Burroughs

B 1800/B 1700 Systems

BASIC

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. May 1979 1067535

Burroughs

B 1800/B 1700 Systems

BASIC

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. May 1979 1067535

Burroughs believes that the software described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, can be accepted for any consequences arising out of the use of this material, including loss of profit, indirect, special, or consequential damages. There are no warranties which extend beyond the program specification.

The Customer should exercise care to assure that use of the software will be in full compliance with laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change. Revisions may be issued from time to time to advise of changes and/or additions.

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P. O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO – West.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

INTRODUCTION

The purpose of this manual is to provide a description of the BASIC language as implemented on the B 1800/B 1700 systems. BASIC, an acronym for Beginners All-purpose Symbolic Instruction Code, was initially developed under the direction of Professor Kemeny and Professor Kurtz at Dartmouth College. B 1800/B 1700 BASIC includes the capabilities of the original Dartmouth College BASIC plus extensions provided by Burroughs.

BASIC is a problem-oriented language designed for a wide range of applications and may be easily applied to both business/commercial and 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 1800/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 a B 1800/B 1700 computer system using the BASIC interpreter. The interpreter causes the system hardware to perform the operations specified by the S-code and thus the source program.

The B 1800/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.

SECTION 1
COMPONENTS OF BASIC

GENERAL

BASIC statements are composed of certain key words that are used with the fundamental elements of the language. The fundamental elements of B 1800/B 1700 BASIC are discussed in this section.

CONSTANTS

A constant in the BASIC language represents a value that cannot change during program execution. Constants can be either numeric or string and are assigned by the programmer.

Numeric Constants

A numeric constant consists of a series of decimal digits which may be preceded by a sign (+ or –) and may optionally contain a decimal point. Unlike some programming languages, BASIC does not distinguish between numbers containing a decimal point (real numbers) and those written without a decimal point (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 by which the number preceding the letter E is to be raised. For example, 2.145E–4 represents the value .0002145. The decimal point in a number expressed in E notation may follow or precede any digit if the correct power of 10 is expressed.

In B 1800/B 1700 BASIC, the range for numeric values is 4.31808E–78 through 5.7896E+76. Zero is also allowed. The precision for numeric values is nine significant digits. The maximum integer value which can be accurately represented is 2**30–1 (1073741823).

String Constants

A string constant is enclosed in quotation marks and consists of any sequence of EBCDIC characters valid to the B 1800/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, $*1-%#

VARIABLES

Unlike a constant, whose value remains fixed, a variable name represents a data item whose value may be changed during program execution. Variables are associated with either numeric or string values, and may be either simple variables or array elements. Explicit declarations of variable types are not required; a currency symbol ($) serves to distinguish string from numeric variables, and the presence of a subscript distinguishes an array element from a simple variable.

Numeric Variables

A numeric variable name is a symbolic name used to represent a numeric value. 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 1800/B 1700 BASIC, each numeric variable used in a program is initialized to 0 during compilation.

A string variable is a symbolic name used to represent an EBCDIC character string. A string variable name is a letter, optionally followed by a digit, followed by a currency symbol ($). Valid string variable names are:

A$ B$ C$ D$.....X$ Y$ A1$ B6$ C9$

In B 1800/B 1700 BASIC, each string variable used in a program is initialized to null during compilation. The length of the character string associated with a string variable can vary during execution of a program from a length of 0 characters (signifying the null or empty string) to 4095 characters.

Subscripted 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 1- or 2-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 in the array.

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 enclosed in parentheses which indicate the position of the element within the array. 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 1-dimensional and 2-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 converted to an integer by using the formula INT(X + 0.5) before being used as a subscript.

In B 1800/B 1700 BASIC, each array used in a program is initialized to 0 (or to null if the array is a string array), during compilation.

EXPRESSIONS

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

Arithmetic
Relational
String

Arithmetic Expressions

An arithmetic expression is a means 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
**	Exponentiation
↑	Exponentiation
! (exclamation point)	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, two constants, or a combination of variables and constants. No arithmetic operation may be assumed to be present.

Examples of valid and invalid arithmetic expressions follow:

```
Valid                Invalid
-----                -------
-A+B                 A//B
(A*B)-(C**5)         (A+B) (C*D)
X+4*5
```

Parentheses may be used in an arithmetic expression to denote the order in which operations are to be performed. Parentheses have highest 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
	Unary minus
	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. For example, 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 as follows:

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

Chains of relations are not permitted. For example, A<B<C is not a valid relational expression.

String Expressions

A string expression represents a sequence of string operations to be performed over the given set of string constants, string variables (either simple or subscripted), and string function references, or combination of these, resulting in a string value. Concatenation is the only operation permitted in a string expression. The concatenation operator is either ampersand (&) or plus (+).

Example:

```
10 A$ = "FIRST ---"
20 B$ = "SECOND ---"
30 C$ = A$ + B$ & "THIRD"
40 PRINT C$
50 END
```

Result:

```
FIRST ---SECOND ---THIRD
```

SYNTAX RULES

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

1. Each source statement must be on a separate card image.
2. A statement may not be continued onto a second card image.
3. Each statement must begin with a unique line number. The line number must start 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. A given BASIC line of code may contain a maximum of 243 characters per line.
4. Blanks are ignored except in strings, image statements, and within or preceding line numbers at the beginning of lines. Blanks, with these exceptions, may occur anywhere in a BASIC statement and may be used as follows to improve the appearance and readability of a program.

```
15  GO TO 245
15 GO TO 245
15GOTO245
15 G O  TO  2 4 5
```

SECTION 2
ASSIGNMENT STATEMENTS

GENERAL

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

Arithmetic assignment statement.
Multiple arithmetic assignment statement.
String assignment statement.
Multiple string 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:

```
10 LET A = 4
15 LET P(2,3) =  SQR(A**2 + B**2)
20 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.

The multiple arithmetic assignment 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
50 LET V(1) =W=X=Y=X= 2*A+B
```

STRING ASSIGNMENT STATEMENT

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

<n> LET <variable> = <string expression>

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

The word LET is optional.

Examples:

```
010 Z1$ = "DEF"
015 LET P$(2,3) = "ABC" + Z1$
100 LET A$ = P$(2,3)
```

MULTIPLE STRING ASSIGNMENT STATEMENT

The multiple string assignment statement is used to assign a value to more than one string variable, and has the following format:

<n> LET <variable1> = <variable2> = ... <string. expression >

where <n> is the line number of the multiple string assignment statement, <variable1> = <variable2>... represents a sequence of simple or subscripted string variable names separated by the equal sign, and <string

expression> is a string constant, string variable (either simple or subscripted), or a string expression. This statement assigns the value of the <string expression> to each variable on a left-to-right basis. For example, the string assignment statement:

```
10 LET I = 3
15 LET X$ = Y$ = A$(I) = "FOUR"
```

assigns the string "FOUR" to X$, Y$ and A$(3) as though the following statements had been used:

```
10 LET X$ = "FOUR"
20 LET Y$ = "FOUR"
30 LET A$(3) = "FOUR"
```

The word LET is optional.

Examples:

```
10 LET A$ = B$ = C$ = "ZERO"
15 D$ = E$ = F1$ = 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 the control to another part of the program, terminate program execution, or control iterative processes. The following control statements are discussed in this section:

GO TO statement.
IF statement.
ON statement.
FOR statement.
NEXT statement.
END statement.
STOP statement.
CHAIN 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 is 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> IF <relational expression> [THEN] <line number>
[GO TO]

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

Example:

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

specifies that program control is to be transferred from line number 30 to line number 70 if the value of A is less than 100 when this line is executed. If A is greater than or equal to 100, control is passed to the line 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> ON <arithmetic expression> [THEN] <line1>,<line2>,...
                               [GO TO]
```

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 <line1>, <line2>, <line3>, ... represent the line numbers to which program control is transferred, depending upon the value of the <arithmetic expression>.

If the value of the <arithmetic expression> is 1, program control transfers to the statement whose line number is <line1>. A value of 2 transfers control to the statement whose line number is <line2> and so on. If INT(<arithmetic expression>) is less than 1 or greater than the number of elements in the list, a run-time error message is printed and the program halts.

Example:

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

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

PROGRAM LOOPS

The BASIC language provides the capability to execute a portion of code repeatedly. This function is referred to as a program loop. Certain parameters are supplied to the loop to specify the number of times the portion of code is to be executed.

Example:

```
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 12,5,15,14,1,18,4,15
```

In this example, line numbers 10 through 40 are executed in the normal manner. When line number 50 is executed, program control is unconditionally transferred back to line number 10. Line numbers 10 through 40

are repeated and when line number 50 is executed, control is once more returned to line number 10. This repetition continues until all of the data items in line number 60 are exhausted. At that time, program execution is terminated. Since the loop is one of the most useful tools in programming, the FOR and NEXT statements are provided in BASIC for controlling program loops.

FOR AND NEXT STATEMENTS

The FOR and NEXT statements are used together 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:

<n> FOR <variable>=<expression1> TO <expression2>

<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>, known as the index of the loop, must be an unsubscripted numeric variable. 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. 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 range of the loop is delimited by the NEXT statement, whose format is as follows:

<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:

1. 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.
2. The value of <expression1> is assigned to a temporary simple variable supplied by the compiler.
3. 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 0, step 4 is performed; otherwise, step e is performed.
4. 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 exited and program control is transferred to the statement following the NEXT statement; otherwise, step 6 is performed.
5. 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 6 is performed.
6. The value of the temporary variable is assigned to the index <variable>.
7. All executable statements within the range of the FOR... NEXT loop are executed.
8. The value of <expression3> is added to the temporary variable, and steps 3 through 7 are repeated until the loop is satisfied.

When it is desired to increment the index <variable> by something other than 1, the STEP clause (expression) is used in the FOR statement. The step size need not be an integer.

Example:

```
10 FOR I= 1 TO 10 STEP .1
```

The above example 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 is executed indefinitely.

The values 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 is not executed and program control is 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 overlap 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=2 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, and FOR...NEXT loops may not be nested to more than 20 levels.

Example:

```
05 FOR I=1 TO 9 STEP 4
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 12,5,15,14,1,18,4,15
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 1800/B 1700 BASIC, the END statement is optional in a program. The END statement, if used, must be the last statement in a program.

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.

CHAIN STATEMENT

The CHAIN statement is used to begin compilation and execution of the designated program and terminate the current program without operator intervention. The CHAIN statement has the following format:

<n> CHAIN <file name>

where <n> is the line number of the CHAIN statement. The <file name> must conform to the standard rules for disk names and is the name of a BASIC source program residing on disk.

All files are closed before the new program is initiated. The new object file is not saved.

Examples:

```
CHAIN "USER/PROGRAM/2"
CHAIN "USER/PROGRAM3/"
CHAIN "PROGRAM/4"
CHAIN "PROGRAM5"
```

Note

The first two examples require source files to be on a user pack labeled USER. The last two examples require source files to be on the system disk.

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 separated by commas. An array declaration consists of an array name followed by a left parenthesis, followed by either a single positive integer indicating a 1-dimensional array or two positive integers separated by a comma indicating a 2-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 1800/B 1700 BASIC may be any single alphabetic letter. Examples of valid array names are:

A, B, C, D$, E$

Numeric array names are a single letter. String array names are a single letter followed by a $.

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 1-dimensional arrays (11 spaces total), and elements in both rows and columns 0 through 10 for 2-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 1-dimensional and 2-dimensional array, respectively.

The maximum number of elements permitted in an array is 262143 (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, or ON statement. This enables control to be transferred to a description of the routine which follows, thereby 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 ('). Anything in the columns between the apostrophe and the end of that line is ignored by the compiler and treated as a comment.

NULL STATEMENT

NULL statements are used to enhance the readability of the BASIC program. The NULL statement can be used anywhere in the BASIC program, and consists of a line number entry only. The following example contains NULL statements at line numbers 055 and 185.

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
055                                 'THIS IS A NULL STATEMENT
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
185                                 'THIS IS A NULL STATEMENT
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 contain data entered into the program by means of a DATA statement, or a remote terminal. The following input/output statements pertaining to internal files are discussed in this section:

1. READ and DATA statement.
2. RESTORE statement.
3. INPUT statement.
4. PRINT statement.
5. PRINT USING statement.
6. WRITE USING statement.

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 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 or string constants separated by commas.

Prior to the execution of a BASIC program, all DATA statements contained in a program are combined into one 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), they are ignored.

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 data block to the beginning of the 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 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 or remote terminal 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 or string variables (either simple or subscripted) separated by commas.

The INPUT statement allows data values to be assigned to variable names during execution of the program. 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 the INPUT file, individual data values

are punched free-field, and delimited by commas. Individual punch cards in the INPUT card file must not contain line numbers, because any line numbers in the file are treated as data elements. The INPUT statement obtains data from a remote terminal if the program was compiled and executed under control of CANDE.

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. The following is a list of possible error conditions that can occur during input:

Message	Condition
INVALID DATA ITEM	Illegal data type or wrong data type.
NOT ENOUGH INPUT	Insufficient data.
EXCESS INPUT	Excess input.
NUMBER OUT OF BOUNDS	Conversion of numeric data causes overflow or underflow.

PRINT STATEMENT

The PRINT statement is used to print output on the line printer or remote terminal during program execution. If the program was compiled from cards the output goes to the line printer. If the program was compiled under control of CANDE, the output goes to a remote terminal. 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), numeric or string expressions, 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

Zoned format provides the ability to evenly space data items across a print file record. There are five print zones, each 15 characters in length, in a print file record. 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, and so on. If more than five items are to be printed in the zoned format, output is continued in 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 a PRINT statement list that does not end with a control character (comma or semicolon), the next PRINT statement begins output on the succeeding line.

A comma encountered in a PRINT statement list causes the next data item to be printed either:

1. In the first position if the next print zone within the current output line, or
2. At the beginning of the first print zone on the next output line if the last zone of the current line has been used.

Skipping of print zones in the output line may be accomplished by using successive commas in the PRINT statement list. If the PRINT statement list ends with a control character (comma or semicolon), the succeeding PRINT statement merely begins output in the first position of the next print zone.

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 to be printed:

```
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 to be printed:

```
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 is made in the output line other than those spaces automatically inserted 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, output is continued on 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 is printed in packed format, it is necessary to understand how string constants and numeric values are printed by the PRINT statement.

String expressions 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:

1. Integer form. A number containing one to ten decimal digits printed without a decimal point.
2. Fractional form. A number containing one to six 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.
3. Scientific form. A number expressed in fractional form greater than one and less than ten followed by a space, followed by the letter E, followed by a signed 2-digit integer which is the power of 10 to which 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 (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 to be printed:

```
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 to be printed:

```
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 with no list following 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 to be printed:

```
SALES FOR WEEK ENDING MARCH 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, 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 to be printed:

```
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 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 are 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, it is converted to an integer. If the integer is larger than the current margin setting (75 in the absence of a MARGIN statement) it is divided by the current margin setting and the remainder is used. This final number is used 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 are 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.

IMAGE STATEMENT

The purpose of the IMAGE statement is to control the format of the program's output. The IMAGE statement has the following format:

<n> :<picture string>

where <n> is the line number of the statement which contains the picture string. The <picture string> is one or more of the format characters or images described in the following paragraphs.

<n> LET <string variable> = <picture string>

where <n> is the line number of the LET statement which contains the image elements, the <string variable> is any string variable name and <picture string> is a string expression.

Example:

```
10 :### ## ##
20 PRINT USING 10, 1, 1+1, 1-2
30 LET A$ = "## ####"
40 PRINT USING A$, 7-11, 100 *3.147
50 END
```

Execution of the above example causes the following to be printed (b indicates the insertion of a blank character):

```
bb1bb2b-1
-4bb315
```

Picture String Patterns

The picture string associated with an IMAGE statement is composed of numeric, string, or literal patterns or a combination of these patterns.

Numeric Patterns

Numeric patterns are a series of number signs (#) which may contain a decimal point, and which may be preceded or followed by certain special characters.

(Integer Pattern)

In an integer pattern, each number sign reserves a place in the field. The corresponding numeric expression is evaluated and converted to an integer. This value is used to store digits right-justified into the places reserved. If the integral value is negative, the first place to the left of the digits is filled with a minus sign. Unused places remaining at the left are filled with spaces. If insufficient places were reserved by the pattern for this process, an asterisk is printed in the leftmost field position and the field is widened to the right. Each set of multiple spaces in a literal pattern (leading, embedded, or trailing) on the same line and anywhere to the right of the inadequate field is reduced to a single space in an attempt to maintain column alignment.

Example:

```
10 :###        ###        ###        ###
12 :###AB###  A   B    ###
15 :###   ###   ###
20 PRINT USING 10,1,2,3,4
30 PRINT USING 10,1,1,999999,12345.3,17
40 PRINT USING 10,1,2,3,4
50 PRINT USING 12,-1234,123,123
60 PRINT USING 15,-1234,123,123
70 END
```

Execution of the above example causes the following to be printed:

```
bb1bbbbbbbb2bbbbbbbb3bbbbbbbb4
bb1bbbbbbbb1bbbbbb*999999bb*12345
b17
bb1bbbbbbbb2bbbbbbbb3bbbbbbbb4
*-1234AB123bAbBb123
*-1234b123b123
```

(Decimal Pattern)

In a decimal pattern, each number sign (and the decimal point) reserves a place in the field. The corresponding numeric expression is evaluated and rounded so that its fractional part contains as many digits as there are places reserved to the right of the decimal point in the field. This rounded value is used to store digits into the places reserved. If the value is negative, the first place to the left of the digits is filled with a minus sign. Unused places to the left of the decimal point are filled with spaces; unused places to the right of the decimal point are filled with zeros. Overflow of the field is treated in the same manner as specified for integer patterns.

Example:

```
10 :###.###      ###.###     ###.###
20 PRINT USING 10, 1.234567,1234.567,12,34567
30 END
```

Execution of the above example causes the following to be printed:

```
bb1.235bbbbb*1234.567bb12.000
*34567.000
```

|||| (Exponential Pattern)

Exponential patterns are used as follows. Each number sign, the decimal point, and the four vertical bars (the exclamation mark on the keyboard is printed as a vertical bar on the line printer) reserve places in the field. The corresponding numeric expression is evaluated, and scaled and rounded so that it has one digit fewer than the number of number signs in the field. The first number sign is filled with a blank character if the value is positive, or with a minus sign if the value is negative. The scaled, rounded value is then stored in the remaining number sign positions. Then the first vertical bar is replaced by E, the second by a plus or minus, and the third and fourth by the appropriate exponent as adjusted during the scaling. Unused places to the right of the decimal point are filled with zeros. No widening can occur. The circumflex (@), and the right bracket (]) may be substituted for the vertical bar (|).

Example:

```
10 :##.####!!!!
20 FOR I=0 TO 5
30 PRINT USING 10, 1.234567*10**I
40 NEXT I
50 END
```

Execution of the above example causes the following to be printed:

```
b1.235E+00
b1.235E+01
b1.235E+02
b1.235E+03
b1.235E+04
b1.235E+05
```

– (Minus Pattern)

The minus pattern is an extension of the integer pattern, decimal pattern, or exponential pattern which permits the minus sign associated with a negative value to float. Minus signs may appear only at the left of the specified pattern and are equivalent to number signs except that a minus sign is printed to the left of a negative number in that field. No sign is printed for a positive number. Only one such sign is printed. It is floated to the right until it is adjacent to the numeric output, or until the next position in the pattern is a number sign or the decimal point.

Example:

```
10  :######  ####.##  ####.####!!!!
20  :--###  --##.##  --##.####!!!!
30 PRINT USING 10, 1,2,3
40 PRINT USING 20, 1,2,3
50 PRINT USING 10, -1,-2,7,-3.591
60 PRINT USING 20, -1,-2.7,-3.591
99 END
```

Execution of the above example causes the following to be printed:

```
bbbb1bbbbb2.00bbb300.00E-02
bbbb1bbbbb2.00bbb300.00E-02
bbb-1bbbb-2.70bb-359.10E-02
b-  1bbb- 2.70bb-359.10E-02
```

+ (Plus Pattern)

In general, the plus pattern is just like the minus pattern, but with plus signs also visible when appropriate. It is an extension of the integer pattern, decimal pattern, or exponential pattern and permits either algebraic sign associated with a value to be displayed and to float. Plus signs may appear only at the left of the specified pattern and are equivalent to number signs, except that they cause a plus or minus sign to be printed to the left of a value. Only one plus or minus sign is printed and is floated to the right until it is adjacent to the numeric output or until the next position in the pattern is a number sign or the decimal point.

Example:

```
10 :#####   ####.##   #####.##!!!!
20 :++###   ++##.##   ++###.##!!!!
30 PRINT USING 10, 1,2,3
40 PRINT USING 20, 1,2,3
50 PRINT USING 10, -1,-2.7,-3.591
60 PRINT USING 20, -1,-2.7,-3.591
99 END
```

Execution of the above example causes the following to be printed:

```
bbb1bbbbb2.00bbb3000.00E-03
b+  1bbb+ 2.00bb+3000.00E-03
bbb-1bbbb-2.70bb-3591.00E-03
b-  1bbb- 2.70bb-3591.00E-03
```

$ (Currency Pattern)

The currency pattern is basically an integer pattern or decimal pattern but also makes use of a floating currency symbol. The currency symbol may appear only at the left of the integer pattern or decimal pattern and is equivalent in effect to the number sign except that in the output a currency symbol appears to the left of the (possibly signed) value. Only one such currency symbol is printed. The currency symbol is floated to the right until it is adjacent to the signed numeric output, or until the next position in the pattern is a number sign or the decimal point.

Example:

```
10 :$###.##   $$##.##   $$$#.##   $$$$.##
20 PRINT USING 10,1,1,1,1
30 PRINT USING 10, -1,-1,-1,-1
40 PRINT USING 10, .12,.12,.12,.12
50 PRINT USING 10, -.12,-.12,-.12,-.12
99 END
```

Execution of the above example causes the following to be printed:

```
$  1.00bbb$ 1.00bbbb$1.00bbbb$1.00
$ -1.00bbb$-1.00bbb$-1.00bbb$-1.00
$  0.12bbb$ 0.12bbbb$0.12bbbbb$.12
$ -0.12bbb$-0.12bbb$-0.12bbb$-0.12
```

* (Asterisk Pattern)

The asterisk pattern is essentially a check-protection variant of the currency pattern. It consists of a currency symbol, followed by a sequence of asterisks, followed by either an integer pattern or decimal pattern. The asterisks are treated as number signs appended to the left of the integer pattern or decimal pattern, except that when asterisks appear in the pattern, asterisks rather than spaces are used for left-side fill characters.

Example:

```
10   :$******.##
20 FOR I=-2 TO 5
30 PRINT USING 10,3.7*10**I
40 NEXT I
50 END
```

Execution of the above example causes the following to be printed:

```
$******.04
$******.37
$****3.70
$***37.00
$**370.00
$*3700.00
$37000.00
*$370000.00
```

String Patterns

A string pattern consists of an apostrophe followed by a justifier. A justifier is a series of zero or more L, R, E or C characters.

(String Justifier)

The apostrophe reserves a place for a character of the string, as do the E, L, C, or R characters which complete the field. The string expression is evaluated, and its characters are used to fill the reserved places.

L (String Justifier)

If the justifier is L, the string is to be left-justified within the field. Unused places at the right are filled with spaces; excess characters are truncated on the right.

E (String Justifier)

If the justifier is E, the string is to be left-justified within the field. Unused places at the right are filled with spaces; excess characters cause the field to be extended to the right. If the field is extended, each set of multiple spaces (leading, embedded, or trailing) in a literal pattern on the same line and anywhere to the right of the inadequate field is reduced to a single space in an attempt to maintain column alignment.

R (String Justifier)

If the justifier is R, the string is to be right-justified within the field. Unused places at the left are filled with spaces; excess characters are truncated on the right.

C (String Justifier)

If the justifier is C, the string is to be centered within the field. The algorithm for determining the leading spaces is:

(INT((places-in-field - characters-in-string)/2)

Unused places at both ends are filled with spaces; excess characters are truncated on the right.

Example:

```
10 :'LLLLL          'LLLLL          'LLLLL
20 :'EEEEE          'EEEEE          'EEEEE
30 :'RRRRR          'RRRRR          'RRRRR
40 :'CCCCC          'CCCCC          'CCCCC
50 LET A$="ABC"
60 LET B$="LMNOPQRS"
70 LET C$="XYZ"
80 PRINT USING 10, A$,B$,C$
90 PRINT USING 20, A$,B$,C$
91 PRINT USING 30, A$,B$,C$
92 PRINT USING 40, A$,B$,C$
99 END
```

Execution of the above example causes the following to be printed:

```
ABC             LMNOPQ          XYZ
ABC             LMNOPQRS        XYZ
   ABC          LMNOPQ             XYZ
 ABC            LMNOPQ           XYZ
```

Literal Patterns

A literal pattern is composed of characters or character strings which are neither string patterns nor numeric patterns. A literal pattern appears on the print line exactly as it appears in the image. The letters C, E, L and R may be used since they are only format control characters when they are preceded by an apostrophe.

Formatted Output Rules

The following are the rules governing output when using format specifiers.

1. The IMAGE statement can be used in conjunction with the following output statements:

   a. PRINT USING statement.
   b. WRITE USING statement.
   c. MAT PRINT USING statement.
   d. MAT WRITE USING statement.

2. The execution of a WRITE USING statement generates a line which contains a delimiter immediately following each non-literal field. The delimiters are extra characters not accounted for in the picture string.

   Example:

```
10 :###XXX'LLL
20 WRITE USING 10, 123, "ABC"
30 END
```

   Execution of the above example causes the following to be printed:

```
123,XXXABC ,
```

3. Each expression is evaluated in turn and its value used to complete the corresponding field as described below. Output proceeds until a non-literal pattern is encountered in the picture string and the expression list is exhausted.

   Example:

```
10 :##########/##/##/##/##/
20 PRINT USING 10, 1,2
30 END
```

   Execution of the above example causes the following to be printed:

```
bbb1///b2//
```

4. If all patterns in the string have been exhausted by the time an expression is evaluated, then a new line is begun and the first pattern in the picture string is used again. This process terminates when the end of the expression list is reached.

   Example:

```
10 :###;;###::
20 PRINT USING 10, 1,2,3,4,5
25 PRINT USING 10,6,7
30 END
```

   Execution of the above example causes the following to be printed:

```
bb1;;bb2::
bb3;;bb4::
bb5;;
bb6;;bb7::
```

5. Each formatted output statement starts at the beginning of its image. If the expression list ends in a comma or semicolon separator, the line is not transmitted until another output statement or the end of the program causes the line to be transmitted.

   Example:

```
10 :####A ###B ###C ###D ###E ###F ###G ###H
20 PRINT USING 10,7,8,9,10
30 PRINT USING 10, 11,12,13
40 PRINT USING 10, 14,15,16,
50 PRINT USING 10, 17,18,19,20
99 END
```

   Execution of the above example causes the following to be printed:

```
bb7Abbb8Bbbb9Cbb10D
b11Abb12Bbb13C
b14Abb15Bbb16Cbb17Abb18Bbb19Cbb20D
```

6. If the inclusion of any complete literal field would cause the length of the current line to exceed the margin, then as many characters of the literal are generated as are needed to bring the length of the current line up to the margin. A new line is begun, and the rest of the output is generated. If the inclusion of any complete non-literal field would cause the length of the current line to exceed the margin, then the item's value begins a new line, and is not split unless its length is greater than the margin size.

   Example:

```
10 MARGIN #0:13
20 :### #### ########
30 :'LLLLLLLLLLLL
40 :###XXXXXXXXXXXX###
50 :########.#######
60 :'LLLLLLLLXX'LLL
70 PRINT 1;2;3;4;5;6;7;8
80 PRINT USING 20, 12,345
81 PRINT USING 20, 98,765,4321012
82 PRINT USING 20, 20,71,285
83 PRINT USING 30, "ABCDEFGHIJKLMN"
84 PRINT USING 40, 123,456
85 PRINT USING 50, 1.2345*10**(-3)
86 PRINT USING 60, "ABC","DEF"
90 END
```

   Execution of the above example causes the following to be printed:

```
b1bb2bb3bb4
b5bb6bb7bb8
b12bb345
b98bb765
b4321012
b20bbb71
bbbbb285
ABCDEFGHIJKLM
N
123XXXXXXXXX
XXX456
bbbbbbb.00123
45
ABCbbbbbbXX
DEF
```

PRINT USING STATEMENT

The PRINT USING statement is used to provide formatted output. A given line of output may contain up to 174 characters of data. The PRINT USING statement has the following format:

<n> PRINT USING <image> , <expression list>

where <n> is the line number of the PRINT USING statement, <image> is either the line number of the image statement to be used for formatting the output or a string expression containing the format image to be used, and <expression list> is the set of elements to be placed in the output file. The <expression list> may contain any numeric or string expressions. It may not contain a TAB function. The elements in the list must be separated from one another by a comma or a semicolon.

Example:

```
10 PRINT USING 10,A$,B,C+7,25
```

Each PRINT USING statement, when executed, begins with the first element of the associated image, even though a previous execution of that statement may not have used all of the elements of the image. The PRINT USING statement begins a new line each time it is executed unless a previous PRINT statement ended with a comma or a semicolon. If there are more data elements in the list than there are image elements in the associated image, the excess data elements are printed on a new line and the image is used again beginning with the first image element. The items in the expression list are formatted in exactly the manner shown by the format elements in the IMAGE statement. Unlike the PRINT statement, the separators between the items in the expression list (semicolons or commas) do not control the spacing between the items on the output line. The spacing between items is identical to the spacing between elements in the IMAGE statement.

WRITE USING STATEMENT

The WRITE USING statement performs the same function as the PRINT USING statement; however, with the WRITE USING statement delimiters follow the data. The WRITE USING statement has the following format:

<n> WRITE USING <image>, <expression list>

where <n> is the line number of the WRITE USING statement, <image> is either the line number of the image statement to be used for formatting the output or a string expression containing the format image to be used, and <expression list> is the set of elements to be placed in the output file. The <expression list> may contain any numeric or string expressions. It may not contain a TAB function. The elements in the list must be separated from one another by a comma or a semicolon.

Example:

```
10 WRITE USING 20, A, B, C, D
```

SECTION 7
DISK FILES

GENERAL

One of the features of the BASIC language 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. The use of disk files frees the program from containing all of the data which is to be processed, thereby permitting the construction of larger programs. Disk files can be created either by a program or by a user at the terminal and saved for future use. The following paragraphs explain how files are made accessible to the BASIC program through file declaration statements.

In BASIC there are two major types of files. Data can either be stored as EBCDIC characters, or as memory-image "words".

FILE DECLARATIONS

The FILES and FILE statements are used in BASIC to make files available to the program, and to specify the type of file that is being accessed.

FILES Statement

The FILES statement is used to specify the external files which are available to the BASIC program. It is also used to specify temporary files, and to reserve space for files which the programmer does not want to explicitly name at the time of the declaration. A program may contain more than one FILES statement and a FILES statement may appear anywhere in a program. The first file name declared in the program is designated as file 1, the second as file 2, and so on, and the files are referenced by these unique numbers. If a second FILES statement occurs, and 4 files had been declared in the first FILE statement, then the first file in the second FILE declaration statement would be number 5, and so on. A maximum of 16 total file names may appear in FILES statements within a program. The FILES statement has the following format:

<n> FILES <file-name1>; <file-name2>; ...

where <n> is the line number of the BASIC statement, and <file-name> is the name of a file on disk, a file place holder, or temporary file identifier.

If the file name is to refer to an actual file on disk, the name of the file may not be a quoted string and follow standard Burroughs naming conventions. Refer to the B 1800/B 1700 Systems Software Operational Guide for further information on naming conventions. When the file name is a file identifier, the file is opened and placed in the read mode.

A file place holder is denoted by an asterisk (*) and reserves space for a disk file that can be specified later on in the program. The file place holder permits the programmer to reserve room for a file before the program logic has decided which file it needs to process. When it has been decided which file to use, an explicit declaration can be made in a FILE statement, which is described later. When the file name is a file place holder, a file number is reserved for a file; however, no file may be referenced by that number until a FILE statement references the file.

The temporary file identifier, denoted by two asterisks (**) creates a temporary work file for use by the BASIC program.

Example of the FILES statement:

```
10 FILES ALPHA;BETA;*;BETA/GAMMA;**;A/B/C
```

In the previous example ALPHA, and BETA/GAMMA are unique names of files on the system disk. A/B/C is the name of a file on a User disk named A. The * reserves a file number (3) for a disk file yet to be specified by the program. The ** makes file number 5 a temporary file, accessible to the BASIC program.

File Designator

A file designator is a symbol which refers to the type of file being accessed, whether character or memory-image, followed by a numeric constant or numeric expression. The value of the expression is made an integer to obtain the file number. A character-file designator is a number sign (#) followed by the file number (constant or expression). A memory-image-file designator is a colon (:) followed by the file number (constant or expression).

Examples:

```
10 SCRATCH #1
20 SCRATCH :2+K
```

Statement 10 in the previous example performs an operation on a character file, while statement 20 operates on a memory-image file.

Once a file is classified as a character or memory-image file, subsequent statements referring to it must be consistent with that classification until the file is reclassified. The distinction between character files and memory-image files is based solely on the nature of the program statements referring to the file. Reference to a memory-image file with a statement whose format is appropriate to a character file results in the error message FILE NOT CHARACTER, and execution of the program is terminated. The file remains intact, however. In order to reclassify a file, scratch the file with a SCRATCH statement of the form appropriate to the new classification of the file.

FILE Statement

The FILE statement is used to open or close designated files, and to declare the type of file being accessed, whether character or memory-image. It is also used to specify the name of a file for a file number which had previously been given a file place holder in a FILES statement. The FILE statement can optionally include a length attribute for memory-image files which are to be randomly accessed. The FILE statement has the following format:

<n> FILE <file-designator><file-number><p>[<file-name>][,<l>]

where <n> is the line number of the statement, <file-designator> is either a number sign (#) or a colon (:), and <file-number> is the relative number of the file established in the FILES statement. This number may not be 0 or greater than the number of files declared. The punctuation, <p>, after the <file-number> can be either a comma or a colon. The <file-name> can be the actual name of a file given in quotes, a file place holder (*), or a temporary file designator (**). The length, <l>, gives the number of "words" in a random

access file and specifies to the program that a file is random. Without this attribute a file is considered to be sequential.

Examples of the FILE statement:

```
100 FILE #1,"EPSILON"
200 FILE :2,"**",20
300 FILE :3,7
400 FILE #4:"*"
```

In the previous statements, line 100 associates file 1 with a disk file named EPSILON. If file 1 had previously referred to another disk file, that file is closed and EPSILON is opened in its place. Further references to file 1 become references to EPSILON. This allows the program to access more than 16 files as long as only 16 files are open at any given time. Line 100 also gives the method by which a program can give an actual file name for a file place holder described in the FILES statement.

Line 200 in the previous example makes file 2 a temporary file. Whatever file 2 had previously been is overridden. Line 200 and line 300 give lengths for the files to which they refer. This specifies that the files are to be random access and to have the given lengths.

Closing a disk file is demonstrated on line 400. The asterisk indicates that the file associated with number 4 is to be closed and made inaccessible.

FILE Modes

When a file, either memory-image or character, is being processed, it is in either the read mode, which means that only READ or INPUT statements may be executed with the file, or the write mode, which means that only WRITE or PRINT statements may be executed with the file. The FILE statement places a file in the read mode unless the file is a temporary file. Before WRITE statements or PRINT statements can be executed with a file which is in the read mode, it must be placed in the write mode by executing either a SCRATCH statement or an APPEND statement. Before READ statements or INPUT statements can be executed with a file which is in the write mode, it must be placed in the read mode by executing a RESTORE statement, BACKSPACE statement, or FILE statement. Memory-image files are not subject to these mode restrictions.

CHARACTER FILES

Character files refer to the standard EBCDIC representation of data, each character taking 8 bits to be represented. There are two divisions of character files, sequenced and unsequenced. When a file is saved, the sequence information is saved with it as an attribute. Sequenced files differ from unsequenced files in that they are numbered. Sequenced files may not be accessed using the INPUT or PRINT statements. Unsequenced files may not be accessed using the READ or WRITE statements. A record in both types of character files may only be read or written by accessing each record before it in order, with either a READ or a WRITE statement. The following statements regard the handling of character files.

INPUT Statement

The INPUT statement for character files reads items from the file specified by the character-file designator into the variables in the variable list. When the value of the character-file designator is 0, data items are read from the terminal on a time sharing system or from a card file on a batch system. The INPUT statement as-

sumes that each record in the file contains only data and that it does not contain a line number. The INPUT statement has the following format:

<n> INPUT #<numeric-expression><p><list-of-variable-names>

where <n> is the line number of the statement, and <numeric-expression> is any arithmetic expression giving the file number. The punctuation, <p>, can be either a comma or a colon. The <list-of-variable-names> contains a list of one or more variables (either simple or subscripted) separated by commas.

Example:

```
10 INPUT #1:L$,M$,X(2)
```

The INPUT statement is record oriented and ignores any margin restriction that may have been placed on the file. When there are more items in a record than variables in the list, the next INPUT statement reads the first item from the next record. When there are fewer items in a record than variables in the list, the INPUT statement reads all the items from that record and then continues reading from the next record. When the variable-list is empty, the next record is skipped.

The absence of data in an input item is to be interpreted as either 0 or the null string, depending on the type of the corresponding variable in the variable list. The absence of data in the last input item is to be interpreted as no data (that is, a trailing delimiter in an input record is ignored). On input from an external file, if an ampersand appears as the last data item, it is treated as valid data as opposed to a line continuation character.

When the INPUT statement reads beyond the end of file, 0 is assigned to all remaining numeric variables and the null string is assigned to all remaining string variables.

PRINT Statement

The PRINT statement for character files writes the items in the print statement list onto the file specified by the character-file designator. The PRINT statement generates a file with no line numbers and no delimiters. The format conventions of the PRINT statement for the terminal apply to the PRINT statement for character files. A comma following an item in the list spaces the file to the beginning of the next 15-character zone; a semicolon suppresses spacing. The PRINT statement has the following format:

<n> PRINT #<numeric-expression><p><output list>

where <n> is the line number, <numeric-expression> is an arithmetic expression giving the file number, and <p> is a comma or a colon. The <output list> is a list of numbers, characters, expressions, or variables to be output.

Example:

```
10 PRINT #1,A,B,C
```

Each PRINT statement writes one line of output unless the margin limit is exceeded or unless a comma or a semicolon terminates the list. When the margin limit is exceeded, the current line is written and the output continues on the next line. When a comma or a semicolon terminates the list, the next PRINT statement continues on the current line. When the PRINT statement list is empty and the current output line is not empty, that line is written. When both the PRINT statement list and the line are empty, a blank line is written.

Before a PRINT statement can be executed with an external file which is in the read mode, the file must be placed in the write mode using either a SCRATCH statement or an APPEND statement.

When the value of the character-file designator is 0, the output is written on the terminal. No SCRATCH statement is required. File 0 is the only file which can be accessed with a PRINT statement after it has been accessed with a WRITE statement.

Character READ Statement

The READ statement for character files reads items from the file specified by the character-file designator into the variables in the variable list. It assumes each record in the file begins with a line number, which is skipped by the READ statement. The READ statement ignores the margin size. It has the following format:

<n> READ #<numeric-expression><p><list-of-variables>

where <n> is the line number of the statement, <numeric-expression> is an arithmetic expression giving the file number, and <p> is a comma or a colon. The <list-of-variables> is a list of one or more simple or subscripted variables.

Example:

```
10 READ #2,X,Y,Z
```

The READ statement is item oriented. When there are more items in a record than variables in the list, the next READ statement reads the next item from the same record. When there are fewer items in a record than variables in the list, the READ statement reads all the items from that record and then, after first skipping the line number, continues reading from the next record. If the variable-list is empty, the next data item is skipped.

The absence of data in an input item is to be interpreted as either 0 or the null string, depending on the type of the corresponding variable in the variable list. The absence of data in the last input item is to be interpreted as no datum (a trailing delimiter in an input response is ignored).

When the READ statement reads beyond the end of file, 0 is assigned to all remaining numeric variables in the list and the null string is assigned to all remaining string variables.

When the value of the character-file designator is 0, data items are read from the block of numeric and string data created by the data statements.

Character WRITE Statement

The WRITE statement for character files writes the items in the PRINT statement list onto the file specified by the character-file designator. The WRITE statement generates a line-numbered file. The line numbers start with 100 and increase by increments of 10. A delimiter is written immediately following each item written onto the file. Unless otherwise specified by a DELIMIT statement, the delimiter is a comma. The WRITE statement has the following format:

<n> WRITE #<numeric-expression><p><output-list>

where <n> is the line number of the statement, <numeric-expression> is any arithmetic expression giving the file number, and <p> is a comma or a colon. The <output-list> contains one or more numbers, characters in quotes, expressions, or variables to be output to the file.

Example:

```
10 WRITE #N,X;TAB(15);SIN(X)
```

Each WRITE statement writes one output record unless the margin limit is exceeded or unless a comma or a semicolon terminates the list. If appending the output item and its subsequent delimiter to the current record causes the margin limit to be exceeded, the current record is written, a new line number is generated, and the output continues. When a comma or a semicolon terminates the list, the next WRITE statement continues with the current record. When the output list is empty and the current output record is not empty, that record is written. When both the output list and the record are empty, a record containing only a new line number is written.

The format conventions of the PRINT statement apply when writing a file. A comma following an item in the list spaces the file to the beginning of the next 15-character zone; a semicolon suppresses spacing. The first character position into which an item can be written is considered to be a character position 0.

Before a WRITE statement can be executed with an external file which is in the read mode, the file must be placed in the write mode using either a SCRATCH statement or an APPEND statement.

When the value of the character-file designator is 0, the output is written on the terminal. No SCRATCH statement is required and no line numbers are supplied. Delimiters are written. File 0 is the only file which can be accessed with a WRITE statement after it has been accessed with a PRINT statement.

FILE WRITE USING Statement

The FILE WRITE USING statement is used to format records written to an external file. The FILE WRITE USING statement has the following format:

<n> WRITE #<numeric-expression>, USING <image><expression-list>

where <n> is the line number of the FILE WRITE USING statement, <numeric-expression> is any arithmetic expression giving the file number, <image> is as described under the IMAGE statement, and <expression-list> is as described under the PRINT USING statement.

FILE PRINT USING Statement

The FILE PRINT USING statement is used to format records written to an external file without any delimiters or line numbers. The FILE PRINT USING statement has the following format:

<n> PRINT #<numeric-expression>, USING <image>,<expression-list>

where <n> is the line number of the FILE PRINT USING statement, <numeric-expression> is any arithmetic expression giving the file number, <image> is as described under the IMAGE statement, and <expression-list> is as described under PRINT USING statement.

DELIMIT Statement

The DELIMIT statement specifies the delimiter to be used with the file specified by the character-file designator. The DELIMIT statement has the following form:

<n> DELIMIT #[<numeric-expression><p>] (<delimiter>)

where <n> is the line number of the statement, <numeric-expression> is any arithmetic expression giving the file number, and <p> is a comma or a colon. The <delimiter> is any EBCDIC character.

Example:

```
10. DELIMIT #1, (;)
20 DELIMIT (;)
```

A comma is the default delimiter. The DELIMIT statement is the only way to specify a delimiter other than a comma. If the delimiter is specified to be the space character, multiple spaces are considered to be a single delimiter. If the file-designator is omitted the default is FILE 0.

MARGIN Statement

The MARGIN statement specifies the number of character positions in a line of the file specified by the character-file designator. If the file designator is omitted the default is FILE 0. In the absence of a MARGIN state-

ment, the margin size is set to 75. The INPUT statement and READ statements ignore the margin setting. The MARGIN statement has the following format:

<n> MARGIN [#<numeric-expression><p>]<margin-limit>

where <n> is the line number of the statement, <numeric-expression> is any arithmetic expression giving the file number, and <p> is a comma or a colon. The <margin-limit> is the setting for the margin.

Example:

```
10 MARGIN #N,60
20 MARGIN 60
```

The value of the expression is rounded up to an integer. When the integer is less than 1 or greater than the maximum margin size, the margin is set to the maximum margin size. The maximum margin size for any file other than the terminal is determined at the time of the first WRITE statement or PRINT statement to that file.

Before the first WRITE or PRINT operation to a scratched or a temporary file, the maximum margin size is 174. At the time of that first WRITE or PRINT operation, the maximum margin size of that file is set to the current margin setting.

Upon execution of an APPEND statement which references a saved character file, the maximum margin size is set to the maximum margin size saved with that file.

BACKSPACE Statement

The BACKSPACE statement moves the record pointer backward and sets the mode for the file specified by the character-file designator. If the pointer is moved to the previous record, the value of the VPS file function (described under the heading File Functions) is decremented by 1. The BACKSPACE statement has the following format:

<n> BACKSPACE #<numeric-expression>

where <n> is the line number of the statement and <numeric-expression> is any arithmetic expression giving the file number.

Example:

```
10 BACKSPACE #10
```

When the file is unsequenced, the BACKSPACE statement moves the pointer back one record. When the pointer is at the beginning of a record, the pointer is moved to the beginning of the previous record. When the pointer is in the middle of a record, it is moved to the beginning of that record.

When the file is sequenced, the BACKSPACE statement moves the pointer back one item.

When the BACKSPACE statement is executed with the pointer positioned at the beginning of the file, the pointer is not moved.

MEMORY-IMAGE FILES

Memory-image files refer to the 48-bit word representation of a data item, the form used when performing arithmetic operations in BASIC. This representation allows data to be accessed without conversion and lends itself to faster physical input/output. Memory-image files can be declared as sequential or random access type.

When a memory-image file is random access type (created by specifying a length in its FILE declaration), selective access to any specific data item may be accomplished directly, without reading through the file from the beginning to that data item. There is a pointer associated with each random file which points to the current word of the file. This pointer can be set using the SETW statement.

The following statements refer to the handling of memory-image files.

Memory-Image READ Statement

The READ statement for memory-image files reads items from the file specified by the memory-image-file designator into the variables in the variable list. There are no line numbers or delimiters in a memory-image file, just the memory image representations of the numbers and strings. A null entry is interpreted as 0 for a numeric variable and as the null string for a string variable. The memory-image READ statement has the following format:

<n> READ :<numeric-expression><p><list-of-variables>

where <n> is the line number of the statement, <numeric expression> is any arithmetic expression giving the file number, and <p> is a comma or a colon. The <list-of-variables> consists of one or more simple or subscripted numeric or string variables.

Example:

```
10 READ :12:S,D,L,X$(7)
```

When the variable in the list is numeric, the number is read from the position indicated by the pointer and the pointer is advanced one word.

When the variable in the list is of type string, the string control word is accessed from the position indicated by the pointer, the following string is read, and the pointer is advanced to the word following the string.

When the variable list is empty, the next data item is skipped. If the pointer is pointing at a string control word, the pointer is advanced to the word following the string; otherwise, the pointer is advanced one word.

For a sequential memory-image file, when the READ statement reads beyond the end of file, 0 is assigned to all remaining numeric variables in the list and the null string is assigned to all remaining string variables. For a random memory-image file, when the READ statement reads beyond the end of file, an error occurs.

Memory-Image WRITE Statement

The WRITE statement for memory-image files writes the items in the output list onto the file specified by the memory-image-file designator. The WRITE statement for memory-image files does not write line numbers or delimiters into the file, just the memory-image representations of the numbers and strings. The WRITE statement for memory-image files has the following format:

<n> WRITE :<numeric-expression><p><output-list>

where <n> is the line number of the statement, <numeric-expression> is any arithmetic expression giving the file number, and <p> is a comma or a colon. The <output-list> contains one or more numbers, constants, expressions (numeric or string), or variables.

Example:

```
10 WRITE:I+1,X,SIN(X)
```

The memory-image representation of any number occupies one word. The memory-image representation of a

string occupies one or more words. The first word contains a string control word which contains the length of the string. The following words contain the string, six characters per word. If the last word does not contain six characters, it is filled on the right with blanks. The string "ILLUSTRATE" is written on the disk as follows:

Word 1: The string control word
Word 2: ILLUST
Word 3: RATE

When the item in the list is a number, the number is written at the position indicated by the pointer and the pointer is advanced one word.

When the item in the list is a string, a string control word, followed by the string, is written starting at the position indicated by the pointer and the pointer is advanced to the word following the string.

When the output list is empty, a null entry is written at the position indicated by the pointer and the pointer is advanced one word. A sequential memory-image file which is in the read mode must be placed in the write mode using either a SCRATCH statement or an APPEND statement before a WRITE statement can be executed with it.

When a WRITE statement attempts to write beyond the end of a random memory-image file, an error occurs.

Memory-Image READ FORWARD Statement

The READ FORWARD statement for memory-image files selectively reads items from the file specified by the memory-image-file designator into the variables in the variable list. The memory-image READ FORWARD statement has the following format:

<n> READ FORWARD :<numeric-expression><p><variable-list>

where <n> is the line number of the statement, <numeric-expression> is any arithmetic expression giving the file number, and <p> is a comma or a colon. The <variable-list> contains one or more variables, either simple or subscripted.

Example:

```
10 READ FORWARD :3,A$,I,J,B$(X)
```

When the variable in the list is numeric, a forward search for a number is initiated at the position indicated by the pointer. When the pointer is not pointing at a string control word, the number is read and the pointer is advanced one word. If the pointer is pointing at a string control word, the pointer is advanced to the word following the string and the process is repeated. When the pointer is pointing at a word which contains string characters, that word is read as a number and the pointer is advanced one word.

When the variable in the list is of type string, a forward search for a string control word is initiated at the position indicated by the pointer. When the pointer is pointing at a string control word, the following string is read and the pointer is advanced to the word following the string. If the pointer is not pointing at a string control word, the pointer is advanced one word and the process is repeated.

When the variable list is empty, the next data item is skipped. If the pointer is pointing at a string control word, the pointer is advanced to the word following the string; otherwise, the pointer is advanced one word.

For a sequential memory-image file, when the READ FORWARD statement reads beyond the end of file, 0 is assigned to all remaining numeric variables in the list and the null string is assigned to all remaining string variables. For a random memory-image file, when the READ FORWARD statement reads beyond the end of file, an error occurs.

BACKSPACE$ Statement

The BACKSPACE$ statement moves the pointer backward to the last string control word in the file specified by the memory-image-file designator and, if the file is sequential, sets the read mode for that file. The BACKSPACE$ statement has the following format:

<n> BACKSPACE$:<numeric-expression>

where <n> is the line number of the statement and <numeric-expression> is any arithmetic expression giving the file number.

Example:

```
10 BACKSPACE$ :I+2
```

If no string control word is found, the pointer is moved to the beginning of the file. When the BACKSPACE$ statement is executed with the pointer positioned at the beginning of the file, the pointer is not moved.

SETW Statement

The SETW statement sets the pointer in the random memory-image file specified by the memory-image file number to the word specified by the word number. The SETW statement has the following format.

<n> SETW <numeric-expression> TO <numeric-expression>

where <n> is the line number of the statement and the first <numeric-expression> is the number of the file. The second <numeric-expression> is the word in the file to which the pointer is set (the first word is word 1).

Example:

```
10 SETW X TO V * C1-C2
```

Both the memory-image file number and the word number are converted to an integer before they are used. A word holds either a number, a string control word, or six string characters.

NON-TYPE DEPENDENT FILE STATEMENTS

This subsection gives the format and uses for other statements which manipulate disk files. These statements can be used with both character and memory-image files.

RESTORE Statement

The RESTORE statement positions the pointer to the beginning of the file and sets the mode for the file specified by the file designator. The RESTORE statement has the following format:

<n> RESTORE <file-designator>

where <n> is the line number of the statement and <file-designator> is any arithmetic expression preceded by a number sign or colon denoting either a character or memory-image file.

Example:

```
10 RESTORE #SGN(2)+2
```

SCRATCH Statement

The SCRATCH statement erases any existing data, positions the pointer to the beginning of the file, and sets the mode for the file specified by the file designator. The SCRATCH statement has the following format:

<n> SCRATCH <file-designator>

where <n> is the line number of the statement and <file-designator> is any numeric expression preceded by a number sign or colon designating the type of file.

Example:

```
10 SCRATCH :1
```

For a random memory-image file, all data items in the file are replaced by null entries so that, when the file is read, zeros or null strings are assigned to the variables in the read list.

APPEND Statement

The APPEND statement positions the pointer immediately following the last record, sets the mode for the file specified by the file designator, and sets the VPS file function (described under the heading FILE FUNCTIONS) to the number of records in the file. The APPEND statement may not reference a memory-image random file. The APPEND statement has the following format:

<n> APPEND <file-designator>

where <n> is the line number of the statement and <file-designator> is any numeric expression preceded by a number sign or colon.

Example:

```
10 APPEND #2
```

IF END Statement

The IF END statement tests for an end of file condition on the file specified by the file designator. The IF END statement has the following format:

<n> IF END <file-designator> <expression>

where <n> is the line number of the statement and <file-designator> is any numeric expression preceded by a number sign or colon. The <expression> is any BASIC expression except another IF statement, or a FOR statement.

Example:

```
10 IF END :3 X=X+1
```

When a sequential file is being read, an end-of-file condition exists after an attempt has been made to read beyond the end of the data. An end-of-file condition exists on a random memory-image file when the pointer is greater than the number of words in the file.

When a sequential file is being written, the file grows as items are added to it. It is impractical to test for an end-of-file condition when writing a sequential file.

IF MORE Statement

The IF MORE statement tests for more space available in the file specified by the file designator. The IF MORE statement has the following format:

<n> IF MORE <file-designator><expression>

where <n> is the line number of the statement and <file-designator> is any numeric expression preceded by a number sign or colon. The <expression> is any BASIC statement other than another IF statement, or a FOR statement.

Example:

```
10 IF MORE #0 GO TO 300
```

The above example tests the DATA line for unread data and branches if there is data that has not been read.

When a file is being read, the IF MORE statement tests for more data items remaining to be read in the file. When the file has been accessed with an INPUT statement, a test is made for more records remaining in the file; when the file has been accessed with a READ statement, a test is made for more data items remaining in the record or more records remaining in the file.

When a file is being written, the IF MORE statement tests for more space available to write items to the file. For a sequential file, there is almost always more space available since the space grows as the file is written. For a random file, there is more space available when the pointer is less than or equal to the number of words in the file.

FILE FUNCTIONS

The following file functions are built into BASIC and can be used in any expression. The argument of the function, x, is converted to an integer to be used as the file number and may be any numeric expression.

Function	Description
HPS(x)	Returns the character position in the current record of the character file. The first character is located at character position 0. The initial value of the HPS function is 0.
LIN(x)	Returns the line number of the current record of the character file. The initial value is 0.
VPS(x)	Returns the number of records which have been read from or written to the character file. The initial value is 0.
LCW(x)	Returns the current value of the word pointer for the random memory-image file. The first word is word number 1. The initial value is 1.
LFW(x)	Returns the size in words of the random memory-image file.

EXTERNAL FILE ERROR CONDITIONS

The following error conditions result in the associated error messages. Since recovery from these error conditions is not possible, they are considered fatal errors.

TOO MANY FILES
: More than 16 files declared in FILES statements.

DUPLICATE FILE NAMES
: File name appears more than once in FILES statements.

ILLEGAL FILE NAME
: Illegal file name.

INVALID FILE NUMBER

Value of file-designator less than 1 in some statements, less than 0 in others, greater than number of files declared, or specifies (in other than a FILE statement) a file which is closed.

INVALID FILE LENGTH

File length less than 1.

FILE NOT CHARACTER = <file name>

Tried to do character operation on memory-image file.

FILE NOT MEMORY-IMAGE = <file name>

Tried to do memory-image operation on character file

FILE NOT RANDOM

Tried to do random operation on sequential file.

FILE NOT SEQUENTIAL = <file name>

Tried to do sequential operation on random file.

FILE NOT SEQUENCED = <file name>

Tried to READ or WRITE an unsequenced file.

FILE NOT UNSEQUENCED = <file name>

Tried to INPUT or PRINT a sequenced file.

FILE NON-EXISTENT = <file name>

Tried to access a non-existent file.

ILLEGAL FILE COMMAND

Tried to INPUT or READ a file which was in the output mode or tried to PRINT or WRITE a file which was in the input mode (not referenced with scratch or append or last referenced with INPUT or READ).

DATA ITEM NOT NUMERIC

While reading a memory-image file, tried to read into a numeric variable when pointing at a string control word.

DATA ITEM NOT STRING

While reading a memory-image file, tried to read into a string variable when not pointing at a string control word.

INVALID WORD NUMBER

Word number less than 1 or greater than file length.

END OF FILE

Tried to read or write beyond end of file on a random memory-image file.

SECTION 8
INTRINSIC FUNCTIONS

GENERAL

Numeric and string 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.

The value of the standard functions, as well as the number and types of arguments required for each function, are described in this section. In all cases, X, Y, J, and K indicate numeric expressions, while A$, B$, and C$ indicate string expressions.

NUMERIC FUNCTIONS

The numeric intrinsic functions are:

Function	Meaning
ABS(X)	Absolute value
ATN(X)	Arctangent
BCL	Numeric time of day
COS(X)	Cosine
COT(X)	Cotangent
DET	Determinant
EBC(X)	EBCDIC
EXP(X)	Exponential
IDA	Numeric date
INT(X)	Integer
LEN(A$)	Length
LOG(X)	Logarithm
MOD(X,Y)	Modulo
NUM	Number of data items
RND	Random number generator
SCN(A$,B$,J,K)	Scan
SGN(X)	Sign
SIN(X)	Sine
SQR(X)	Square root
TAN(X)	Tangent
TIM	Elapsed time
VAL (A$)	Value

ABS(X) Function

The absolute value function, ABS(X), returns the absolute value of X.

Example:

```
10 LET X = 25.5
20 PRINT "ABSOLUTE VALUE OF A = ";ABS(X)
30 PRINT "ABSOLUTE VALUE OF -123.45 = ",ABS(-123.45)
40 END
```

Execution of the above example causes the following to be printed:

```
ABSOLUTE VALUE OF A = 25.5
ABSOLUTE VALUE OF -123.45 = 123.45
```

ATN(X) Function

The arctangent function, ATN(X), returns the arctangent of X in radians. The range of the function is:

$$-(PI/2) <= ATN(X) <= (PI/2)$$

where PI is the ratio of the circumference of a circle to its diameter.

BCL Function

The numeric time of day function, BCL, returns the time of day in hours and decimal fractions of hours based on a 24-hour clock. As an example, the following program sequence was executed at 11:02:

```
10 LET A = BCL
15 PRINT A
99 END
```

Execution of the above example causes the following to be printed:

```
11.0333
```

COS(X) Function

The cosine function, COS(X), returns the trigonometric cosine of X, where X is in radians.

COT(X) Function

The cotangent function, COT(X), returns the trigonometric cotangent of X, where X is in radians.

DET Function

The determinant function, DET, returns the determinant of the last matrix inverted, using the matrix function INV. The initial value is 0.

EBC(X) Function

The EBCDIC function, EBC(X), returns the EBCDIC code for an EBCDIC graphic character or EBCDIC mnemonic.

Examples:

```
10 EBC(")
20 EBC())
30 EBC(NAK)
```

EBCDIC Mnemonics

The following list of EBCDIC mnemonics can be used with either the EBC(X) function or to specify a delimiter:

ACK	BEL	BS	CAN	CR	DC1
DC2	DC3	DC4	DEL	DLE	EM
ENQ	EOT	ESC	ETB	ETX	FF
FS	GS	HT	LF	MZ	NAK
NL	NUL	PZ	RS	SI	SO
SOH	SP	STX	SUB	SYN	US
VT					

EXP(X) Function

The exponential function, EXP(X), returns the anti logarithm of X, which is the value of the base of the natural logarithm 2.71828..., raised to the power of X (e**X).

IDA Function

The numeric date function, IDA, returns the date in integer form, YYMMDD, where YY represents the last two digits of the year, MM represents two digits for the month, and DD represents two digits for the day. As an example, the following program sequence was executed on April 1, 1978.

```
10 LET A = IDA
15 PRINT A
99 END
```

Execution of the above example causes the following to be printed:

```
780401
```

INT(X) Function

The integer function, INT(X), returns the largest integer not greater than X.

Example:

```
10 A = 6.9
20 B = 6
30 C = -6.14
40 PRINT "GREATEST INTEGER NOT GREATER THAN"; A;" IS "; INT(A)
50 PRINT "GREATEST INTEGER NOT GREATER THAN"; B;" IS "; INT(B)
60 PRINT "GREATEST INTEGER NOT GREATER THAN";C;" IS "; INT(C)
70 PRINT "GREATEST INTEGER NOT GREATER THAN -2 IS "; INT(-2)
80 END
```

Execution of the above example causes the following to be printed:

```
GREATEST INTEGER NOT GREATER THAN 6.9 IS 6
GREATEST INTEGER NOT GREATER THAN 6 IS 6
GREATEST INTEGER NOT GREATER THAN -6.14 IS -7
GREATEST INTEGER NOT GREATER THAN -2 IS -2
```

LEN(A$) Function

The length function, LEN(A$), returns the number of characters in the specified string.

Example:

```
10 READ A$,B$,C$
20 LET A = LEN(A$)
30 LET B = LEN(B$)
40 LET C = LEN(C$)
50 PRINT "A$ = ";A$;" LENGTH OF A$ =";A
60 PRINT "B$ = ";B$;" LENGTH OF B$ =";B
70 PRINT "C$ = ";C$;" LENGTH OF C$ =";C
80 DATA ABC,DEFGH,IJKLMNOPQRST
99 END
```

Execution of the above example causes the following to be printed:

```
A$ = ABC LENGTH OF A$ = 3
B$ = DEFGH LENGTH OF B$ = 5
C$ = IJKLMNOPQRST LENGTH OF C$ = 12
```

LOG(X) Function

The logarithm function, LOG(X), returns the natural logarithm of X; X must be greater than 0.

MOD(X,Y) Function

The modulo function, MOD(X,Y), returns the value of X – (Y * INT (X/Y)); Y must not equal 0.

NUM Function

The number function, NUM, returns the number of data items input into the last array in a MAT INPUT statement.

RND Function

The RANDOM function, RND, returns the next number in a sequence of pseudo-random numbers between 0 and 1. The RND function does not require an argument. The same set of pseudo-random numbers is always used unless a RANDOMIZE statement has been executed.

RANDOMIZE Statement

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

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 is assigned to the array Z. If the randomize statement is removed from the program, the first 10 values from the standard set of random numbers are assigned to array Z each time the program is executed.

SCN(A$,B$,J,K) Function

The SCAN function, SCN(A$,B$,J,K), locates the specified occurrence of a segment of a string within a designated string and returns the number of the character position at which that occurrence was found.

The value returned is the character position in A$ of the first character of the Jth occurrence of B$, searching from character position K in A$. If a value of 0 is returned, there is no first Jth occurrence of B$ after the Kth character position in A$. The first character position is numbered 1.

Example:

```
10 A$ = "ABZABZZBA"
20 B$ = "BZ"
30 C$ = "ZZ"
40 D = SCN(A$,B$,1,1)
50 E = SCN(A$,B$,2,1)
60 F = SCN(A$,C$,1,1)
70 G = SCN(A$,C$,7,1)
80 PRINT D, E, F, G
99 END
```

Execution of the above example causes the following to be printed:

```
2             5             6             0
```

In the preceding example, the SCN function returned a value of 2 to the variable D. The function began its search at the first character position of A$ and searched for the first occurrence of the string BZ. As the first occurrence was found at the second character position of A$, the function returned a value of 2. A value of 6 was returned to the variable F for similar reasons. A value of 5 was returned to the variable E. The function began its search at the first character position of A$ and searched for the second occurrence of the string BZ. The second occurrence was found to begin at the fifth character position of A$. A value of 0 was returned to the variable G. The function again began its search at the first character position of A$ and searched for the seventh occurrence of the string ZZ. A seventh occurrence was not found.

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.

SIN(X) Function

The sine function, SIN(X), returns the trigonometric sine of X, where X is in radians.

SQR(X) Function

The square root function, SQR(X), returns a positive square root of X; X must be positive.

TAN(X) Function

The tangent function, TAN(X), returns the trigonometric tangent of X, where X is in radians.

TIM Function

The time function, TIM, returns the elapsed execution time in seconds.

Example:

```
10 FOR X = 1 TO 5E5
20 LET A = X
30 NEXT X
40 PRINT "ELAPSED TIME IS: ;TIM
99 END
```

Execution of the above example causes the following to be printed:

```
ELAPSED TIME IS: 14.95
```

VAL(A$) Function

The value function, VAL(A$), returns the numeric value of the EBCDIC string contained in the variable A$.

This function may be used to convert numbers which entered the program as strings to numeric values which may be used in arithmetic expressions. The numeric value may then be converted back to strings with the STR$(X) function.

Example:

```
10 LET A$ = "12"
20 LET B = VAL(A$)
30 LET Q =2*VAL(A$)
40 PRINT Q;
50 PRINT B;
99 END
```

Execution of the above example causes the following to be printed:

```
24  12
```

STRING FUNCTIONS

The string intrinsic functions are:

String Function	Meaning
CHR$(X)	Character
CLK$	Clock
DAT$	Date
EXT$(A$,J,K)	Extract
REP$(A$,B$,C$,J,K)	Replace
STR$(X)	Numeric to string conversion
UNO$	User number

CHR$(X) Function

The character function, CHR$(X), returns the 1-character string which corresponds to EBCDIC code X. Arguments greater than 255 are treated modulo 256.

CLK$ Function

The clock function, CLK$, returns an 8-character string that gives the time of day on a 24-hour clock in the form HH:MM:SS, where HH is hours, MM is minutes and SS is seconds.

Example:

```
10 PRINT CLK$
```

Execution of the above example causes the following to be printed:

```
23:14:55
```

DAT$ Function

The date function, DAT$, provides the calendar date as a string in the form MM/DD/YY.

Example:

```
10 LET A$ = DAT$
20 PRINT A$
99 END
```

Execution of the above example causes the following to be printed:

```
04/19/73
```

EXT$(A$,J,K) Function

The extract function, EXT$(A$,J,K), extracts the designated segment of the specified string for use in a string expression. The A$ is the string or string expression from which the extraction is to be made; J is the character

position in the string at which the extraction is to begin; and K is the character position in the string at which the extraction is to end. The first character position is numbered 1.

Example:

```
100 A$ = "BASIC"
110 B$ = EXT$(A$,2,4)
120 PRINT "THE STRING VARIABLE B$ = ";B$
999 END
```

Execution of the above example causes the following to be printed:

```
THE STRING VARIABLE B$ = ASI
```

In the preceding example, the string variable B$ was assigned the value of the segment of the specified string, A$, which began at the second character position and ended at the fourth character position.

REP$(A$,B$,C$,J,K) Function

The replace function, REP$(A$,B$,C$,J,K), replaces the specified occurrences of a segment of a string with the designated string. A$ is the string within which the segment is to be found, B$ is the segment which is to be replaced, and C$ is the string which is to replace B$. J specifies the number of occurrences of B$ to be replaced, and K is an integer which specifies the character position in A$ at which search and replacement begins.

If the parameter J is specified as less than 0, all occurrences of B$, including and following the Kth character in A$, are replaced with C$. If J is equal to 0, no replacements are made and the value returned by REP$ is equal to A$. If J is greater than 0, the number of occurrences of B$, specified by J are replaced by C$ beginning with the Kth character in A$. If the parameter J is specified as less than 0, the string to be replaced, B$, may not be null.

Example:

```
100 A$ = "A9999B333999B3339999A"
110 B$ = "B333"
120 C$ = "C666"
130 D$ = REP$(A$,B$,C$,1,6)
150 E$ = REP$(A$,B$,C$,2,6)
160 PRINT "D$ = ";D$
170 PRINT "E$ = ";E$
999 END
```

Execution of the above example causes the following to be printed:

```
D$ = A9999C666999B3339999A
E$ = A9999C666999C6669999A
```

STR$(X) Function

The numeric-to-string conversion function, STR$(X), returns the string value of X as shown in the following example. STR$(X) gives a value on conversion that is the same as if the value had been printed. Rounding may occur.

```
100 READ A
110 A$=STR$(A)
120 PRINT A; LEN (A$);A$
125 IF LEN(A$) = 10 GO TO 999
130 GO TO 100
140 DATA 1234.567, 1111.222,1234567890.
999 END
```

Execution of the above example causes the following to be printed:

```
1234.57   9   1234.57
1111.22   9   1111.22
1.23457E+09   13   1.23457E+09
```

NOTE

When a numerical expression is used, the LEN function counts a leading space (or a leading minus sign in the case of negative numbers), and a trailing blank, just as the PRINT statement would treat it.

UNO$ Function

The user number function, UNO$, returns the 10-character family name of the program.

Example:

```
100 LET A$ = UNO$
110 IF A$="B1700BASIC" THEN 150
120 PRINT "YOU ARE AN UNAUTHORIZED USER"
130 STOP
150 REM START OF PROGRAM
999 END
```

In the preceding example, if the file name returned by UNO$ is not "B1700BASIC", the current user is not allowed further access to the program.

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:

1. Function subprograms.
2. Subroutine subprograms.

FUNCTION SUBPROGRAMS

In addition to the standard functions provided by the BASIC compiler, the user may define functions within a 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> = <expression>
2. <n> DEF FN<letter> (<argument list>) = <expression>
3. <n> DEF FN<letter>$ = <expression>
4. <n> DEF FN<letter>$ (<argument list>) = <expression>

where <n> is the line number of the DEF statement; <letter> is any single alphabetic letter; <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. Options (3) and (4) are for string functions and have the same form as options (1) and (2) except for the $ character in the function name, and the fact that <expression> is a string expression.

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 3-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. A parameter that is passed to the function does not have its value changed even if the dummy variable associated with it is changed during the execution of the function.

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 therefore 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.

In a numeric function, the <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. In addition to the dummy variables, the <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)
40 DEF FNE$(W$) = W$ & W$
50 DEF FNF$(I) = "(" & STR$(I) & ")"
```

Multiple Statement Functions

Multiple statement functions are not limited to those functions which can be expressed in a single DEF statement as are single statement functions. A multiple statement function consists of a DEF statement followed by the statements which define the function and terminated by a FNEND statement. The DEF statement in the multiple statement function may have any of the following formats:

1. <n> DEF FN<letter> <local variable list>
2. <n> DEF FN<letter> (<argument list>) <local variable list>
3. <n> DEF FN<letter>$ <local variable list>
4. <n> DEF FN<letter>$ (<argument list>) <local variable list>

where <n> is the line number of the DEF statement; <letter> is any single alphabetic letter; <local variable list> contains a list of variables that are uniquely defined for that function; and in option 2, <argument list> is a list of one to seven unsubscripted dummy variable names separated by commas. String functions follow the same pattern as in single line functions.

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 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 is indicated by the FNEND statement.

Multiple statement functions may not be nested. In addition, neither transfer of program control from within a multiple statement function to some point in the program outside the function, nor the reverse, is 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, 0 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=I 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
999 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
```

SECTION 10
MATRIX OPERATIONS

GENERAL

As in many programming languages, matrices in BASIC can be manipulated by manipulating their elements. However, it is often more convenient to regard matrices as entities rather than as indexed collections of entities, and to manipulate the entire entity at one time. BASIC provides a number of standard operations to facilitate such manipulations.

All matrix statements (with the exception of data statements containing matrix data) are prefixed with the word MAT.

A matrix may be dimensioned in a DIM statement. In addition, a matrix may be redimensioned implicitly or explicitly when it appears in certain of the MAT statements. If a matrix is redimensioned in this manner, only the number of elements in a row and/or column may be changed. It is an error to attempt to redimension a 2-dimensional matrix as 1-dimensional or a 1-dimensional matrix as 2-dimensional. A matrix may not be redimensioned to contain more elements than it contained in the original declaration.

MAT ADDITION STATEMENT

The purpose of the MAT ADDITION statement is to add two numeric arrays together giving a third numeric array. The MAT ADDITION statement has the following format:

<n> MAT <matrix-1> = <matrix-2> + <matrix-3>

where <n> is the line number of the MAT statement and <matrix-1> is replaced by the sum of <matrix-2> and <matrix-3>. <Matrix-2> and <matrix-3> must have the same dimensions. <Matrix-1> must not have

smaller dimensions than <matrix-2>. If <matrix-1> is larger than <matrix-2> then <matrix-1> is redimensioned to the size of <matrix-2>. <Matrix-2> may appear on the left side of the equal sign.

Example:

```
100 DIM A(3,3) B(3,3), C(3,3)
110 MAT READ A,B
120 DATA 2,2,2
130 DATA 2,2,2
140 DATA 2,2,2
150 DATA 3,3,3
160 DATA 3,3,3
170 DATA 3,3,3
180 PRINT "MATRIX A IS"
200 MAT PRINT A;
210 PRINT
220 PRINT "MATRIX B IS"
240 MAT PRINT B;
250 PRINT
260 PRINT "MATRIX C IS"
280 MAT C = A + B
290 MAT PRINT C;
999 END
```

Execution of the above example causes the following to be printed:

```
MATRIX A IS
 2  2  2
 2  2  2
 2  2  2

MATRIX B IS
 3  3  3
 3  3  3
 3  3  3

MATRIX C IS
 5  5  5
 5  5  5
 5  5  5
```

MAT ASSIGNMENT STATEMENT

The purpose of the MAT ASSIGNMENT statement is to move the elements from one matrix to another. The MAT ASSIGNMENT statement has the following format:

<n> MAT <matrix-1> = <matrix-2>

where <n> is the line number of the MAT ASSIGNMENT statement. <Matrix-1> must not be smaller than <matrix-2>. If <matrix-1> is larger than <matrix-2>, <matrix-1> is redimensioned after the elements have been moved.

Example:

```
05 DIM B(4,4)
10 MAT READ A(2,2)
15 DATA 1,2,3,4
20 MAT B = A
25 MAT PRINT B;
99 END
```

Execution of the above example causes the following to be printed:

```
1  2
3  4
```

MAT CON STATEMENT

The purpose of the MAT CON statement is to initialize all of the data elements of a specified matrix to the numeric constant, ONE. The MAT CON statement may also be used to specify the dimensions of the matrix. The MAT CON statement has the following format:

<n> MAT <matrix-1> = CON (<bound>,<bound>)

where <n> is the line number of the MAT CON statement, <matrix-1> is the name of the matrix to be initialized, <bound> may be any numeric expression to specify the dimensions of <matrix-1>. If <bound> is not specified no, redimensioning occurs and the size of the matrix remains the same.

A matrix may not be redimensioned to have more elements than were specified in the DIM statement for the matrix, nor may the number of dimensions be changed.

Example:

```
10 DIM A(3,5)
20 MAT A = CON
30 MAT PRINT A;
40 END
```

Execution of the above example causes the following to be printed:

```
1  1  1  1  1
1  1  1  1  1
1  1  1  1  1
```

MAT IDN STATEMENT

The purpose of the MAT IDN statement is to zero the matrix and place ones on the main diagonal. The MAT IDN statement has the following format:

<n> MAT <matrix-1> = IDN (<bound>,<bound>)

where <n> is the line number of the MAT IDN statement. <Matrix-1> must be a square matrix: it must have two subscripts and have its number of rows equal to its number of columns. <Bound>, if used, may be any expression.

Example:

```
10 DIM X(3,3)
20 MAT X=IDN
30 MAT Y=IDN(4,4)
40 PRINT X;
50 PRINT Y;
60 END
```

Execution of the above example causes the following to be printed:

```
1  0  0
0  1  0
0  0  1

1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1
```

MAT INPUT STATEMENT

The purpose of the MAT INPUT statement is to cause the data elements of a matrix to be read from an external device. The MAT INPUT statement has the following format:

<n> MAT INPUT [#<numeric-expression>,]<list>

where <n> is the line number of the MAT INPUT statement, <numeric-expression> is any arithmetic expression giving the file number, and <list> is one or more numeric or string matrices.

Examples:

```
10 MAT INPUT A(12,7)
20 MAT INPUT A$(5)
30 MAT INPUT V
40 MAT INPUT #0, A(6,5)
```

MAT INV STATEMENT

The purpose of the MAT INV statement is to determine the inverse of the specified matrix and to place the result in the designated matrix. The inverse of a matrix is analagous to the reciprocal of a numeric constant. The MAT INV statement has the following format:

<n> MAT <matrix-1> = INV (<matrix-2>)

where <n> is the line number of the MAT INV statement and <matrix-1> is the name of the matrix where the results of the inverse are placed (must be 2-dimensional). <Matrix-2> must be enclosed in parentheses and must be a square matrix.

The same matrix name may not appear on both sides of the equal sign. <Matrix-1> may not be smaller than <matrix-2>. If <matrix-1> is larger than <matrix-2>, then <matrix-1> is redimensioned to the size of <matrix-2>. A singular matrix is not inverted and causes a run-time warning to be printed. The determinant of the singular matrix is set to 0. The value returned by the INV function is undefined if the argument was a singular matrix.

The DET function may be used to return the value of the determinant of the matrix whose inverse was last computed.

Example:

```
100 DIM A(3,3), B(3,3)
110 MAT READ A
120 DATA 1,2,3,4,5,6,7,8,8
130 MAT PRINT A;
140 MAT B = INV(A)
150 MAT PRINT B,
160 MAT C = A * B
170 MAT PRINT C
180 PRINT
190 PRINT "DETERMINANT IS"; DET
999 END
```

Execution of the above example causes the following to be printed:

```
 1   2   3
 4   5   6
 7   8   8

-2.66667          2.666667        -1
 3.33333         -4.33333          2
-1                2.              -1

 1               -7.45058E-09      0
 1.49023E-08      1.               0
 0.              -2.98023E-09      1

 DETERMINANT IS  3
```

MAT MULTIPLICATION STATEMENT

The MAT MULTIPLICATION statement performs algebraic multiplication on two matrices and assigns the result to another matrix. The MAT MULTIPLICATION statement has the following format:

<n> MAT <matrix-1> = <matrix-2> * <matrix-3>

where <n> is the line number of the MAT MULTIPLICATION statement, <matrix-1> is the matrix to which the result of the multiplication is assigned, and <matrix-2> and <matrix-3> are the two matrices that are multiplied together. <Matrix-1> must not be smaller than <matrix-2>. If <matrix-1> is larger than <matrix-2>, <matrix-1> is redimensioned to the size of <matrix-2> after the new values are assigned. All three matrices must be 2-dimensional. The same numeric matrix may not appear on both left and right sides of the equal sign.

Although the matrices to be multiplied must be 2-dimensional, a vector can be represented as a row vector or a column vector (using two dimensions) so that it can be used in a multiplication. A row vector must be dimensioned (n,1) to indicate a row vector consisting of n data elements. A column vector must be dimensioned (1,m) to indicate a column vector consisting of m data elements.

Example 1: Matrix Multiplication

```
100 DIM C(2,2)
110 MAT READ A(2,3) B(3,2)
120 DATA 2,2,2,2,2,2
130 DATA 3,3,3,3,3,3
140 MAT C = A * B
150 PRINT "MAT A ="
160 MAT PRINT A;
170 PRINT
180 PRINT "MAT B ="
190 MAT PRINT B;
200 PRINT
210 PRINT"MAT C = "
220 MAT PRINT C;
230 END
```

Execution of the above example causes the following to be printed:

```
MAT A =
 2   2   2
 2   2   2

MAT B =
 3   3
 3   3
 3   3

MAT C =
 18   18
 18   18
```

Example 2: Matrix Multiplication with Vectors

```
10 MAT READ A(1,2),B(3,1)
20 MAT C = A * B
30 MAT PRINT C
40 DATA 5,5,5,10,10,10
99 END
```

Execution of the above example causes the following to be printed:

```
150
```

MAT NUL$ STATEMENT

The purpose of the MAT NUL$ statement is to generate a string array whose elements are all null strings. The MAT NUL$ statement has the following format:

<n> MAT <matrix-1) = NUL$ (<bound>,<bound>)

where <n> is the line number of the MAT NUL$ statement, <matrix-1> is the matrix to be generated, and <bound> is the size of the array to be generated or redimensioned. <Matrix-1> can be either a single or double-dimensioned array. If <bound> is not present, the size of <matrix-1> remains the same.

Example:

```
10 MAT A$ = NUL$
20 MAT B$ = NUL$ (5)
30 MAT C$ = NUL$ (5,6)
```

MAT PRINT STATEMENT

The purpose of the MAT PRINT statement is to print an array row by row (except for row and column 0). The MAT PRINT statement has the following format:

<n> MAT PRINT [#<numeric-expression>,]<list>

where <n> is the line number of the MAT PRINT statement, <numeric-expression> is any arithmetic expression giving the file number, and <list> is a list of matrices to be printed.

If a file number is not specified, then the output is directed to the terminal, when compilation is performed from CANDE, or to the line printer, if the compilation is done from cards.

The rules discussed under the PRINT statement are also applicable to the MAT PRINT statement.

A blank line is printed after each array in the <list>.

Example 1:

```
10 MAT READ A$(3,3)
20 MAT PRINT A$;
30 DATA ONE, "2", THREE, "4", FIVE, "6", SEVEN, "8", NINE
40 END
```

Execution of the above example causes the following to be printed:

```
ONE2THREE
4FIVE6
SEVEN8NINE
```

Example 2:

```
10 MAT READ A$(3,3)
20 MAT PRINT A$,
30 DATA ONE, "2", THREE, "4", FIVE, "6", SEVEN, "8", NINE
40 END
```

Execution of the above example causes the following to be printed:

```
ONE             2               THREE
4               FIVE            6
SEVEN           8               NINE
```

Example 3:

```
100 MAT READ A (4,4)
110 MAT PRINT A
115 PRINT
120 MAT PRINT A,
125 PRINT
130 MAT PRINT A;
140 DATA 11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44
150 END
```

Execution of the above example causes the following to be printed:

```
 11              12              13              14
 21              22              23              24
 31              32              33              34
 41              42              43              44

 11              12              13              14
 21              22              23              24
 31              32              33              34
 41              42              43              44

 11  12  13  14
 21  22  23  24
 31  32  33  34
 41  42  43  44
```

MAT PRINT USING STATEMENT

The purpose of the MAT PRINT USING statement is to produce formatted matrices. The MAT PRINT USING statement has the following format:

<n> MAT PRINT [#<numeric-expression>,] USING <image>, <list>

where <n> is the line number of the MAT PRINT statement, <numeric-expression> is any arithmetic expres-

sion giving the file number, <image> is as described under IMAGE statement, and <list> is one or more names of matrices to be printed.

If a file number is specified, then the output is sent to a disk file; otherwise, output is sent to the line printer or a remote terminal, depending on the medium used for the compilation. The rules discussed under PRINT USING statement are also applicable to the MAT PRINT USING statement.

A blank line is printed after each array in the <list>.

Example:

```
120:### ### ### ### ### ### ### ### ### ###
130 FOR I = 1 TO 10
140 FOR J = 1 TO 10
150 LET A(I,J) = I * J
160 NEXT J
170 NEXT I
180 PRINT "THIS WAS PRINTED USING THE MAT PRINT STATEMENT"
190 MAT PRINT A;
195 FOR I = 1 TO 4
196 PRINT
197 NEXT I
200 PRINT "THIS WAS PRINTED USING THE MAT PRINT USING STATEMENT"
210 MAT PRINT USING 120,A
240 END
```

Execution of the above example causes the following to be printed:

```
THIS MATRIX WAS PRINTED USING THE MAT PRINT STATEMENT
 1   2   3   4   5   6   7   8   9   10
 2   4   6   8   10  12  14  16  18  20
 3   6   9   12  15  18  21  24  27  30
 4   8   12  16  20  24  28  32  26  40
 5   10  15  20  25  30  35  40  45  50
 6   12  18  24  30  36  42  48  54  60
 7   14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90  100



THIS MATRIX WAS PRINTED USING THE MAT PRINT USING STATEMENT
  1   2   3   4   5   6   7   8   9  18
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  26  40
  5  10  15  20  25  30  35  40  45  50
```

MAT READ STATEMENT

The purpose of the MAT READ statement is to read a matrix of specified size from data statements or a file. The MAT READ statement has the following format:

<n> MAT READ [#<numeric-expression>,] <list>

where <n> is the line number of the MAT READ statement, <numeric-expression> is any arithmetic expression giving the file number, and <list> is one or more matrices to be read from data statements or external files.

Matrices are filled row by row until the matrix is full or until the data list is emptied.

Example 1:

```
10 MAT READ A, B, C
20 MAT READ A(12,15),B(25,25), C(50), D$(20), E$(50,60)
30 MAT READ A$, B$, C$
```

When the DIM statement is used to specify the bounds of the matrix, the MAT READ statement used may be without a bound specification to read a matrix of the size specified in the associated DIM statement.

Example 2:

```
10 DIM A(4,12)
15 MAT READ A
20 MAT PRINT A;
30 DATA 1,2,3,4,5,6,7,8,9,10,11,12
40 DATA 1,2,3,4,5,6,7,8,9,10,11,12
50 DATA 1,2,3,4,5,6,7,8,9,10,11,12
60 DATA 1,2,3,4,5,6,7,8,9,10,11,12
99 END
```

Execution of the above example causes the following to be printed:

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

The DIM statement may be used to specify the maximum bounds of the matrix during the execution of the program, whereupon the MAT READ statement may specify the exact bounds of the matrix for the current read operation. The specification of the exact bounds in the MAT READ statement may be a numeric expression.

Example 3:

```
10 DIM A(4,12)
15 MAT READ A(4,4)
20 MAT PRINT A;
30 DATA 1,2,3,4,5,6,7,8,9,10,11,12
40 DATA 1,2,3,4,5,6,7,8,9,10,11,12
50 DATA 1,2,3,4,5,6,7,8,9,10,11,12
60 DATA 1,2,3,4,5,6,7,8,9,10,11,12
99 END
```

Execution of the above example causes the following to be printed:

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

In the MAT READ statement, data elements enter the matrix in a row-by-row fashion; that is, the first row of the matrix is filled with data elements then the second row is filled, and so on.

MAT SCALAR MULTIPLICATION STATEMENT

The purpose of the MAT SCALAR MULTIPLICATION statement is to multiply each element of a specified matrix by a specified multiplier and to place the result in the designated matrix. The MAT SCALAR MULTIPLICATION statement has the following format:

<n> MAT <matrix-1> = (<multiplier>) * <matrix-2>

where <n> is the line number of the MAT SCALAR MULTIPLICATION statement, <matrix-1> is the matrix to which the product is assigned, <multiplier> may be any numeric expression, and <matrix-2> is the multiplicand. <Matrix-1> must not be smaller than <matrix-2>. If <matrix-1> is larger than <matrix-2>, then <matrix-1> is redimensioned to the same size as <matrix-2>. <Matrix-2> may appear on both sides of the equal sign.

Example 1:

```
10 DIM A(3,3) B(3,3)
20 MAT READ A
30 DATA 2,2,2
40 DATA 2,2,2
50 DATA 2,2,2
60 PRINT "MATRIX A IS"
70 MAT PRINT A;
80 PRINT
90 PRINT "MATRIX B IS"
100 MAT B = (2) * A
110 MAT PRINT B;
120 PRINT
999 END
```

Execution of the above example causes the following to be printed:

```
MATRIX A IS
 2   2   2
 2   2   2
 2   2   2

MATRIX B IS

 4   4   4
 4   4   4
 4   4   4
```

Example 2:

```
100 DIM B(2,12)
110 MAT READ A(2,12)
120 DATA 1,2,3,4,5,6,7,8,9,10,11,12
130 DATA 13,14,15,16,17,18,19,20,21,22,23,24
140 PRINT "MAT A="
150 MAT PRINT USING 190, A
160 MAT B = (2) * A
170 PRINT "MAT B = "
180 MAT PRINT USING 190,B
190:## ## ## ## ## ## ## ## ## ## ## ##
200 END
```

Execution of the above example causes the following to be printed:

```
MAT A =
 1  2  3  4  5  6  7  8  9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24

MAT B =
 2  4  6  8 10 12 14 16 18 20 22 24
26 28 30 32 34 36 38 40 42 44 46 48
```

MAT SUBTRACTION STATEMENT

The purpose of the MAT SUBTRACTION statement is to determine the arithmetic difference between two specified matrices and to place the result in the designated matrix. The MAT SUBTRACTION statement has the following format:

<n> MAT <matrix-1> = <matrix-2> - <matrix-3>

where <n> is the line number of the MAT SUBTRACTION statement. <Matrix-3> is subtracted from <matrix-2> element by element, and the result is assigned to <matrix-1>. <Matrix-2> and <matrix-3> must have the same dimensions. <Matrix-1> may appear on both sides of the equal sign and must have dimensions equal to or greater than <matrix-2> and <matrix-3>. If greater, <matrix-1> is redimensioned to conform to the dimensions of <matrix-2> and <matrix-3>.

Example:

```
100 DIM A(3,3) B(3,3), C(3,3)
110 MAT READ A,B
120 DATA 2,2,2
130 DATA 2,2,2
140 DATA 2,2,2
150 DATA 3,3,3
160 DATA 3,3,3
170 DATA 3,3,3
180 PRINT "MATRIX A IS"
200 MAT PRINT A;
210 PRINT
220 PRINT "MATRIX B IS"
240 MAT PRINT B;
250 PRINT
260 PRINT "MATRIX C IS"
280 MAT C = A - B
290 MAT PRINT C;
999 END
```

Execution of the above example causes the following to be printed:

```
MATRIX A IS
 2  2  2
 2  2  2
 2  2  2

MATRIX B IS
 3  3  3
 3  3  3
 3  3  3

MATRIX C IS
-1 -1 -1
-1 -1 -1
-1 -1 -1
```

MAT TRN STATEMENT

The purpose of the MAT TRN statement is to interchange the rows and columns of the specified matrix and to place the results in the designated matrix. The first row becomes the first column, the second row becomes the second column, and so on. The MAT TRN statement has the following format:

<n> MAT <matrix-1> = TRN (<matrix-2>)

where <n> is the line number of the MAT TRN statement. <Matrix-1> is where the result of the transposition is to be placed, and <matrix-2> is the matrix to be transposed. The dimensions of <matrix-1> and <matrix-2> must be such that the two matrices conform to the matrix transposition operation. In order to conform in this manner, the number of rows in <matrix-1> must be equal to or greater than the number of columns in <matrix-2>, and the number of columns in <matrix-1> must be equal to or greater than the number of rows in <matrix-2>. If either or both are greater, <matrix-1> is redimensioned accordingly.

Example:

```
100:## ##
110:## ## ## ## ## ## ## ## ## ## ## ##
120 DIM A(12,2)
130 MAT READ B(2,   12)
140 PRINT "MAT B = "
150 MAT PRINT USING 110,B
160 PRINT "MAT A = "
170 MAT A = TRN(B)
180 MAT PRINT USING 100,A
190 DATA 1,2,3,4,5,6,7,8,9,10,11,12
200 DATA 13,14,15,16,17,18,19,20,21,22,23,24
210 END
```

Execution of the above example causes the following to be printed:

```
MAT B =
 1  2  3  4  5  6  7  8  9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24

MAT A =

 1 13
 2 14
 3 15
 4 16
 5 17
 6 18
 7 19
 8 20
 9 21
10 22
11 23
12 24
```

MAT WRITE STATEMENT

The purpose of the MAT WRITE statement is to transmit a matrix to a designated disk file. The MAT WRITE statement has the following format:

<n> MAT WRITE #<numeric-expression>, <list>

where <n> is the line number of the MAT WRITE statement, <numeric-expression> is any arithmetic expression giving the file number, and <list> is one or more matrix names. If there is more than one matrix name in the <list>, each name must be separated by commas or semicolons.

The MAT WRITE statement produces a line-numbered file with the data elements separated by the delimiter for the file (normally a comma).

Example:

```
10 FILES MATA
20 MAT READ A(10)
25 SCRATCH #1
30 MAT WRITE #1,A;
40 RESTORE
50 MAT READ B(2,5)
55 APPEND #1
60 MAT WRITE #1,B;
70 DATA 1,2,3,4,5,6,7,8,9,10
99 END
```

Execution of the above example causes the following to be printed:

```
00100  1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 ,
00110  1 , 2 , 3 , 4 , 5 ,
00120  6 , 7 , 8 , 9 , 10 ,
```

MAT WRITE USING STATEMENT

The purpose of the MAT WRITE USING statement is to transmit a formatted matrix to a disk file with line numbers and with delimiters following the data items. The MAT WRITE USING statement has the following format:

<n> MAT WRITE #<numeric-expression> USING <image>, <list>

where <n> is the line number of the MAT WRITE USING statement, <numeric-expression> is any arithmetic expression giving the file number, <image> is as discussed under the heading IMAGE STATEMENT, and <list> is one or more matrix names separated by commas or semicolons.

The following example illustrates the difference between the MAT PRINT USING and the MAT WRITE USING statements:

```
10 DIM A(2,5)
20 FOR I = 1 TO 2
30 FOR J = 1 TO 5
40 A(I,J) = I*J
50 NEXT J
60 NEXT I
70 PRINT "MAT PRINT USING:"
80 MAT PRINT USING 130,A
90 PRINT
100 PRINT "MAT WRITE USING:"
110 MAT WRITE USING 130,A
120 STOP
130  :## ## ## ## ##
140 END
```

Execution of the above example causes the following to be printed:

```
MAT PRINT USING:

 1  2  3  4  5
 2  4  6  9  10

MAT WRITE USING:

 1,  2,  3,  4,  5,
 2,  4,  6,  8,  10,
```

Formatting by the IMAGE statement occurs only after the line number and between delimiters.

MAT ZER STATEMENT

The purpose of the MAT ZER statement is to numerically initialize the data elements of the specified matrix to the constant 0. In addition, the MAT ZER statement may be used to specify new bounds for the matrix. The MAT ZER statement has the following format:

<n> MAT <matrix-1> = ZER (<bound>,<bound>)

where <n> is the line number of the MAT ZER statement, <matrix-1> is the name of the matrix to be initialized, and <bound> is a numeric expression.

A matrix must not be redimensioned to have more elements than were originally declared, and the number of dimensions must not be changed.

Example:

```
10 DIM A(5,5)
20 MAT A = ZER
30 MAT PRINT A;
99 END
```

Execution of the above example causes the following to be printed:

```
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0

 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

APPENDIX A
BASIC CARD READER INPUT

GENERAL

The BASIC compiler, in conjunction with the Master Control Program (MCP), 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 source 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 source card deck is described in this appendix.

COMPILATION CARD DECK

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

1. COMPILE card.
2. Label equation card (optional).
3. MCP label card.
4. Compiler option control card (optional).
5. Source input cards.
6. END (end-of-file) card.

MCP control cards are made distinguishable from other cards by entering in column 1 an invalid character for 80-column cards or 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 free-form format in columns 2 through 72. The presence of a percent sign (%) in an MCP control card terminates the control information on that card and any information following the percent sign is treated as a comment by the MCP. Refer to the B 1800/ B 1700 Systems Software Operational Guide, form number 1068731, 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:

1. ?COMPILE <program-name> BASIC

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

2. ?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.

3. ?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 (a combination of 1 and 2 above).

4. ?COMPILE <program-name> BASIC SYNTAX

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

In the absence of a COMPILE card, the system operator can manually execute one of the COMPILE options through the console keyboard 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 compiler's 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
```

Compiler Option Control Card

The BASIC compiler option control card ($ sign following the line number) may be included in the source deck as an option. This control card is used to notify 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 for 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 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:

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

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

SINGLE
Causes the line printer listing specified by LIST or default to be printed in a single-spaced format. SINGLE is the default option and need not be specified.

DOUBLE
Causes the line printer listing specified by LIST or default to be printed in a double-spaced format.

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

NO
Each of the options 1 through 5 may be preceded with NO, which enables 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.

MARGIN <LPRS>[,<PPRS>]
<LPRS> is the logical record print size and defaults to 80 if the BASIC user is at a terminal. If specified, it must be greater than or equal to 11. <PPRS> is the physical print record size and must be greater than or equal to <LPRS>. This statement affects compilation only.

STACK
Specifies the number of words available for the numeric and string stacks. The default set by the compiler is 100 elements of 48 bits per element.

STRING SPACE
Specifies the number of data pages available for string concatenation, input, and other operations which generate new strings. A string space data page holds 512 characters. The default set by the compiler is 8. The data pages may be extended using the $STRINGSPACE control option. The maximum number of data pages allowed is 128.

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. The line number must start in column 1 and contain 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 printed on the line printer.

END Card

The END card designates the end-of-file of the source 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 source deck to cause the line printer 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 source deck, is 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
```

APPENDIX B
REMOTE BASIC

The B 1800/B 1700 BASIC compiler can be used to compile and execute programs which are submitted from a remote terminal. When these programs are submitted through the use of the Command AND Edit (CANDE) program, and the usercode/password file security system is being used, please refer to the B 1800/B 1700 CANDE User's Manual, form number 1090586, for complete information.

If the usercode/password file security system is not being used, the following conditions must be observed when submitting jobs to BASIC from CANDE.

1. All station names for stations from which programs are submitted are limited to three characters.
2. The file names for those stations which use BASIC must be in the following format:

   "F" <station name>

For example, if the station name is to be "AIR" then the file name would be "FAIR".

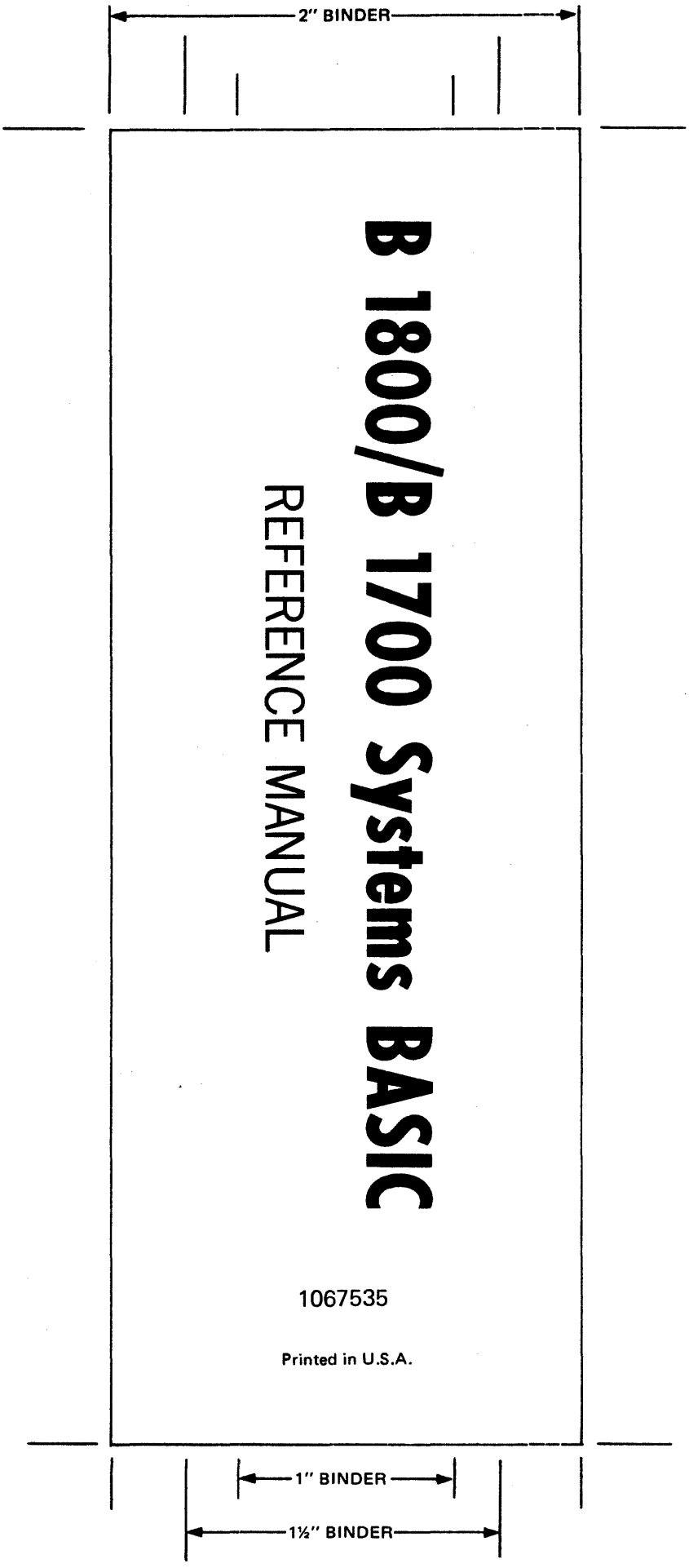
There are certain other CANDE/BASIC requirements. They are as follows:

1. A line number of 0 is allowed by CANDE, but is invalid in BASIC.
2. CANDE requires five decimal digits as a sequence number, but BASIC allows less than five digits as a sequence number.

INDEX

INDEX (Cont)

INDEX (Cont)


2" BINDER
B 1800/B 1700 Systems BASIC
REFERENCE MANUAL
1067535
Printed in U.S.A.
1" BINDER
1½" BINDER