semicolon suppresses this spacing. The file WRITE statement differs, however, from the PRINT statement in that each file output line begins with a line number and thus position zero (0) in the file output line is considered to be the second

character position following the last digit of the line number. As discussed in section 6, references to the TAB function may also be included in the file WRITE statement list, thus enabling output to be placed in any described position within the file line.

The character position used as the right margin limit in an EBCDIC file may be specified by means of the MARGIN statement. In the absence of a MARGIN statement, the margin limit is set at character position 75. If other than the standard margin limit is desired, then the margin limit to be used must be specified with a MARGIN statement prior to the execution of any file WRITE statements on that file.

Example:

```
10 FILES OUTPUT/FILEWRITE
20 LET A=B=4
30 SCRATCH #1
40 WRITE #1, "A+B=", A+B, "A*B=", 16
50 LET C=16
60 WRITE #1, "A+B="; A+B; "A*B="; C
70 END
```

After execution of the above example, the EBCDIC file OUTPUT/FILEWRITE contains the following:

```
00100 A+B=,           8 ,          A*B=,          16 ,
00110 A+B=, 8 ,A*B=, 16 ,
```

A file WRITE statement may not reference an external file following a file PRINT statement to that file.

Zero may be used as the value of the file designator in a file WRITE statement. In this case the list of items in the file WRITE statement is written to the line printer; however, no line numbers are supplied. When a zero file designator is referenced in a file WRITE statement, no prior SCRATCH statement referencing the zero file designator is required.

Example:

```
10 FOR I = 1 TO 5
20 WRITE #0,I;
30 NEXT I
40 END
```

Execution of the above example causes the following output to be printed on the line printer:

```
 1 , 2 , 3 , 4 , 5 ,
```

FILE PRINT STATEMENT

The file PRINT statement is used to place the value of each item listed in the file PRINT statement into the file specified by the file designator. The file produced by the file PRINT statement contains no line numbers and no delimiters. The file PRINT statement has the following format:

```
n PRINT # <file designator>, <list>
```

where n is the line number of the file PRINT statement, <file designator> is as described in this section under FILE DESIGNATOR, and <list> consists of one or more numeric constants, string constants, numeric variable names (either simple or subscripted), arithmetic expressions, or references to the TAB function separated by commas or semicolons. The punctuation between the <file designator> and the <list> may be a comma or a colon.

The file PRINT statement serves the same purpose as the file WRITE statement, except that the file PRINT statement generates an external file with no line numbers and no delimiters. The DELIMIT statement has no effect on PRINT statements which reference external files. Output from the file PRINT statement will be written to an external file with no line numbers and no delimiters regardless of previously executed DELIMIT statements referencing that file.

Prior to execution of a file PRINT statement, the designated file must be placed in write mode by means of the SCRATCH or the APPEND statement.

Each file PRINT statement generates one line of output unless the margin limit for the designated file is exceeded, or unless a comma or a semicolon follows the last item in the file PRINT statement list. When the margin limit is exceeded, the current file line is terminated, and output continues on the next file line. When the file PRINT statement list ends in a comma or a semicolon, subsequent file PRINT statements to that file continue on the current file line if space is available.

The zoned and packed format specifications discussed in section 6 under the PRINT statement apply to the file PRINT statement. A comma following an item in the file PRINT statement list causes the next item to be placed in the file line at the beginning of the next 15 character zone; a semicolon suppresses this spacing. As discussed in section 6, references to the TAB function may also be included in the file PRINT statement list, thus enabling output to be placed in any desired position within the file line.

The character position used as the right margin limit in an EBCDIC file may be specified by means of the MARGIN statement. In the absence of a MARGIN statement, the margin limit is set at character position 75. If other than the standard margin limit is desired, the margin limit to be used must be specified with a MARGIN statement prior to the execution of any file PRINT statements on that file.

The file PRINT statement may reference an external file following the execution of a file WRITE statement to that file. A file WRITE statement, however, may not reference an external file following a file PRINT statement to that file.

Example:

```
10 FILES OUTPUT/FILEPRINT
20 LET A=B=4
30 SCRATCH #1
40 WRITE #1, "A+B="; A+B; "A*B="; 16
50 PRINT #1, "A+B="; A+B; "A*B="; 16
99 END
```

After execution of the above example, the EBCDIC file OUTPUT/FILEPRINT contains the following:

```
00100 A+B=,  8  ,A*B=,  16  ,
A+B= 8 A*B=  16
```

Zero may be used as the file designator in the file PRINT statement. In this case the list of items in the file PRINT statement is written to the line printer. When a zero file designator is referenced in a file PRINT statement no prior SCRATCH statement referencing the zero file designator is required.

FILE MANIPULATION STATEMENTS

The following file manipulation statements provide additional capability when processing external files:

a. SCRATCH.
b. RESTORE.
c. IF END.
d. IF MORE.
e. APPEND.
f. DELIMIT.
g. MARGIN.
h. BACKSPACE.

SCRATCH Statement

The SCRATCH statement is used to erase the contents of an external file, position the data pointer to the beginning of the file, and place the file in write mode. The SCRATCH statement has the following format:

```
n SCRATCH #<file designator>
```

where n is the line number of the SCRATCH statement and <file designator> is as described in this section, under the heading FILE DESIGNATOR.

An EBCDIC file is opened to a program in the read mode when its name appears in a FILES statement, or when its name replaces the name of another file or asterisk in a FILE statement. Before any output statement may be executed with that file, it must be placed in the write mode by means of either the SCRATCH statement or the APPEND statement.

The SCRATCH statement, in addition to placing the designated EBCDIC file in the write mode, erases any existing data in the file and positions the data pointer for that file to the beginning of the file. In contrast, the APPEND statement, while placing an EBCDIC file in the write mode, does not affect any existing data in the file, but merely positions the data pointer after the last data item in the file.

The SCRATCH statement may not reference a file designator of zero.

Example:

```
10 FILES OUTPUT/FILE1
20 SCRATCH #1
30 WRITE #1, A, B, C
50 END
```

RESTORE Statement

The RESTORE statement is used to position the data pointer of the designated EBCDIC file from its current position to the beginning of the file. In addition, the RESTORE statement places the designated file in the read mode. The RESTORE statement has the following format:

```
n RESTORE #<file designator>
```

where n is the line number of the RESTORE statement and <file designator> is as described in this section under the heading FILE DESIGNATOR.

Once an EBCDIC file has been placed in the write mode, a RESTORE statement (or a BACKSPACE statement) must be executed with that file before it may be accessed by a file READ or a file INPUT statement. Execution of a RESTORE

statement places the designated EBCDIC file in the read mode and returns the data pointer from its current position to the beginning of the file.

Example:

External file FILE/INPUT contains the following lines of data:

```
100 10,11,12
110 13,14
120 15
```

The following program accesses the above file:

```
10 FILES FILE/INPUT; FILE/OUTPUT
20 READ #1, A, B, C, D, E, F
30 SCRATCH #2
40 WRITE #2, A;B;C;D;E;F
50 RESTORE #1
60 READ #1, A, B, C, D, E, F
70 WRITE #2, A;B;C;D;E;F
80 END
```

After execution of the above program the file FILE/OUTPUT contains the following:

```
00100  10 , 11 , 12 , 13 , 14 , 15 ,
00110  10 , 11 , 12 , 13 , 14 , 15 ,
```

IF END Statement

The IF END statement is used to check for an end-of-file condition when reading from an EBCDIC file, or to check for an end-of-file space condition when writing to an EBCDIC file. The IF END statement has the following format:

```
n IF END # <file designator> THEN <line number>
```

where n is the line number of the IF END statement, <file designator> is as described under the heading FILE DESIGNATOR, and <line number> is the line number of the statement to which program control is to be transferred when the tested condition is true. The punctuation between the <file designator> and the word THEN may be a space, a comma, or a colon.

When reading from an EBCDIC file, neither the file READ statement nor the file INPUT statement informs the user when an attempt has been made to read beyond the end of the designated file. If there are no more data items in an EBCDIC file, the value zero (0) is automatically assigned to all numeric variables encountered in subsequent file READ or file INPUT statements referencing that

file. The IF END statement enables the user to check the last file READ or file INPUT statement executed on the designated file for the occurrence of an end-of-file condition.

When an end of file condition has been encountered by the last file READ or file INPUT statement executed on the designated file, program control is transferred by the IF END statement to the specified line number; otherwise, program control continues with the next statement following the IF END statement. It should be noted that the IF END statement does not test the designated file directly; instead it checks the last file READ or file INPUT statement executed with the designated file for the occurrence of an end-of-file condition. For this reason the IF END statement, to be used effectively, must be executed after a read operation as one additional file READ or file INPUT statement will be executed after all data elements have been accessed from a designated file.

Example 1:

External file FILEA contains the following lines of data:

```
100 10,11,12
110 13,14,15
```

The following program reads the above file:

```
10 FILES FILEA
20 READ #1, X, Y, Z
30 PRINT "AT LINE 20 X, Y, Z =";X;Y;Z
40 READ #1, X, Y, Z
50 PRINT "AT LINE 40 X, Y, Z =";X;Y;Z
60 READ #1, X, Y, Z
70 PRINT "AT LINE 60 X, Y, Z =";X;Y;Z
```

Execution of the above program prints the following output on the line printer:

```
AT LINE 20 X, Y, Z = 10  11  12
AT LINE 40 X, Y, Z = 13  14  15
AT LINE 60 X, Y, Z = 0  0  0
```

In order to test for an end-of-file condition when reading from FILEA, IF END statements are placed in the above program as follows:

```
10 FILES FILEA
20 READ #1, X,Y,Z
25 IF END #1 THEN 75
30 PRINT "AT LINE 20 X, Y, Z =";X;Y;Z
40 READ #1, X, Y, Z
45 IF END #1 THEN 75
50 PRINT "AT LINE 40 X, Y, Z =";X;Y;Z
60 READ #1, X, Y, Z
65 IF END #1 THEN 75
70 PRINT "AT LINE 60 X, Y, Z =";X;Y;Z
75 PRINT "END OF FILE - FILEA"
80 END
```

Execution of the above program prints the following output on the line printer:

```
AT LINE 20 X, Y, Z = 10  11  12
AT LINE 40 X, Y, Z = 13  14  15
END OF FILE - FILEA
```

Example 2:

The following program uses the file, FILEA, from the previous example as input.

```
10 FILES FILEA
20 READ #1, A, B, C
30 PRINT "AT LINE 20 A, B, C =";A;B;C
40 READ #1, A, B, C, D
50 PRINT "AT LINE 40 A, B, C, D =";A;B;C;D
60 END
```

Execution of the above program prints the following output on the line printer:

```
AT LINE 20 A, B, C = 10  11  12
AT LINE 40 A, B, C, D = 13  14  15  0
```

In order to test for an end-of-file condition when reading from FILEA, IF END statements are placed in the above program as follows:

```
10 FILES FILEA
20 READ #1, A, B, C
25 IF END #1 THEN 60
30 PRINT "AT LINE 20 A, B, C =";A;B;C
40 READ #1, A, B, C, D
45 IF END #1 THEN 60
50 PRINT "AT LINE 40 A, B, C, D =";A;B;C;D
60 END
```

Execution of the above program prints the following output on the line printer:

```
AT LINE 20 A, B, C =  10   11   12
```

When writing to an EBCDIC file in B 1700 BASIC, additional areas of file space are allocated to a designated file as data items are written to that file. For this reason it is impractical to test for an end-of-file space condition when writing an EBCDIC file.

The IF END statement may not be used with a zero file designator.

IF MORE Statement

The IF MORE statement is used to check for more data when reading from an EBCDIC file or to check for more file space when writing to an EBCDIC file. The IF MORE statement has the following format:

n IF MORE # <file designator> THEN <line number>

where n is the line number of the IF MORE statement, <file designator> is as described under the heading FILE DESIGNATOR, and <line number> is the line number of the statement to which program control is to be transferred when the tested condition is true. The punctuation between the <file designator> and the word THEN may be a space, a comma, or a colon.

When reading from an EBCDIC file, neither the file READ statement nor the file INPUT statement informs the user when an attempt has been made to read beyond the end of the designated file. If there are no more data items in an EBCDIC file, the value zero (0) is automatically assigned to all numeric variables encountered in subsequent file READ or file INPUT statements referencing that file. The IF MORE statement enables the user to programmatically check a designated EBCDIC file for the presence of more data.

The IF MORE statement tests the designated file to determine whether or not another data item remains in the file. If a data item remains in the

designated file, program control is transferred to the specified line number; otherwise, program control continues with the next statement following the IF MORE statement. In contrast to the IF END statement, it should be noted that the IF MORE statement checks the designated file directly for the presence of additional data. For this reason the IF MORE statement should be executed prior to the file READ or file INPUT statement referencing the designated file.

Example 1:

External file FILEA contains the following lines of data:

```
100 10,11,12
110 13,14,15
```

The following program reads the above file:

```
10 FILES FILEA
20 IF MORE #1 THEN 40
30 GO TO 70
40 READ #1, X,Y,Z
50 PRINT "AT LINE 40 X, Y, Z =";X;Y;Z
60 GO TO 20
70 PRINT "END OF FILE - FILEA"
80 END
```

Execution of the above program prints the following output on the line printer:

```
AT LINE 40 X, Y, Z = 10  11  12
AT LINE 40 X, Y, Z = 13  14  15
END OF FILE - FILEA
```

Example 2:

The following program uses the file, FILEA, from the previous example as input.

```
10 FILES FILEA
20 IF MORE #1 THEN 40
30 GO TO 140
40 READ #1, A, B, C
50 PRINT "AT LINE 40 A, B, C =";A;B;C
60 IF MORE #1 THEN 80
70 GO TO 140
80 READ #1, A, B, C, D
90 PRINT "AT LINE 80 A, B, C, D =";A;B;C;D
100 IF MORE #1 THEN 120
```

```
110 GO TO 140
120 READ #1, A, B, C, D, E
130 PRINT "AT LINE 120 A, B, C, D, E =";A;B;C;D;E
140 END
```

Execution of the above program prints the following output on the line printer:

```
AT LINE 40 A, B, C = 10  11  12
AT LINE 80 A, B, C, D = 13  14  15  0
```

When writing to an EBCDIC file in B 1700 BASIC, additional areas of file space are allocated to a designated file as data items are wirtten to that file. For this reason it is impractical to test for the availability of more file space when writing an EBCDIC file.

The IF MORE statement may not be used with a zero file designator.

APPEND Statement

The APPEND statement is used to place an EBCDIC file in the WRITE mode and to position the data pointer associated with that file after the last data item in the file. The APPEND statement has the following format:

n APPEND # <file designator>

where n is the line number of the APPEND statement and <file designator> is as described in this section under FILE DESIGNATOR.

The APPEND statement enables data to be added to an EBCDIC file. Both the SCRATCH statement and the APPEND statement place the designated file in the write mode. When a file is scratched, the file's data pointer is positioned to the beginning of the file and all data previously contained in the file are lost. The APPEND statement positions the data pointer associated with the designated file immediately following the last data item in that file, thus enabling a file to be placed in the write mode without losing previously stored data. If the file contains no data the APPEND statement positions the data pointer at the beginning of the file.

Example:

External file EXAMPLE/APPEND contains the following data:

```
00100  1 , 2 , 3 , 4 ,
```

The following program appends data to the above file:

```
10 FILES EXAMPLE/APPEND
20 APPEND #1
30 FOR I = 5 TO 8
40 WRITE #1, I;
50 NEXT I
60 END
```

After execution of the above program file EXAMPLE/APPEND contains the following:

```
00100  1 , 2 , 3 , 4 ,
00110  5 , 6 , 7 , 8 ,
```

DELIMIT Statement

The purpose of the DELIMIT statement is to specify the delimiter, other than the standard file delimiter (i.e., the comma), to be used with a designated EBCDIC file. The DELIMIT statement for EBCDIC files has the following format:

n DELIMIT #<file designator>, (<delimiter>)

where n is the line number of the DELIMIT statement, <file designator> is as described for EBCDIC files in this section under FILE DESIGNATOR, and <delimiter> is a single character to be used as the delimiter with the designated EBCDIC file. The punctuation between the <file designator> and the left parenthesis may be a comma or a colon.

If a file contains a delimiter other than the standard comma, the non-standard delimiter contained in the file must be specified with a DELIMIT statement prior to the execution of any file READ and/or file INPUT statements with that file.

If a delimiter other than the standard comma is to be written to a file, the delimiter to be used must be specified with a DELIMIT statement prior to the execution of any file WRITE statements with that file. The DELIMIT statement has no effect on the file PRINT statement. Output from the file PRINT statement will be written to an external file with no line numbers and no delimiters, regardless of previously executed DELIMIT statements referencing that file.

The DELIMIT statement may be used with a zero file designator to specify the use of a non-standard delimiter in input from the card reader and output to the line printer. Use of a zero file designator in the DELIMIT statement has no effect on the DATA statement. DATA statements contained in a program must use the standard delimiter.

Example:

External file FILEIN contains the following lines of data in which a blank is used instead of the standard delimiter:

```
100 11  12  13  14
110 15  16  17  18
120 19  20
```

The following program reads the above file:

```
10 FILES FILEIN; FILEOUT
20 DELIMIT #1, ( )
30 READ #1, A, B, C, D
40 SCRATCH #2
50 DELIMIT #2, (@)
60 WRITE #2, A;B;C;D
70 END
```

After execution of the above program the file FILEOUT contains the following:

```
00100  11 @ 12 @ 13 @ 14 @
```

MARGIN Statement

The purpose of the MARGIN statement is to specify the character position to be used as the right margin limit for the designated EBCDIC file. The MARGIN statement for EBCDIC files has the following format:

n MARGIN # <file designator>, <expression>

where n is the line number of the MARGIN statement, <file designator> is as described for EBCDIC files, under FILE DESIGNATOR, and <expression> is a numeric constant, a numeric variable (either simple or subscripted), or an arithmetic expression. The punctuation between the <file designator> and the <expression> may be a comma or a colon.

In the absence of a MARGIN statement, the right margin limit is set at character position 75. If the right margin in an EBCDIC file is other than character position 75, the non-standard margin must be specified in a MARGIN statement prior to the execution of file READ or file INPUT statements with that file.

If the value of the expression used in the MARGIN statement is not an integer, it is truncated and the integer part is used to specify the character position of the right margin. Character position 158 is the maximum value to which the

margin may be set. If the value of the expression is less than 1 or greater than 158, the margin is set at character position 158.

BACKSPACE Statement

The BACKSPACE statement is used to backspace the data pointer associated with a designated EBCDIC file and to place the file in the read mode. The BACKSPACE statement for EBCDIC files has the following format:

n BACKSPACE # <file designator>

where n is the line number of the BACKSPACE statement and <file designator> is as described for EBCDIC files in this section, under the FILE DESIGNATOR heading.

The manner in which the data pointer is backspaced depends upon the last statement that was executed with the designated EBCDIC file prior to execution of the BACKSPACE statement. If the last statement to access the file was a file INPUT statement or a file PRINT statement, execution of a BACKSPACE statement moves the data pointer to the beginning of the current line. If the data pointer is presently at the beginning of a line, each execution of a BACKSPACE statement moves the data pointer to the beginning of the previous line.

If the last statement to access the designated file was a file READ or file WRITE statement, each execution of a BACKSPACE statement moves the data pointer back over one delimiter to the previous data item. If the data pointer is presently at the first data item on a line, execution of the BACKSPACE statement moves the pointer to the last item on the previous line.

When the BACKSPACE statement is executed with the data pointer positioned at the beginning of the designated file, the pointer is moved to the end of the first line in the file. Backspacing past the beginning of the file proceeds as described above utilizing the first line of the file for subsequent executions of the BACKSPACE statement.

FILE FUNCTIONS

File functions are provided in BASIC to give additional capability when using EBCDIC files. Each file function requires a file designator as an argument and returns a number as the function value.

HPS(#N) Function

The HPS(#N) function returns the location of the file pointer in the current line of the designated EBCDIC file. The letter N, shown as the argument,

represents a file designator as described in this section, under the heading FILE DESIGNATOR.

LIN(#N) Function

The LIN(#N) function returns the line number of the current line in the designated EBCDIC file. The letter N, shown as the argument, represents a file designator as described in this section, under the heading FILE DESIGNATOR.

VPS(#N) Function

The VPS(#N) function returns the number of lines which have been read from or written to the designated EBCDIC file. The letter N, shown as the argument, represents a file designator as described in this section, under the heading FILE DESIGNATOR.

SECTION 8

INTRINSIC FUNCTIONS

GENERAL

Certain functions are provided in the BASIC language. The names of these functions are known to the compiler and need only be referenced in order to be used.

MATHEMATICAL FUNCTIONS

In addition to the arithmetic operators described in section 1 under the heading ARITHMETIC EXPRESSIONS, BASIC has the capability to evaluate the following mathematical functions. The letter X, shown as the argument of each function, can be replaced by any numeric constant, numeric variable (either simple or subscripted), or arithmetic expression.

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).
COT(X)	Cotangent of X(X in radians).
ATN(X)	Arctangent of X, expressed in radians.
ABS(X)	Absolute value of X.
EXP(X)	Natural exponential of X.
LOG(X)	Natural logarithm of X.
SQR(X)	Square root of X.

INT(X) Function

The integer function, INT(X), returns the greatest integer algebraically less than or equal to the argument. As an example, the value of INT(7.5) is 7; the value of INT(-7.5) is -8.

SGN(X) Function

The sign function, SGN(X), determines the sign of the argument and returns the value of 0, +1, or -1, depending on whether the value of the argument is 0, positive and non-zero, or negative and non-zero, respectively. As an example, the value of SGN(-5) is -1; the value of SGN(0) is 0.

MOD(X,Y) Function

The modular function, MOD(X,Y), returns the value of X-INT(X/Y)*Y. Each of the arguments X and Y may be replaced by any numeric constant, numeric variable (either simple or subscripted), or arithmetic expression.

RND Function

The random function, RND, is used to generate a sequence of pseudo-random numbers between 0 and 1. The RND function does not require an argument.

The first reference to RND executed in a BASIC program returns the same number as the first in the sequence of random numbers. Repeated execution of RND always generates the same set of pseudo-random numbers.

RANDOMIZE STATEMENT

The random function, RND, always generates the same set of random numbers. While this can be very useful, particularly during the debugging of a BASIC program, the capability to generate different sets of random numbers is often required. The RANDOMIZE statement, when included in a BASIC program that references the RND function, causes the RND function to produce different sets of random numbers.

The RANDOMIZE statement may be abbreviated as RAN.

Example:

```
10 RAN
20 FOR I = 1 TO 10
30 Z(I) = RND
40 NEXT I
50 STOP
60 END
```

Each time the program in the above example is executed, a different set of random numbers will be assigned to the array Z. If line number 10, that is, the RANDOMIZE statement, is removed from the program, the first 10 values from the standard set of random numbers will be assigned to array Z each time the program is executed.

SECTION 9

SUBPROGRAMS

GENERAL

Subprograms allow a particular calculation or a given routine to be coded once within a program and then referenced repeatedly throughout the program. Subprograms in BASIC are of two types:

a. Function subprograms.

b. Subroutine subprograms.

FUNCTION SUBPROGRAMS

In addition to the standard functions provided by the BASIC compiler, the user may define functions within his program with the DEF statement. User-defined functions may have two forms -- single statement functions or multiple statement functions.

Single Statement Functions

A single statement function, as its name suggests, is a function that can be defined by a single DEF statement. A single statement function is useful when a particular arithmetic expression must be evaluated repeatedly within a program for different values of the arguments involved. The single statement function may have either of the following formats:

1. n DEF FN<letter> = <arithmetic expression>
2. n DEF FN<letter> (<argument list>) = <arithmetic expression>

where n is the line number of the DEF statement; <letter> is any single alphabetic letter; <arithmetic expression> is any arithmetic expression; and, in option 2., <argument list> is a list of one to seven unsubscripted dummy variable names separated by commas.

The name of a defined function must consist of three letters, the first two of which must be the letters FN. In the general form, <letter> completes the function name and distinguishes a given function from other functions which may be defined in the same program. In a given program each user-defined function must have a unique name. Since defined functions are given

three-character names, the first two of which are always FN, as many as 26 unique functions may be defined. Valid function names are FNA, FNB, FNC, ...,FNZ.

The <argument list> consists of one to seven dummy variables, separated by commas, each of which must be a distinct unsubscripted variable name. Each variable appearing in the <argument list> is replaced by the value of the corresponding actual argument when the function is referenced. Dummy arguments merely reserve a place for the actual arguments, and thus are undefined outside of the function definition. A dummy variable is not related to any variable of the same name appearing elsewhere in the program.

The <arithmetic expression> on the right side of the equal sign may be any arithmetic expression that can be placed on the remainder of that statement line. Aside from the dummy variables the <arithmetic expression> may contain any combination of variable names (either simple or subscripted) appearing elsewhere in the program, references to other functions (including functions previously or subsequently defined by other DEF statements), and numeric constants. Variable names not listed in the <argument list> use their current values assigned elsewhere in the program.

The DEF statement defining a single statement function may be placed at any point within a BASIC program. The function definition is only evaluated when a reference to that function is executed. Execution of a function reference in an expression results in an association of the actual argument values with the corresponding dummy arguments in the function definition and a subsequent evaluation of the function definition. If a function reference or an arithmetic expression is used as an actual argument, then these quantities are evaluated before the association can take place. After the evaluation of the function definition, the resultant value assigned to the function name then replaces the function reference in the expression.

Examples:

```
10 DEF FNA(X,Y) = X**2 + 2*X*Y + Y**2
20 DEF FNB(X,Y,X1,Y1) = FNA(X,Y)/FNA(X1,Y1)
30 DEF FNZ(A,B) = X * SIN(A) + 4*INT(A*B-Z)
```

Multiple Statement Functions

The single statement function is limited to those functions which can be expressed in a single DEF statement. The multiple statement function is not restricted to a single statement and is written in the following manner--

The first statement of the multiple statement function is a DEF statement and may have either of the following formats:

1. n DEF FN<letter>
2. n DEF FN<letter> (<argument list>)

where n is the line number of the DEF statement; <letter> is any single alphabetic letter; and, in option 2, <argument list> is a list of one to seven unsubscripted dummy variable names separated by commas.

Following the DEF statement in a multiple statement function are those statements which define the function. These statements must be followed by an FNEND statement. The FNEND statement indicates the end of the function definition and has the following format:

n FNEND

where n is the line number of the FNEND statement.

As discussed in this section under SINGLE STATEMENT FUNCTIONS, the name of a defined function must consist of three letters, the first two of which must be FN. In the general form, <letter> completes the function name, thus enabling a maximum of 26 functions (either single statement and/or multiple statement) to be defined in the same program. Valid user-defined function names are FNA, FNB, FNC,...FNZ.

The absence of the equals sign and the arithmetic expression in a DEF statement indicates that a multiple statement function follows the DEF statement. A multiple statement function may consist of as many statements as desired; the end of the function definition being indicated by the FNEND statement.

Multiple statement functions may not be nested. In addition, transfer of program control from within a multiple statement function to some point in the program outside the function, and vice-versa, is not permitted.

As with single statement functions, dummy variables appearing in the <argument list> represent the corresponding actual arguments which are substituted for the dummy arguments when the function is referenced. In addition to the dummy variables the statements appearing in the multiple statement function definition may contain any combination of variable names (either simple or subscripted) appearing elsewhere in the program, references to other functions (including single statement functions defined within the program), and numeric constants. Variable names not listed in the <argument list> use their current values assigned elsewhere in the program.

When using a multiple statement function, the function name,<FN letter>, should be assigned a value before the FNEND statement is encountered. If the function name has not been assigned a value when control reaches the FNEND statement, zero is returned for the function value.

The multiple statement function may be placed at any point within a BASIC program. The function definition is only evaluated when a reference to that function is executed.

Example:

```
010 REM     COMPUTE THE LARGEST FACTOR FOR
020 REM     THE ODD NUMBERS 1001 TO 1019.
030 REM
040 PRINT "NUMBER", "LARGEST FACTOR"
050 FOR I=1 TO 10
060 READ  N
070 PRINT N, FNF(N)
080 NEXT I
100     DEF FNF(N)
110     FOR F = INT(N/2) TO 1 STEP -1
120     IF N/F <> INT(N/F) THEN 150
130     LET FNF=F
140     GO TO 160
150     NEXT F
160     FNEND
170 DATA 1001, 1003, 1005, 1007, 1009
180 DATA 1011, 1013, 1015, 1017, 1019
190 END
```

SUBROUTINE SUBPROGRAMS

A subroutine enables a repetitive calculation that produces more than one value to be coded once as a subprogram and then referenced as needed throughout the program. Program control is transferred to a subroutine through the GOSUB statement. The GOSUB statement has the following format:

```
n GOSUB <line number>
```

where n is the line number of the GOSUB statement, and <line number> is the line number of the first statement in the subroutine.

A subroutine is called using the GOSUB statement. The RETURN statement is used to exit the subroutine and return program control to the statement immediately following the calling GOSUB statement. The RETURN statement has the following format:

```
n RETURN
```

where n is the line number of the RETURN statement.

A GOSUB statement may be used inside a subroutine to call another subroutine. When subroutines are nested, the first RETURN statement to be executed returns control to the statement following the most recently executed GOSUB. The next RETURN statement returns control to the statement following the GOSUB statement which was previously executed, and so on. Execution of a RETURN statement prior to the execution of a GOSUB statement terminates program execution with an error message.

A subroutine may contain more than one RETURN statement.

Example:

```
100 INPUT X,Y
110 IF X>= 0 THEN 140
120 LET A = B = 0
130 GO TO 160
140 GOSUB 400
150 LET B = A/X
160 PRINT A,B
170 INPUT Z
180 GOSUB 420
190 PRINT A
200 STOP
400 REM   SUBROUTINE TO CALCULATE Z, U, AND A
410 LET Z = SQR(X)
420 LET U = Y - Z
430 LET A = SQR(U*U+1)
440 RETURN   'RETURN TO LINE 150 OR 190
999 END
```

APPENDIX A

BASIC CARD READER INPUT

GENERAL

The BASIC Compiler, in conjunction with the Master Control Program, enables source programs to be compiled through use of a card reader or a card device. Compilation of the BASIC source language input is achieved by presenting the compilation card deck to the MCP. Control cards included in the compilation deck are of two general types -- MCP control cards and compiler option control cards ($ cards). The structure of the BASIC compilation deck is discussed in the text that follows.

COMPILATION CARD DECK

The entities comprising the structure of the BASIC compilation deck and the order of their occurrence are as follows:

a. COMPILE Card.
b. Label Equation Card (optional).
c. MCP Label Card.
d. Compiler option control card (Optional).
e. Source input cards.
f. END (end-of-file) Card.

MCP control cards are made distinguishable from other cards by an invalid character in column 1, for 80-column cards, or by a valid question mark (?), for 96-column cards. An invalid character is represented by a ? for clarity in this manual. MCP control information is punched in a free-form format in columns 2 through 72. The presence of a period in a MCP control card terminates the control information on that card and any information following the period is treated as a comment by the MCP. Refer to the B 1700 Systems Software Operational Guide (form 1057171) for further information regarding MCP control cards.

COMPILE Card

The COMPILE Card instructs the MCP to compile the indicated program-name with BASIC using one of the following options:

a. ?COMPILE program-name BASIC

   This option causes the source program to be compiled and executed (i.e., compile and go). The resultant object program is not entered in the disk directory.

b. ?COMPILE program-name BASIC LIBRARY

   This option causes the source program to be compiled and the resultant object program to be entered in the disk directory with the identifier program-name for future execution.

c. ?COMPILE program-name BASIC SAVE

   This option causes the source program to be compiled and the resultant object program to be entered in the disk directory and then executed (i.e., a combination of a. and b. above).

d. ?COMPILE program-name BASIC SYNTAX

   This option causes the source program to be compiled only for a syntax check.

In the absence of this control card the system operator may manually execute one of the COMPILE options through the console printer by keying in the appropriate message.

Label Equation Card

The Label Equation Card optionally may be included in the compilation deck to change a compiler file-name in order to avoid duplication of file-names when operating in a multiprogramming environment. If used, the Label Equation Card (or cards) must immediately follow the COMPILE Card and, precede the MCP Label Card.

The general form of the Label Equation Card is:

```
?FILE internal-file-name file-attribute-1 [file-attribute-2 ...]
```

The BASIC Compilers internal file-names and external file-identifiers for use in label equation are as follows:

Internal File-Name	External File-Id	Description
CARDS	CARDS	Input file from the card reader.
LINE	LINE	Compilation output listing to the line printer.

MCP Label Card

The MCP Label Card informs the MCP which type of input card code to expect and provides the file-id of the card file.

The MCP Label Card is coded as follows:

```
?DATA CARDS
```

The word DATA specifies to the MCP that the card input is punched in standard EBCDIC mode for 80-column devices, and in BCD mode for 96-column devices.

Compiler Option Control Card ($ CARD)

The BASIC Compiler Option Control Card ($ sign following the line number), optionally, may be included in the compilation deck. This control card, if used, notifies the compiler as to which options are required during the compilation. When this card is omitted, $ CARD LIST SINGLE is assumed. There must be at least one space between each option specified on a $-card; however, the options may be listed in any order. Any number of $-cards may be used and may appear anywhere in the source deck. The options specified will become either active or inactive from that point on. The format of the BASIC Compiler Option Control Card is as follows:

```
line-number $ option option ...
```

The options which may be specified on the Compiler Option Control Card are as follows:

a. CARD

   Symbolic input is from source language cards. This option is for documentation purposes only.

b. LIST

Creates a compilation output listing of the source language input, with error and/or warning messages, where required. LIST is the default option and thus need not be specified.

c. SINGLE

Causes the compilation output listing to be printed in a single-spaced format. SINGLE is the default option and need not be specified.

d. DOUBLE

Causes the compilation output listing to be printed in a double-spaced format.

e. CODE

Lists the object code generated for a source statement following that source statement from the point of insertion in the compilation deck.

f. NO

Each of the above options may be preceded with NO, thus enabling options to be turned on for selected program parts and then turned off as desired. When an option is preceded by NO, there must be at least one space between the word NO and the option to be terminated.

Source Input Cards

These cards are the statements comprising the source program. When using BASIC with the card reader, each card is taken as a different line and must contain only one statement. Each card between the ?DATA card and the ?END card must contain a line number. A line number starts in column 1 and contains 1 to 5 digits; it is terminated by a non-numeric character. The line number is used as both a statement label and a sequence number. Each card is sequence checked as it is read. When using BASIC through the card reader, the INPUT statement causes data to be read from a card file labeled INPUT. Likewise, the PRINT statement causes output to be written to the line printer.

END CARD

The END Card designates the end-of-file for the compilation deck to the MCP.

The END Card is coded as follows:

```
?END
```

The END Card must be the last card in the compilation deck.

SAMPLE COMPILATION DECK

In the following example, a BASIC program is to be compiled and executed from the card reader. A $-card is enclosed in the compilation deck to cause the compilation output listing to be printed in a double-spaced format. The options CARD and LIST are not required but are included for documentation purposes only. The card file labeled INPUT, following the compilation deck, will be required during execution of the resultant object program.

Example:

```
?COMPILE PROGRAM/GCD WITH BASIC
?DATA CARDS
100 $CARD LIST DOUBLE
110 REM     CALCULATE THE GREATEST COMMON
120 REM     DIVISOR OF THREE NUMBERS
130 REM     X, Y, AND Z
140 DEF FNG (A,B)
150 LET R = A-INT(A/B)*B
160 IF R = 0 THEN 200
170 LET A = B
180 LET B = R
190 GO TO 150
200 LET FNG = B
210 FNEND
220 INPUT X,Y,Z
230 LET G = FNG(X,Y)
240 LET G = FNG(G,Z)
250 PRINT "X", "Y", "Z", "GCD"
260 PRINT X,Y,Z,G
270 END
?END
?DATA INPUT
12,32,56
?END
```

BURROUGHS CORPORATION
DATA PROCESSING PUBLICATIONS
REMARKS FORM

TITLE: B 1700 SYSTEMS
BASIC
REFERENCE MANUAL

FORM: 1067535
DATE: 6-73

CHECK TYPE OF SUGGESTION:

☐ ADDITION ☐ DELETION ☐ REVISION ☐ ERROR

GENERAL COMMENTS AND/OR SUGGESTIONS FOR IMPROVEMENT OF PUBLICATION:

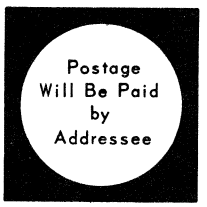
FROM: NAME _____
TITLE _____
COMPANY _____
ADDRESS _____
_____

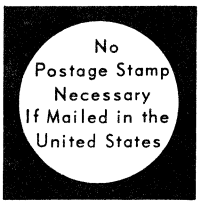
DATE _____

cut along dotted line

STAPLE

FOLD DOWN SECOND FOLD DOWN





BUSINESS REPLY MAIL
First Class Permit No. 817, Detroit, Mich. 48232

Burroughs Corporation
Burroughs Place
Detroit, Michigan 48232

attn: Systems Documentation
Technical Information Organization, TIC-Central

FOLD UP FIRST FOLD UP

Wherever There's
Business There's

Burroughs

1067535

6-73

Printed in U. S. America