Burroughs

B 5000/B 6000/B 7000 Systems

BASIC

REFERENCE MANUAL

(Relative to MARK 3.4 Release)

PRICED ITEM

Printed in U.S. America

July 1983

1160371

Burroughs

Hans Vlems

B 5000/B 6000/B 7000 Systems

BASIC

REFERENCE MANUAL

(Relative to MARK 3.4 Release)

PRICED ITEM

Printed in U.S. America

July 1983

1160371

The names, places, and/or events depicted herein are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to Documentation East , Burroughs Corporation, P.O. Box CB7, Malvern, Pennsylvania, 19355, U.S. America.

LIST OF EFFECTIVE PAGES

Page	Issue
iii	Original
iv	Blank
v thru xi	Original
xii	Blank
1-1 thru 1-4	Original
2-1 thru 2-14	Original
3-1 thru 3-18	Original
4-1 thru 4-20	Original
5-1 thru 5-29	Original
5-30	Blank
6-1 thru 6-17	Original
6-18	Blank
7-1 thru 7-51	Original
7-52	Blank
A-1 thru A-6	Original
B-1	Original
B-2	Blank
C-1 thru C-3	Original
C-4	Blank
D-1 thru D-3	Original
D-4	Blank
1 thru 4	Original

TABLE OF CONTENTS

Section	Title	Page
	INTRODUCTION	ix
	Scope of This Manual	x
	Related Documents	xi
1	RAILROAD DIAGRAMS	1-1
	General Description	1-1
	Railroad Components	1-2
	<required items>	1-2
	<optional items>	1-2
	<loops>	1-3
	<bridges>	1-4
2	LANGUAGE COMPONENTS	2-1
	General Description	2-1
	<language component>	2-1
	Basic Symbol	2-1
	<basic symbol>	2-1
	Delimiters	2-1
	Constants	2-2
	Numeric variables	2-3
	<variable>	2-3
	Strings	2-4
	<string>	2-4
	<string variable>	2-4
	Expressions	2-5
	Arithmetic expressions	2-5
	<arithmetic operator>	2-5
	Relational expressions	2-8
	<relational expression>	2-8
	<relational operator>	2-8
	String expressions	2-9
	Comparing String Expressions	2-10
	Boolean Expressions	2-11
	<boolean expession>	2-11
	<boolean term>	2-11
	Remark	2-12
	<remark>	2-12
	Use of Apostrophe or Percent Sign	2-13
	General BASIC Rules	2-13
3	ASSIGNMENT AND CONTROL LINES	3-1
	Assignment Lines	3-1
	LET Statement	3-1
	DIM Statement	3-3
	<DIM statement>	3-3
	<array name>	3-3
	<subscript>	3-3
	OPTION Statement	3-5
	SWAP Statement	3-6
	Control Lines	3-7

TABLE OF CONTENTS (CONT.)

Section	Title	Page
3 (Cont.)	GO TO Statement.	3-8
	IF Statement	3-8
	IF-THEN Statement.	3-9
	IF-THEN-ELSE statement	3-10
	ON Statement	3-11
	Program Loops.	3-12
	FOR and NEXT Statements.	3-13
	<FOR statement>	3-13
	<NEXT statement>.	3-13
	WHILE and WEND Statements.	3-17
	<WHILE statement>	3-17
	<WEND statement>	3-17
	STOP Statement	3-18
	END Statement.	3-18
4	I/O STATEMENTS	4-1
	Read and Data Statements	4-1
	<READ statement>.	4-1
	<DATA statement>.	4-1
	RESTORE Statement.	4-3
	INPUT Statement.	4-5
	PRINT Statement.	4-6
	Print formats.	4-7
	String Variables in Print Statements	4-10
	Output to a Terminal	4-10
	WRITE Statement.	4-11
	PRINT USING Statement	4-12
	WRITE USING Statement.	4-13
	IMAGE Statement	4-14
	<image element>	4-14
5	FUNCTIONS AND SUBPROGRAMS.	5-1
	General Description.	5-1
	Arithmetic Functions	5-1
	ABS Function	5-1
	BASE 10 LOG Function	5-2
	EXP Function	5-2
	INT Function	5-3
	LOG Function	5-4
	RND Function	5-4
	SGN Function	5-5
	SQR Function	5-5
	Trigonometric Functions.	5-6
	SIN Function	5-6
	COS Function	5-6
	TAN Function	5-7
	COT Function	5-7
	ATN Function	5-7
	String Functions	5-8
	EXT$ Function.	5-8
	LEFT Function.	5-9
	LEN Function	5-10

TABLE OF CONTENTS (CONT.)

Section	Title	Page
5 (Cont.)	REP$ Function.	5-10
	RIGHT Function	5-11
	SCN Function	5-12
	SPACE Function	5-13
	STR$ Function.	5-14
	VAL Function	5-15
	SYSTEM Functions	5-15
	ASC Function	5-16
	BCL Function	5-16
	CLK$ Function.	5-17
	DAT$ Function.	5-17
	IDA Function	5-17
	TIM Function	5-18
	Statement Functions and Subprograms.	5-19
	<DEF statement>	5-19
	<arithmetic function>	5-19
	<string function>	5-19
	Single Line Statement Function	5-21
	Multiple Line Statement Function	5-23
	FNEND Statement.	5-24
	CALL Statement	5-26
	Subprograms.	5-27
	GOSUB and RETURN Statements.	5-27
	<GOSUB statement>	5-27
	<RETURN statement>.	5-27
6	MATRICES	6-1
	General Description.	6-1
	Matrix I/O Statements.	6-1
	MAT READ Statement	6-1
	MAT INPUT Statement.	6-2
	MAT PRINT Statement.	6-3
	MAT WRITE Statement.	6-4
	MAT PRINT USING Statement.	6-4
	MAT WRITE USING Statement.	6-5
	MATRIX Functions	6-6
	CON, ZER, AND IDN Statements	6-6
	INV AND TRN Functions.	6-7
	<INV and TRN functions>	6-7
	DET Function	6-8
	NUM Function	6-8
	Dimensioning	6-10
	Matrix Arithmetic Statements	6-12
	<Matrix arithmetic operations>.	6-12
	Matrix Scalar Multiplication	6-14
	<matrix scalar multiplication>.	6-14
7	EXTERNAL DATA FILES AND FUNCTIONS.	7-1
	General Description.	7-1
	Declaring and Accessing Files.	7-1

Section	Title	Page
	FILES Statement	7-1
	FILE Statement	7-3
	<FILE statement>	7-3
	<file designator>	7-3
	<filename>	7-3
	EBCDIC and Binary Files	7-4
	FILE I/O Statements for EBCDIC Files	7-8
	FILE INPUT Statement	7-8
	FILE MAT INPUT Statement	7-10
	FILE READ Statement	7-12
	FILE MAT READ Statement	7-14
	Comparison of the READ and INPUT Statements	7-15
	FILE PRINT Statement	7-17
	FILE WRITE Statement	7-19
	FILE MAT PRINT AND MAT WRITE Statements	7-21
	<FILE MAT PRINT Statement>	7-21
	<FILE MAT WRITE Statement>	7-21
	FILE PRINT USING and WRITE USING Statements	7-23
	FILE MAT PRINT USING and MAT WRITE USING Statements	7-23
	File Manipulation Statements	7-24
	SCRATCH Statement	7-25
	RESTORE Statement	7-26
	IF END Statement	7-27
	IF MORE Statement	7-29
	BACKSPACE Statement	7-31
	APPEND Statement	7-33
	MARGIN Statement	7-34
	DELIMIT Statement	7-36
	Manipulation of Binary Files	7-37
	Storage of Data on Binary Files	7-37
	Creating Binary Files	7-38
	INPUT and OUTPUT Statements for Binary Files	7-38
	READ and WRITE Statements	7-38
	READ FORWARD Statement	7-41
	BACKSPACE$ Statement	7-42
	SETW Statement	7-43
	Summary of Statements Available to External Files	7-46
	FILE Functions	7-47
	HPS Function	7-47
	LCW Function	7-48
	LFW Function	7-49
	LIN Function	7-50
	VPS Function	7-51
A	COMPILER OPTIONS	A-1
B	USING BASIC FROM A REMOTE TERMINAL	B-1
C	ANSI 1978 STANDARD BASIC COMPILER OPTION	C-1
D	ERROR MESSAGES	D-1

INTRODUCTION

Beginner's All-purpose Symbolic Instruction Code (BASIC) is a language designed for a wide range of uses.

BASIC has six major categories: language components, assignment and control lines, I/O statements, functions, matrices, and files.

Language components form the foundation on which the entire BASIC language is built. Examples are letters, digits, and delimiters, which will be defined.

Assignment lines tell the compiler what values the variables contain. They also tell of the existence of subroutines. Control lines alter the order in which a program is executed.

I/O statements tell the computer that some value is to be input or output.

Functions are inherent programs the computer executes when it encounters certain words.

Matrices are two-dimensional arrays.

Files are stored programs.

These six components are used to form and store BASIC programs.

SCOPE OF THIS MANUAL

Purpose

This manual describes the BASIC programming language used on the B 7000/B 6000/B 5000 series of computer systems. It is a reference manual for the language.

Organization

Of the seven sections making up the main body of this manual, one is an introduction. The next six sections are organized according to the six major categories of BASIC. The four appendices offer information on internal data representation and compiler facilities. A breakdown by section and appendix follows:

Section 1 INTRODUCTION

describes the scope of this manual and the building blocks of a BASIC program.

Section 2 LANGUAGE COMPONENTS

defines the fundamental elements of BASIC.

Section 3 ASSIGNMENT AND CONTROL LINES

explains how values are assigned to variables, what a control statement is, and how it is used in BASIC.

Section 4 I/O STATEMENTS

defines the lines used to input and output data.

Section 5 FUNCTIONS AND SUBPROGRAMS

defines how arithmetic, trignometric, string, system, and user-defined functions are used, and how subprograms work.

Section 6 MATRICES

defines the matrix operations that may be used on one- and two-dimensional arrays.

Section 7 EXTERNAL FILES AND FUNCTIONS

examines external data files. This includes creating external files, file input and output, file manipulating, and file restoring. It also lists the file functions available.

Appendix A COMPILER OPTIONS

describes user-changeable options that control the processing of BASIC source input by the compiler.

Appendix B USING BASIC FROM A REMOTE TERMINAL

gives a few of the details needed to write and run a BASIC program from a remote terminal.

Appendix C THE ANSI 1978 STANDARD BASIC COMPILER OPTION

lists the differences caused by setting the $ SET ANSI compiler option.

Appendix D ERROR MESSAGES

lists possible error messages received when programming in BASIC.

RELATED DOCUMENTS

The following documents contain related information:

Publication	Form No.
B 7000/B 6000/B 5000 CANDE Reference Manual	5011398
B 7000/B 6000/B 5000 Input-Output Subsystem Ref. Manual	5001779

SECTION 1

RAILROAD DIAGRAMS

GENERAL DESCRIPTION

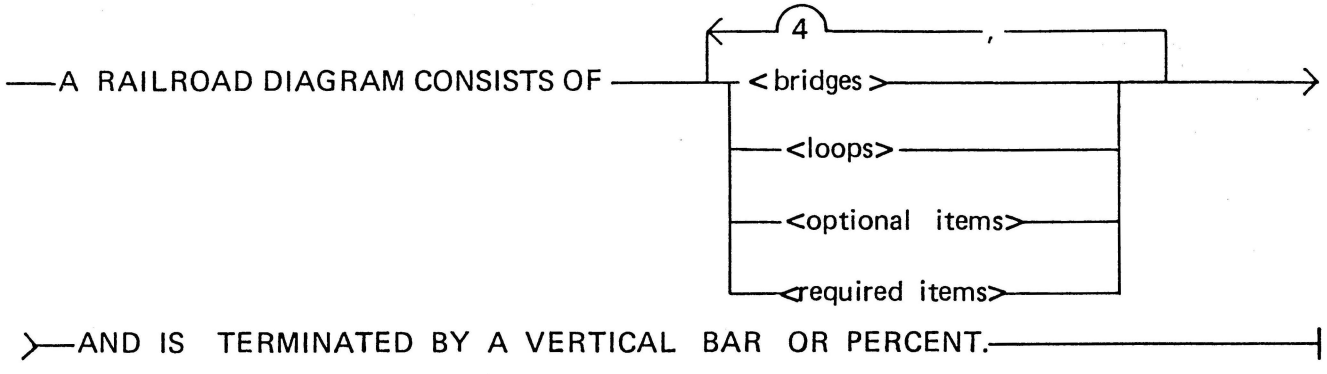
Railroad diagrams are graphic representations which show the syntax of the language.

Railroad diagrams are traversed left to right or in the direction of the arrowhead. Adherence to the limits illustrated by bridges produces a syntactically valid statement. Continuation from one line of a diagram to another is represented by a right arrow (>) appearing at the end of the current line and the beginning of the next line. The complete syntax diagram is terminated by a vertical bar (|) or a percent sign (%).

Items contained in broken brackets (<>) are syntactic variables which are defined in the manual or are information which the user is required to supply.

Uppercase items not enclosed in broken brackets must appear literally; the minimum abbreviations are underlined.

Example:



Some syntactically valid constructs that may be generated from the above diagram are as follows:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

A RAILROAD DIAGRAM CONSISTS OF <optional items> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

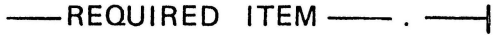
A RAILROAD DIAGRAM CONSISTS OF <optional items>, <required items>, <optional items>, <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

Railroad Components

<required items>

No alternate path through the railroad diagram exists for required items or required punctuation.

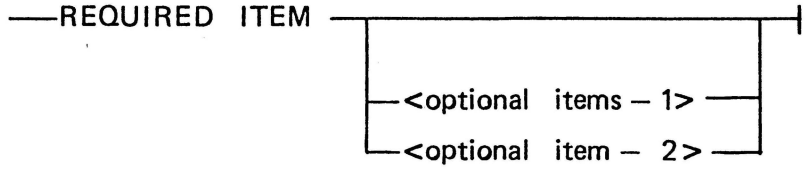
Example:



<optional items>

Items shown as a vertical list indicate that the user may make a choice of the items specified. An empty path through the list allows the optional item to be absent.

Example:



The following valid constructs may be generated from the preceding diagram:

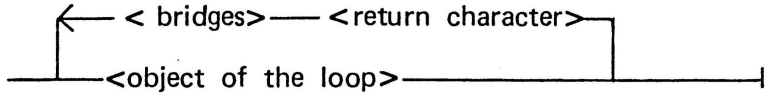
REQUIRED ITEM

REQUIRED ITEM <optional item-1>

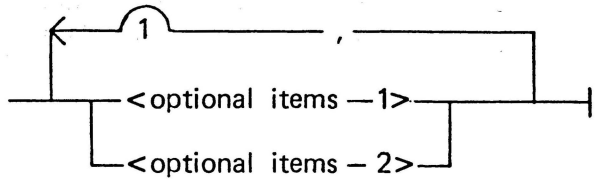
REQUIRED ITEM <optional item-2>

<loops>

A loop is a recurrent path through a railroad diagram and has the following general format:



Example:



Some valid constructs that may be generated from the preceding diagram are as follows:

<optional item-1>

<optional item-1>,<optional item-1>

<optional item-2>,<optional item-1>

The loop must be traversed in the direction in which the arrowhead points, and the limits specified by bridges cannot be exceeded.

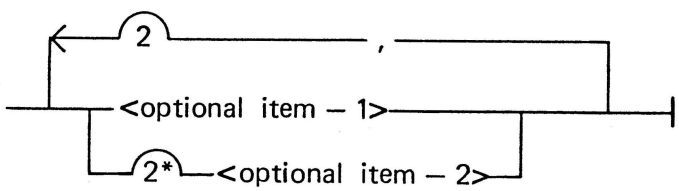
<bridges>

A bridge illustrates the minimum or maximum number of times a path may be traversed in a railroad diagram.

Two forms of bridges exist:

/n\	n is an integer that specifies the maximum number of times the path may be traversed.
/n*\	n is an integer that specifies the maximum number of times the path may be traversed. The asterisk (*) indicates that the path must be traversed at least once.

Example:



The loop may be traversed a maximum of two times, and the path for <optional item-2> must be traversed at least once but no more than twice.

Some valid constructs that may be generated from the preceding diagram are as follows:

<optional item-1>,<optional item-2>

<optional item-2>,<optional item-2>,<optional item-1>

<optional item-2>

SECTION 2

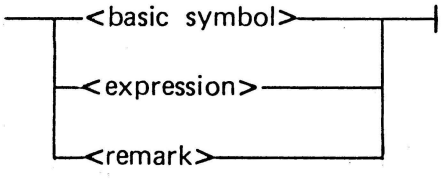
LANGUAGE COMPONENTS

GENERAL DESCRIPTION

The following is a description of BASIC fundamentals.

Syntax

<language component>



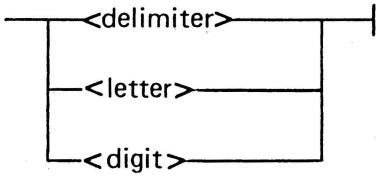
Semantics

All of the above words in brackets are discussed under separate headings in this section.

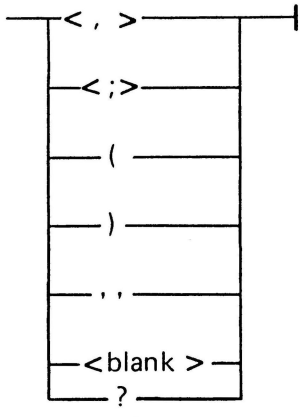
BASIC SYMBOL

Syntax

<basic symbol>



<delimiter>



Semantics

Delimiters

A comma separates the elements of a list. A semi-colon separates the elements of a compressed output line. Parentheses control the order in which an arithmetic expression is evaluated and also enclose the subscripts of an array. Quotes denote a string. Blanks have no significance unless enclosed in quotes, which causes the enclosed information to be printed out verbatim. The question mark is an invalid character when used as input.

<letter>

Any one of the capital letters A through Z.

<digit>

Any one of the arabic numerals 0 through 9.

Letters and digits can be used to form constants and variables.

Constants

Numeric constants can be expressed as integers, reals, or in scientific form.

An integer constant is a signed or unsigned number without a decimal point. Examples of this are 1, 123456, 12345678901, and -456.

A real constant is a signed or unsigned number with a decimal point. Examples of this are 1.0, 123.456, -4.56 and -1.2345678901. Numbers with an absolute value less than 1 but greater than 0 no longer have a 0 before the decimal point.

The scientific form allows very large or very small integer and real numbers to be expressed in a shorter form. For example, -.0000000000267 could be expressed as -2.67@-11 since @ means 'times 10 to the power of.' The "@-11" is known as the exponent part of the constant. The number -.0000000000267 can also be expressed as -2.67E-11. Finally, it can be expressed as -2.67-11, but only if it is assigned this value in an INPUT statement.

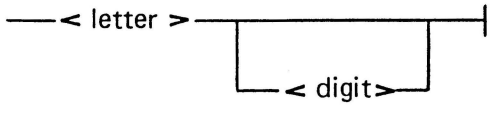
The computer considers all numeric constants to be real and automatically handles the decimal point.

The largest number that may be stored in the computer is 4.314@68: the smallest non-zero number that may be stored in the computer is 8.758@-47. A constant whose magnitude is less than machine infinitesimal is replaced with a zero, and a warning will be generated. A constant whose magnitude is greater than machine infinity will generate a warning that an overflow error will occur at run time, and such an error will occur. The following examples show some acceptable BASIC numeric constants.

1	1.234@11	.69	-1.64@42
123456	1@9	3.1415926536	-2.67@-40
12345678901	1.@-09	-456	-.500

Numeric Variables

<variable>

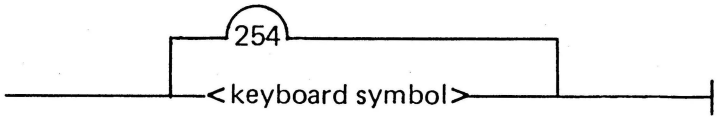


Numeric variables are a letter or a letter followed by a digit and they can represent any numeric value. Valid examples are E, A, X, M5, and P2. Invalid examples are 5A, AAA, 8J, PP, 5, r6, CP30 and P567. All numeric variables are initialized to zero by the compiler.

Strings

Syntax

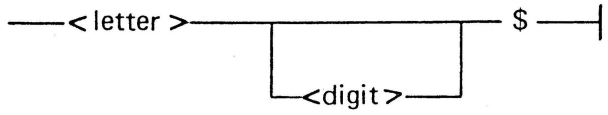
<string>



A keyboard symbol is any <letter>, <digit>, <delimiter> or any other key that can be found on the terminal keyboard, except the quote (").

A string constant is a list of 0 to 255 characters enclosed within quotes. All string variables are initialized to " " (the null string) by the compiler.

<string variable>



A string variable is a variable assigned the value of a string of characters.

Example:

```
100 A$ = "**$56**"
200 PRINT A$
```

when executed would look like this on the terminal:

```
**$56**
```

EXPRESSIONS

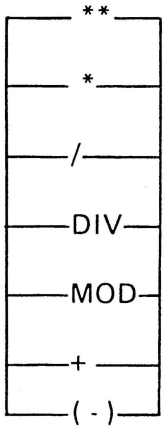
An expression is any constant, variable, or combination of these, separated by operators and/or parentheses. There are three types of expressions:

A. ARITHMETIC
B. RELATIONAL
C. STRING
D. BOOLEAN

Arithmetic Expressions

Syntax

<arithmetic operator>



Semantics

An arithmetic expression is an arithmetic variable or constant, or a line which computes a numeric value and uses an arithmetic operator.

These arithmetic operators are:

**	Exponentiation
*	Multiplication
/	Division
DIV	Integer division
MOD	Remaindering
+	Addition
-	Subtraction
-	Negation

Negation is the only arithmetic operator that needs only one operand (unary). All other operators require two variables (binary).

The DIV operator divides two numbers and returns an integer answer without the remainder. The MOD operator returns the remainder. For example, 9 DIV 4 is equal to 2, and 9 MOD 4 is equal to 1. This also illustrates the fact that A MOD B is equivalent to A-((A DIV B)*B).

Example:

```
If A=9 and B=4
A - ((A DIV B) * B) =
9 - ((9 DIV 4) * 4  =
9 - (2 * 4)         =
9 - 8               =
1.
```

Only one operator can separate two variables, except in the case of negation ,that is, B + (-3) is valid. No arithmetic operator is assumed to be performed. Examples of invalid arithmetic expressions are:

A++B	-the plus signs are operators that do not have a variable between them.
(A+2) (B+3)	-no multiplication is assumed between the expressions in parentheses.

The precedence order used in evaluating an arithmetic expression is as follows:

(Highest)	Exponentiation
	Multiplication, division, integer division, and remaindering
(Lowest)	Addition, subtraction, and negation

Operators of the same level in an expression are evaluated left to right. For example, A**B**C is evaluated (A**B)**C.

Parentheses may be used in an arithmetic expression to change the order in which operations are to be performed. Parentheses have precedence over all arithmetic operators in determining the order of evaluation. When nested parentheses occur, evaluation proceeds from the innermost set to the outermost set.

With most of the arithmetic operators of the same level, the order in which operations are performed is unimportant. For instance, X*Y/Z is equal to X/Z*Y. However when MOD and DIV are used, the left to right order of calculation for operators of equal precedence becomes very important.

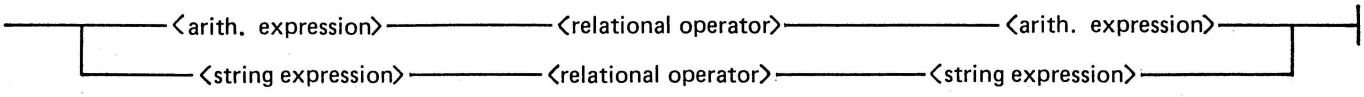
Example:

```
3*5 DIV 2 = 7
5 DIV 2*3 = 6
```

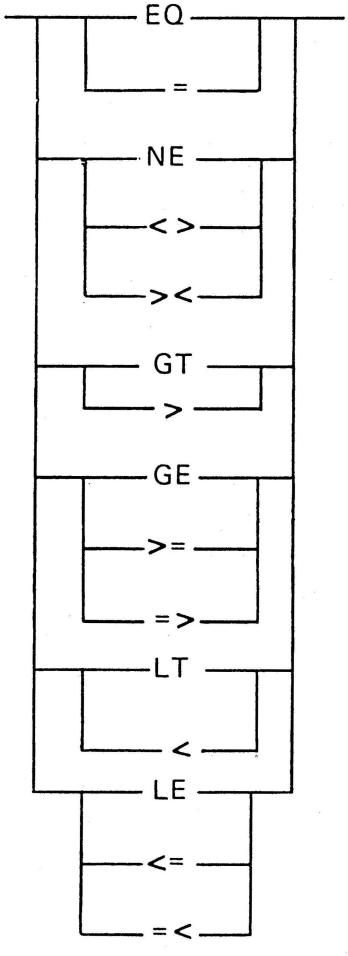
In such cases, the programmer must be especially careful to have the expression ordered to perform the intended calculation.

Relational Expressions

<relational expression>



<relational operator>



A relational expression is an arithmetic constant or variable, or a line which computes a value and uses a relational operator. These operators are:

EQ,=	Equal
NE,<>,><	Not equal
GT,>	Greater than
GE,>=,=>	Greater than or equal
LT,<	Less than
LE,<=,=<	Less than or equal

To prevent ambiguity, use periods around the operator. For example, later in this manual FNE will mean a function labeled 'E'. Periods (F.NE.) will distinguish this to mean 'F is not equal to.'

Relational expressions are evaluated as either TRUE or FALSE. Chains of relations are not permitted; for example, A .LT. B .LT. C is invalid.

String Expressions

A string expression can be a string constant or variable, or any combination of these concatenated by plus signs (+). The format is:

```
A$+B$+C$+. . .+I$
```

When one string is concatenated to another string, they are output with no blanks between them.

Example:

```
100 A$ = "HI"
200 B$ = "THERE"
300 C$ = A$ + B$
400 PRINT A$; B$; C$
500 END
```

When the above program is executed, the following is output:

```
HITHEREHITHERE
```

Comparing String Expressions

String expressions may be compared using the IF statement with a string expression on each side of the relational operator.

Examples:

```
10 IF A$ EQ "ZORBA" THEN 40
20 IF B$ NE C$ THEN 55
30 IF "MARGIN" LE P$(5) THEN 75
40 IF M$ EQ "06" THEN 20
```

The comparison is made on the characters according to the following collating sequence:

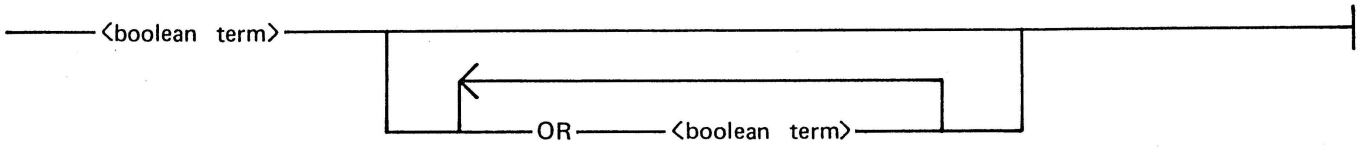
```
(HIGH)                                                    (LOW)
9...0 ZYX...CBA " = ' @ # : ? > % , / - ; ) * $ ] & ! + ( < . [
```

Comparison of strings begins with the left-most character and proceeds to the right until unequal characters are encountered (which establishes that one string is less than the other), or until one of the strings runs out of characters. A longer string is greater than a shorter string; for instance, "AB" is less than "ABC". Two strings can be equal only when they have the same number of like characters in the same sequence. The net effect of this is that strings arranged in ascending order are also in alphabetical order.

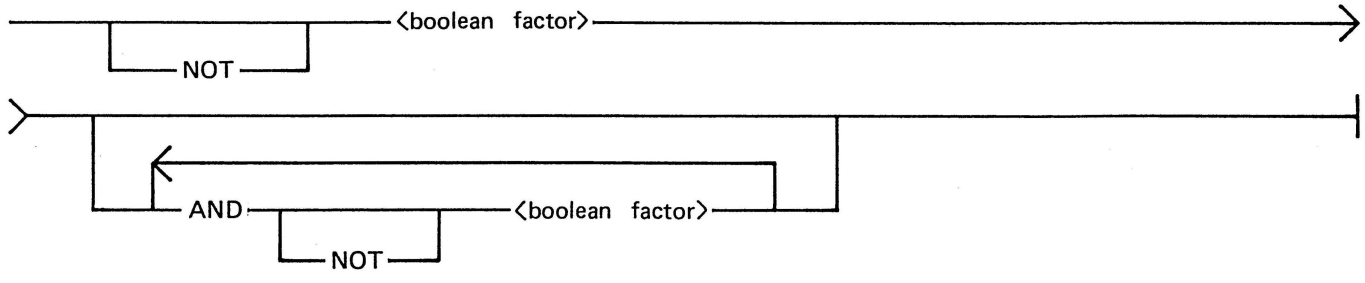
Boolean Expressions

Syntax

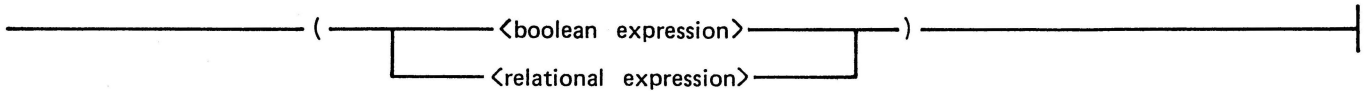
<boolean expression>



<boolean term>



<boolean factor>



Semantics

A boolean expression is an expression whose value is true or false. Blanks are not significant in BASIC, so use of AND, OR, and NOT will occasionally be ambiguous. Avoid this problem by using .AND., .OR. and .NOT. as synonyms for AND, OR, and NOT. Even with these synonyms an expression can still be ambiguous.

Example:

```
1 = 1. AND .1 = 1
```

could be interpreted as:

```
(1 = 1) .AND. (1 = 1)
```

or as (1 = .1) AND (.1 = 1)

Conflicts like this are resolved by using the first interpretation.

Parentheses and NOT have precedence over AND, which has precedence over OR.

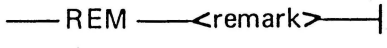
Example:

```
.NOT. A>B is equivalent to .NOT. (A>B)
```

REMARK

Syntax

<remark>



Semantics

An important part of any computer program is the description of what it does, and what data should be supplied. One of the ways a program can be documented is by supplying remarks along with the program itself. The REM (remark) statement provides this capability. For example, the following REM statements could be included in a program to calculate some students' final grades.

```
10 REM FINAL MARK CALCULATION PROGRAM
20 REM W IS THE WEIGHT OF EACH EXAM
30 REM M IS THE STUDENTS GRADE
40 REM BEGIN TO READ DATA
50 REM CALCULATE FINAL GRADE
```

Use of Apostrophe or Percent Sign

The apostrophe character or the percent sign can also be used in a BASIC program statement to indicate that the remainder of the line is a comment. Anything in a BASIC statement following an apostrophe will be treated as an explanatory remark.

Example:

```
A=7 'THIS IS A REMARK
```

General BASIC Rules

A statement cannot continue from one line to the next. Each line of the program must begin with a unique number which is called a line number. These numbers serve to identify each statement in the program. A line number may be between 1 and 9999, and multiple line numbers are not permitted. When the computer executes a BASIC program, the statements are executed in the order of increasing line numbers except where this execution sequence is interrupted by a control statement.

Each line of a program should be no more than 72 characters long, including the line number. If a line is more than 72 characters long, only the first 72 characters will be read and the rest of the line will be ignored.

Programs containing too many variables and/or image statements cause an error message and an aborted compilation. The number of allowable variables and image statements has been doubled.

For example, to calculate compound interest from the formula

```
Amount = Principal (1 + rate) times itself n times,
where n is the number of years
```

execute the following program:

```
100 LET P=1100
200 LET R=.05
300 LET N=2
400 LET A=P*(1+R)**N
500 PRINT "P =";P, "R =";R, "N =";N, "A =";A
600 END
```

The result produced by this program is:

```
P = 1100      R = .05       N = 2          A = 1212.75
```

The equal sign (=) in BASIC signifies replacement, not equality. For example, statement 100 means "assign the value of 1100 to P", or "replace the current value of P by 1100".

Since statements are executed in numerical sequence, it is not necessary to code the program in exact logical sequence. When the program is read by the computer, it is edited to determine if the statements are arranged in ascending numerical sequence. If there are statements of the program that are out of sequence, the computer will place them in their proper order. For example, if in the example above, statement 300 had been omitted, and the statements following it had been assigned line numbers of 300, 400, and 500; then it could be entered at the end of the program as statement 250. When the program is edited by the computer, statement 250 is placed between statement 200 and 300 and, when the program is executed, it produces the same results as in the previous example.

A program may contain up to 9999 statements.

An explanation of the compiler options that may be used while compiling a BASIC program are listed in appendix A.

If the BASIC program is to be written from a remote terminal device, refer to appendix B.

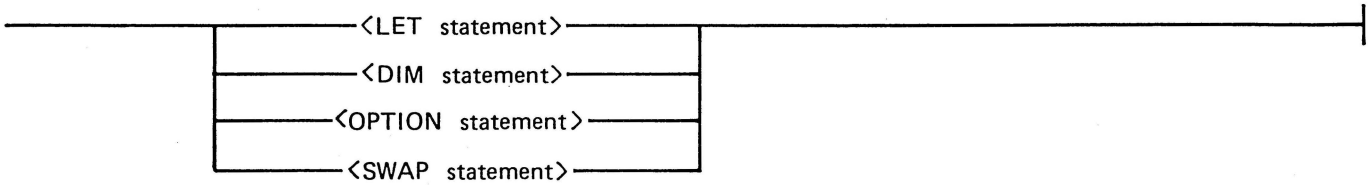
SECTION 3

ASSIGNMENT AND CONTROL LINES

ASSIGNMENT LINES

Syntax

<assignment line>



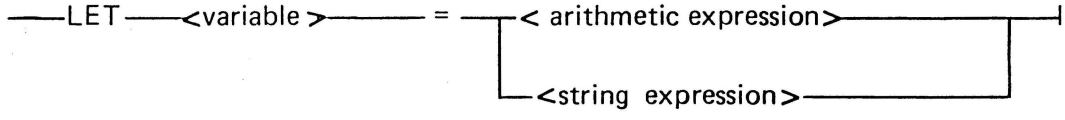
Semantics

The assignment of variables can be accomplished with a LET, READ, or INPUT statement.

LET STATEMENT

Syntax

<LET statement>



Semantics

For example, the statement

```
100 LET A$="XYZ"
```

assigns the string "XYZ" to A$. The statement

```
15 LET Y=2*A+B
```

causes the computer to double the value of A, add the value of B, and assign the result to Y. The values assigned to A and B remain the same.

The same value can be assigned to more than one variable in a single assignment statement. For example:

```
10 LET A=B=C=0
```

assigns the value of 0 to variables A, B, and C, and has the same effect as:

```
10 LET A=0
20 LET B=0
30 LET C=0
```

Similarly,

```
10 LET A$=B$=C$="A"
```

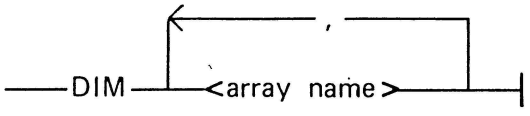
is permissible. When using an assignment line like this, it is important that all of the variables used are of the same type, either arithmetic or string. For example, 10 LET A=B=C$=0 is invalid and will produce a fatal error.

Values may be assigned to variables without using the word "LET". For example, 10 LET A=B is the same as 10 A=B.

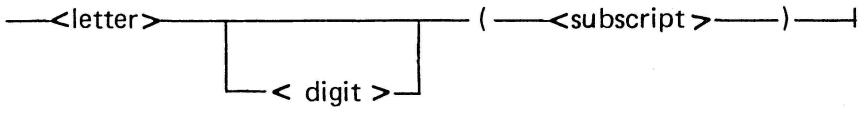
DIM Statement

Syntax

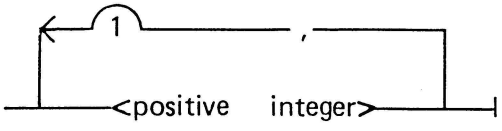
<DIM statement>



<array name>



<subscript>



Semantics

Often it is convenient to store data in a table rather than assigning a variable to each piece of data. This table is called an array, and the individual items in an array are called elements. An arithmetic array can have either one or two dimensions, but a string array can have only one dimension.

BASIC will provide 10 elements for any variable used as a single-dimensioned array, and 10x10 or 100 elements for any variable used as a double-dimensioned array. If less than or more than 10 elements are desired for a dimension of an array, the array must be declared in a DIM statement.

An array name and a variable may be given the same name and remain two distinct variables.

The integers in the array name specify the maximum number of elements allowed in the array. The upper bound of an array may be changed at compilation time after an array has already been referenced or declared in a DIM statement by another DIM statement. The number of dimensions for the array cannot be changed at compilation time. Beware of invalid subscript values, as the last DIM statement encountered in a program determines the size of that array at run time. An array may be redimensioned at run time through the use of certain matrix operations. (See section 6).

Examples:

```
10 DIM Q(60)              Declares an array of 60 elements.
20 DIM A(30,30)           Declares an array of 30X30 elements.
30 DIM A(15,20),B(20)     Declares two arrays.
40 DIM A(2,2)             Declares an array of 2X2 elements.
```

Each element in an array is referenced by the array name followed by a subscript containing the element position. The positions in a one-dimensional array are numbered from one to N, with N being the length of an array. Word 0 cannot be accessed.

The two-dimensional array has a subscript for each dimension. The rows of the array are represented by the first subscript and the columns by the second subscript. Locations in a two-dimensional array are numbered from (1,1) to (M,N) where M is the number of rows and N is the number of columns.

Examples:

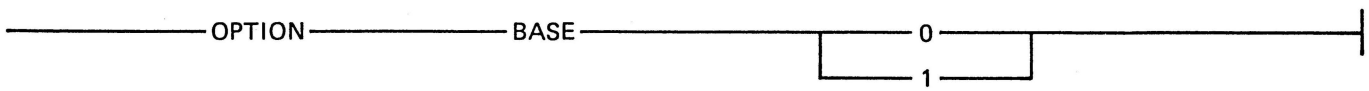
```
A(5)
B(2,3)
```

The first example specifies the fifth element of a one-dimensional array A. The second example references the element in the second row and third column of a two-dimensional array B.

OPTION Statement

<OPTION statement>

Syntax



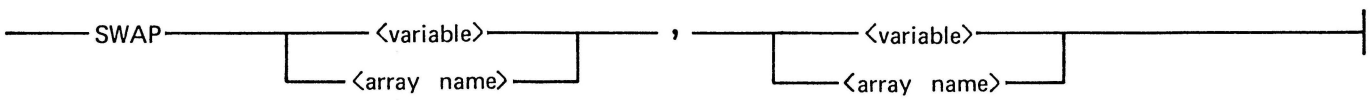
Semantics

Through the OPTION statement, arrays are declared to have a lower bound of zero or one. The Burroughs default is one; the default for an ANSI78 program is zero. If an array A(5) is declared with option base 0, then 6 words (0 to 5) will be allocated. If A(5) is declared with option base 1, then five words (1 to 5) will be allocated. An OPTION statement must occur in a lower-numbered line than any other DIM statement or any reference to an array. If the option base is 1, no upper bound of 0 may be specified. There may be at most one OPTION statement.

SWAP Statement

<SWAP statement>

Syntax



Semantics

The SWAP statement causes the values of variables and the values of array names to be interchanged. They must be of the same type. Contents of entire arrays may not be SWAPed.

Examples:

```
100 C=5
200 K(8)=8
300 PRINT C;K(8);"BEFORE THE VALUES ARE SWAPPED"
400 SWAP C,K(8)
500 PRINT C;K(8);"AFTER THE VALUES ARE SWAPPED"
600 END
```

When the above program is executed, the following is output:

```
 5  8 BEFORE THE VALUES ARE SWAPPED
 8  5 AFTER THE VALUES ARE SWAPPED
```

```
100 C$="MARCH"
200 K$="FEBRUARY"
300 PRINT C$,K$
400 SWAP C$,K$
500 PRINT C$,K$
600 END
```

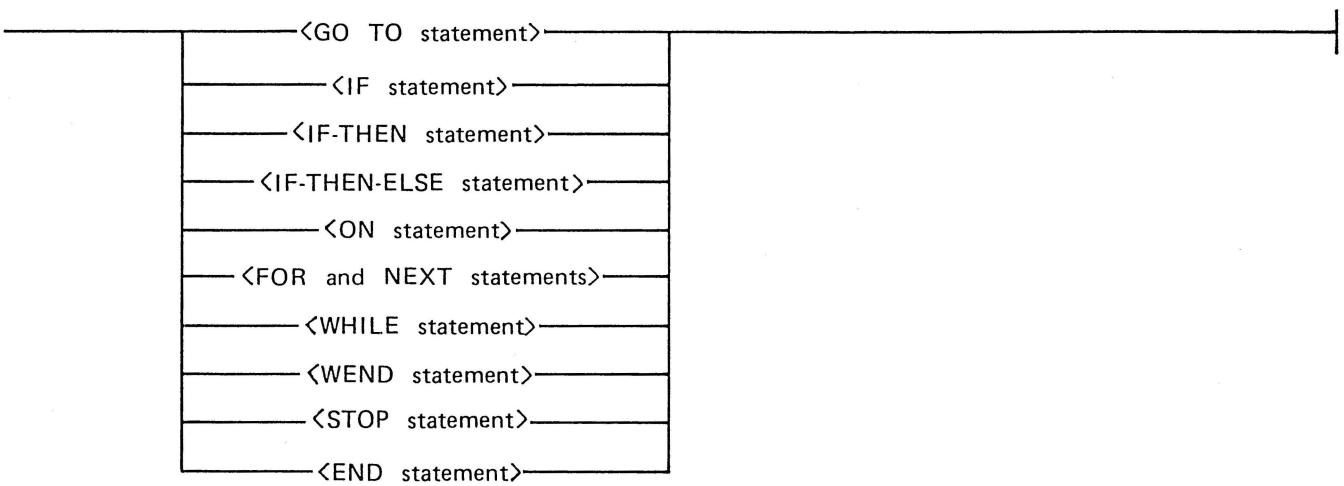
When the above program is executed, the following is output:

```
MARCH          FEBRUARY
FEBRUARY       MARCH
```

CONTROL LINES

Syntax

<control line>



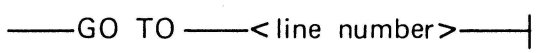
Semantics

Control lines can have any one of three functions. They can change the order in which a program is run, increase the number of times certain lines are run, or terminate the program.

GO TO Statement

Syntax

<GO TO statement>



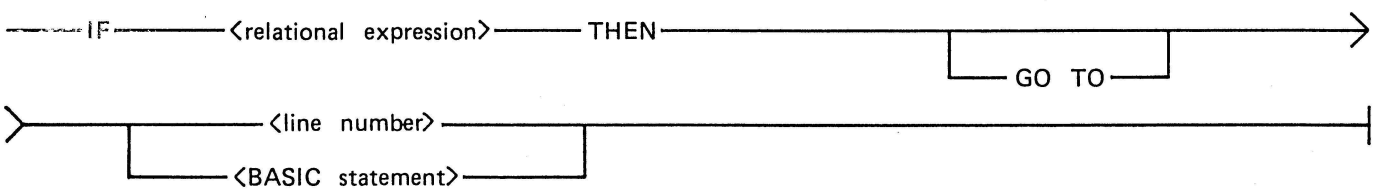
Semantics

The GO TO statement is an unconditional control statement. When there is a GO TO statement, the statement that matches the line number is executed next instead of the line following the GO TO statement.

IF Statement

Syntax

<IF statement>



Semantics

The IF statement is a conditional control statement. If the relational expression is true, the statement matching the line number is executed. If the relational expression is false, then the statement following the IF statement is executed.

For example,

```
30  IF A.LT.100 THEN GO TO 70
```

indicates that line 70 is executed after line 30 if A is less than 100. If A is greater than or equal to 100, then whatever line directly follows 30 is executed.

The program segment below sets

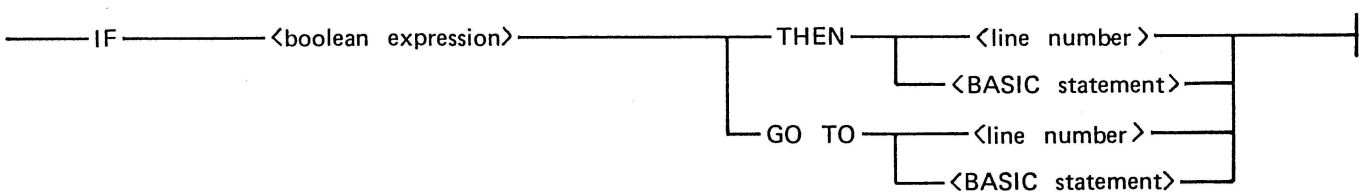
```
S=5*T+3 if T>9,
S=T**3-4 if T<3,
S=T**2+7*T if 3<=T<=9

20 IF T . GT. 9 THEN GO TO 70
30 IF T .LT. 3 THEN 90
40 LET S=T**2+7*T
50 PRINT "T=";T,  "S=";S
60 STOP
70 LET S=5*T+3
80 GO TO 50
90 LET S=T**3-4
100 GO TO 50
110 END
```

Statement 20 is a conditional GO TO statement. It tells the computer to skip the intervening statements and to pass control to statement 70 only if T is greater than 9. If T is not greater than 9, the next statement (line 30) is executed. This is another conditional GO TO statement that tells the computer to skip the intervening statements and to pass control to statement 90 if T is less than 3. If T is not less than 3, the next statement (line 40) is executed. Line 50 tells the computer to print T and S.

IF-THEN Statement

<IF-THEN statement>

Syntax



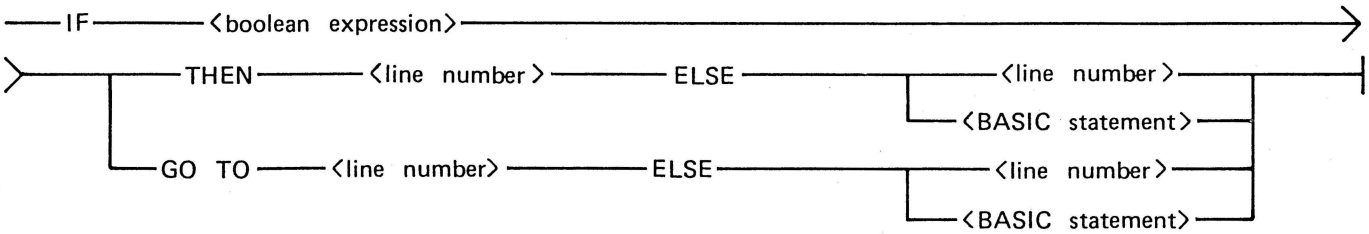
Semantics

The statement matching the line number is executed if, and only if, the boolean expression is true.

IF-THEN-ELSE Statement

<IF-THEN-ELSE statement>

Syntax



Semantics

If the boolean expression is true, the statement matching the line number before the ELSE is executed. If the boolean expression is false, then the statement matching the line number after the ELSE is executed. Notice that the statements following the THEN and the ELSE are not limited to GO TO statements.

IF-THEN and IF-THEN-ELSE statements are limited to single lines. Their component statements may not be the block statements: FOR, NEXT, DEF, FNEND, WHILE, WEND, DIM, DATA, FILES, or END statements. When IF-THEN and IF-THEN-ELSE statements are nested within one another, each ELSE is associated with the nearest preceding IF.

Example:

```
100  LET A=1234
200  LET B=4321
300  PRINT A,B
400  IF .NOT. 1=2 .AND. 4 .LT. 5 THEN SWAP A,B
500  GO TO 800
600  IF 1=2 .AND. 4 .LT. 5 THEN 1400 ELSE 1000
700  IF 1=4 .AND. 4 .GT. 5 GO TO 1400 ELSE 1300
800  PRINT A,B
900  GO TO 600
1000 PRINT "ON LINE 600, PART OF THE EXPRESSION IS FALSE,"
1100 PRINT "SO THE WHOLE THING IS FALSE"
1200 GO TO 700
1300 PRINT "ON LINE 700, THE EXPRESSION IS FALSE"
1400 END
```

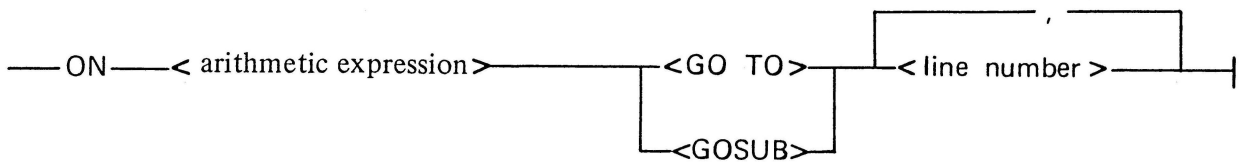
When the above program is executed, the following is output:

```
 4321          1234
ON LINE 600, PART OF THE EXPRESSION IS FALSE,
SO THE WHOLE THING IS FALSE
ON LINE 700, THE EXPRESSION IS FALSE
```

ON Statement

Syntax

<ON statement>



Semantics

The ON statement allows control to be transferred to one of several line numbers depending upon the value of an arithmetic expression.

The arithmetic expression in an ON statement is evaluated as an integer and is used to select a line number from the list following the GO TO or GOSUB. The line numbers in the list are enumerated from left to right, starting with 1. If the arithmetic expression is evaluated as one, then control is transferred to the first line number following the GO TO or GOSUB. If the arithmetic expression is evaluated as two, then control is transferred to the second line number, and so forth. If the arithmetic expression is larger than the number of line numbers, equal to zero, or equal to a negative number, than control passes to the next statement. User input may consist of more than one input list until the expected amount is received. Only the amount of data needed will be used. If an insufficient number of items has been supplied, the user will be prompted for more input. (When GOSUB follows an arithmetic expression, the line number of the ON statement is stored for the RETURN). Control is then transferred to the statement whose line number is selected.

Example:

```
100 INPUT A
110 ON A GO TO 140,160,180
120 PRINT "IGNORED TEST"
130 GO TO 100
140 PRINT "LINE 140"
150 GO TO 100
160 PRINT "LINE 160"
170 GO TO 100
180 PRINT "LINE 180"
190 GO TO 100
200 END
```

When the above program is executed with the following data,

```
?-1, 0, 1, 2, 3, 4
```

the output will be:

```
IGNORED TEST
IGNORED TEST
LINE 140
LINE 160
LINE 180
IGNORED TEST
```

PROGRAM LOOPS

It might be necessary to have the same portion of a program repeated several times, possibly with some slight modification each time that portion is executed. This repetitive process in a program is called a loop (or program loop). An example of this would be:

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

Statements 10 through 40 are executed in the normal manner; however, when step 50 is executed, program control returns to statement 10. This repetitive action continues until all of the data in the list is exhausted. At that time, the message DATA LIST IS EXHAUSTED is printed on the terminal and the program is terminated.

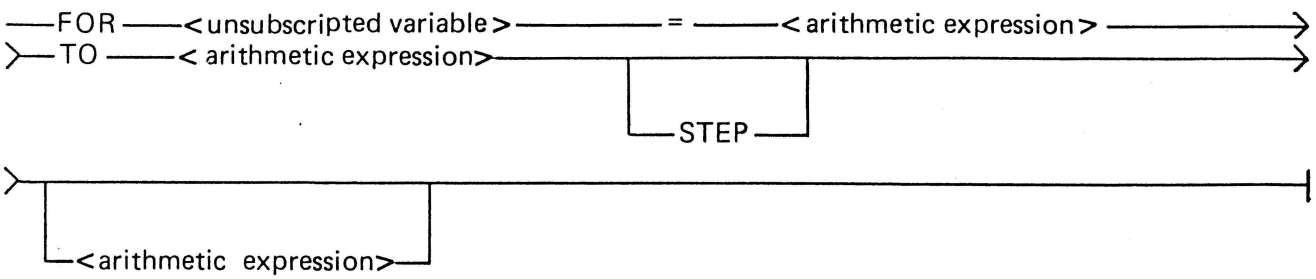
The example below illustrates the manner in which the square root of all numbers between 1 and 100 can be determined using a minimum amount of coding.

```
10 LET A=1
20 PRINT A, SQR(A)
30 LET A=A+1
40 IF A .LE. 100 THEN 20
50 PRINT "THAT IS ALL"
60 END
```

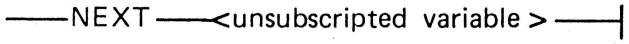
In this example, the variable A is selected as a counter and assigned an initial value of 1. The square root of A is calculated and printed. A is incremented by 1 and then tested to determine if it is greater than 100. If not, the program returns to statement 20 and the square root of A is printed. When A is greater than 100, the program passes control to statement 50 to print the ending message.

FOR and NEXT Statements

<FOR statement>



<NEXT statement>



Example:

```
100 FOR U=1 TO 10 STEP 2
200 NEXT U
```

Semantics

The NEXT statement must chronologically follow the FOR statement. When the FOR statement is executed, the variable U is set to 1, and the value of U is compared to 10. If U is greater than 10, the loop is completed, and the program passes control to the statement following the NEXT statement. If the variable is less than or equal to 10, any statements between the FOR and NEXT statements (called the body of the loop) are executed. When the NEXT statement is reached, the value of U is incremented by 2 and the program returns to the FOR statement. The body of the loop is executed for successively higher values of the counter variable until the U is greater than 10. When the STEP option is not included in the FOR statement, the step size is assumed by the computer to be +1.

Referring to the square root example, the program could be written in the following manner using the FOR and NEXT statements.

```
10 FOR A=1 TO 100
20 PRINT A, SQR(A)
30 NEXT A
40 PRINT "THAT IS ALL"
50 END
```

Notice that this program required only five statements as compared to six statements for the previous example. This is due to the fact that line 30 performs two tasks, namely, increments the counter variable by 1 and returns control to line 10.

In this particular example, the body of the loop contains only one statement. However, the body of a loop may be any number of statements in length, but it always ends with a NEXT statement. When the loop is completed, the statement immediately following the NEXT statement will be the first one to be executed. At this point the value of A will be 101, not 100 since A is always incremented at the NEXT statement before it is tested.

The expressions in the FOR statement are evaluated only once, at the start of the looping procedure. This means that the variables in the expressions can be changed, in the body of the loop, without affecting the test.

Example:

```
100 B=5
200 FOR A=1 TO B
300 PRINT A,B
400 B=3
500 NEXT A
600 END
```

When the above program is executed, the following is output:

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

The counter variable also can be changed in the body of the loop. Each time the end of the loop is reached, the latest value is incremented and tested. This could be accomplished by inserting a LET statement between the PRINT statement and the NEXT statement.

Example:

```
100 FOR A= 1 TO 10
200 PRINT A
300 LET A = 100
400 PRINT A
500 NEXT A
600 END
```

When the above program is executed, the following is output:

```
1
100
```

If the STEP value is negative, the counter variable is decremented by that value each time through the loop. The following examples show the values taken by the counter variable when various FOR statements are used.

FOR Statement	Values of Counter Variable
FOR A=-2 TO 1 STEP .5	-2, -1.5, -1, -.5, 0, .5, 1
FOR A=1 TO -2 STEP -1	1, 0, -1, -2
FOR A=1 TO -2	no output
FOR A=9 TO -2 STEP -2	9, 7, 5, 3, 1, -1

In the second to the last sample, the variable A is compared to -2. Since it is already greater, the loop is finished. The program passes control to the statement following the NEXT statement.

In the last example it is shown that the number following "TO" can be the same as the increment.

Nested loops must be contained; that is, they cannot cross, as illustrated in the following illegal skeleton example.

Illegal

```
10 FOR X=1 TO 5
20 FOR Y=1 TO 3
30 NEXT X
40 NEXT Y
```

Legal

```
10 FOR X=1 TO 5
20 FOR Y=1 TO 3
30 NEXT Y
40 NEXT X
```

It is illegal to use the same counter variable in two loops, one of which is nested in the other. A syntax error will be generated.

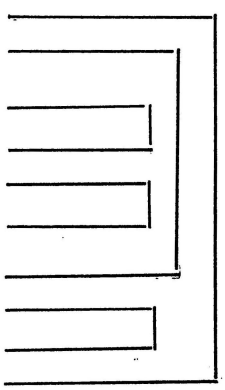
There may be a maximum nesting of 20 levels deep.

Illegal

```
10 FOR Z
20 FOR Z
30 NEXT Z
40 NEXT Z
```

Legal

```
10 FOR X
20 FOR Y
30 FOR Z
40 NEXT Z
50 FOR Z
60 NEXT Z
70 NEXT Y
80 FOR Z
90 NEXT Z
100 NEXT X
```



Control may be transferred out of the body of a FOR-NEXT loop without restriction. If, however, control is transferred into the body of a FOR-NEXT loop by any control statement other than a RETURN, the body of the loop will be executed infinitely (an infinite loop). This is because the increment is 0.

Example:

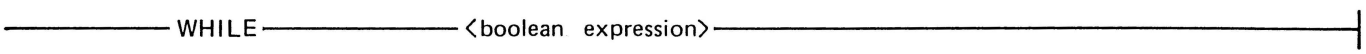
```
100 B=10
200 A=2
300 GO TO 500
400 FOR A=1 TO B
500 PRINT A,B, B*A
600 NEXT A
700 END
```

In the above example, line 300 causes a transfer into the middle of a loop. When executed, this line causes an infinite loop.

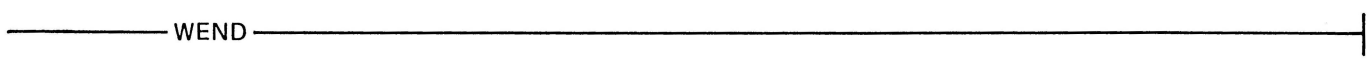
WHILE and WEND Statements

Syntax

<WHILE statement>



<WEND statement>



Semantics

The boolean expression is evaluated before each iteration, and the statements following the WHILE statement are executed repeatedly until the boolean expression becomes false. When that happens, control drops to the statement following the WEND statement. If the expression is initially false, the statements following the WHILE statement are not executed at all. Each statement in the WHILE loop must be on its own line. All block statements within the loop must have their beginning and ending within the loop. (The block statements are FOR, NEXT, DEF, FNEND, WHILE, WEND, DIM, DATA, FILES, and END).

The WEND statement indicates the end of the WHILE statement. An END statement is still needed to indicated the end of the entire program. If there is no WEND statement, the WHILE loop is not terminated and the syntax error 'WEND' STATEMENT EXPECTED is displayed.

Example:

```
100 WHILE C .LT. 5
200 PRINT "C IS";C
300 C=C+1
400 WEND
500 END
```

When the above program is executed, the following is output:

```
C IS 0
C IS 1
C IS 2
C IS 3
C IS 4
```

STOP Statement

Syntax

<STOP statement>

Semantics

The STOP statement is used to stop program execution and return control to the system. In contrast to the END statement, which must appear at the very end of the program and nowhere else, the STOP statement may appear anywhere in the program.

END Statement

Syntax

<END statement>

Semantics

Every program must have exactly one END statement, and it must be the statement with the highest line number. An END statement occurring anywhere other than the last line of the program will cause a warning to be generated and the statement will be ignored. Subsequent statements will be compiled. The absence of an END statement will also cause a warning to be generated. When the END statement is executed, the program terminates.

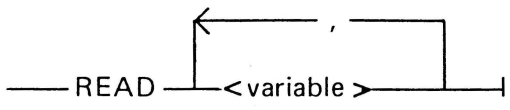
SECTION 4

I/O STATEMENTS

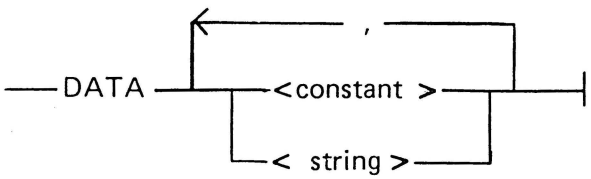
READ AND DATA STATEMENTS

Syntax

<READ statement>



<DATA statement>



Semantics

READ and DATA statements

The READ statement assigns a value from a DATA statement to a variable. These statements must be used together, never separately. All data that is to be assigned during the execution of the program is contained within the DATA statements. The computer takes all of the DATA statements in the program, in the order of their line numbers, and combines them into two data lists: a list of numeric constants, and a list of string constants. The READ statement causes each variable to take the next available constant or string value from the appropriate data list.

A string can be either quoted or unquoted. An unquoted string is considered to be any data list element which does not begin with a quote, number, +, -, ., or @. Leading blanks in unquoted strings are ignored, but any blanks within the string are included. An unquoted string may contain quotes.

Examples:

The statements

```
10 READ X, Y$, Z$
20 DATA 10,"20", ZEE
```

would result in the same action as

```
10 LET X=10
20 LET Y$="20"
30 LET Z$=ZEE
```

Likewise, the same results would occur if the program were written as:

```
10 READ X
20 READ Y$, Z$
30 DATA 10, "20"
40 DATA ZEE
```

The READ and DATA statements can be used to avoid excessive use of constants and LET statements. For example,

```
100 LET A=3
110 LET B=4
120 PRINT A,B,SQR(A**2+B**2)
130 LET A=2
140 LET B=2
150 PRINT A,B,SQR(A**2+B**2)
160 END
```

could be made shorter using READ and DATA statements.

```
100 READ A,B
110 PRINT A,B,SQR(A**2+B**2)
120 READ A,B
130 PRINT A,B,SQR(A**2+B**2)
140 DATA 3,4,2,2
150 END
```

When both numeric constants and strings appear in a DATA statement, their order in relation to each other does not matter. What is important is that the numeric data items appear in the order in which they are to be assigned to the numeric variables, and that the string data items appear in the order in which they are to be assigned to the string variables. This is due to the fact that BASIC stores string data and numeric data internally as two separate data lists.

Example:

```
10 READ A,B,X$,Y$
20 PRINT "A=";A,"B=";B
30 DATA FIRST THING, 9,17,"46"
40 PRINT "X$=";X$,"Y$=";Y$
50 END
```

When executed, this program will produce the following output:

```
A=9             B= 17
X$=FIRST THING Y$=46
```

If there are more values in the data lists than there are variables read by the program, the extra constants or strings are ignored. If there are more variables read in than values supplied, the message DATA LIST IS EXHAUSTED is printed on the remote terminal and the program is terminated.

Examples:

```
10 READ A, B, C
20 PRINT A; B; C
30 DATA 47, 37, 19, 5, 8
40 END
```

When executed, this program will produce the following output:

```
47  37  19
```

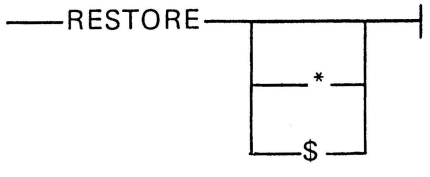
But if line 30 were changed to 30 DATA 47, 37, the output would be:

DATA LIST IS EXHAUSTED

RESTORE STATEMENT

Syntax

<RESTORE statement>



Semantics

The form "RESTORE" instructs the program to return to the start of the program data. "RESTORE *" instructs the program to return to the start of the numeric data list only. The form "RESTORE $" instructs the program to return to the start of the string data list only.

Example:

```
10 READ X,Y$,Z

20 DATA 10, YELLOW
```

would cause the program to be terminated with the message DATA LIST IS EXHAUSTED. However,

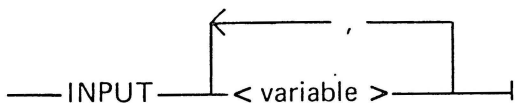
```
10 READ X,Y$
20 RESTORE *
30 READ Z
40 DATA 10,YELLOW
```

would cause the value of 10 to be assigned to X and Z, and the value YELLOW to be assigned to Y$.

INPUT STATEMENT

Syntax

<INPUT statement>



Semantics

The INPUT statement accepts input from a remote terminal during the execution of a program. The values input are assigned to variables.

The record size for remote input files has gone from 75 to 1920 characters per line. This allows more than one line of input to be read, that is, for long strings.

For example,

```
10 INPUT A,B
20 PRINT A,B,SQR(A**2+B**2)
30 INPUT A,B
40 PRINT A,B,SQR(A**2+B**2)
50 END
```

When the above program is executed, the computer stops at the first INPUT statement, displays a question mark at the terminal, and waits for the user to input the data for A and B. The user should separate each data item by a comma. When the data is received, the program will begin execution once more. When it stops at the second INPUT statement, the user must input data for A and B again. A possible result of this program would be the following:

```
?3,4
 3              4              5
?2,2
 2              2              2.8284271248
```

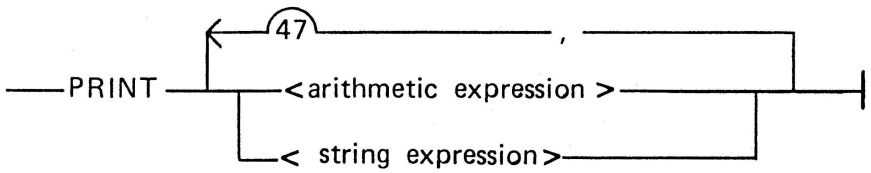
If the user does not supply enough data, the computer will continue to display a question mark. If too much data is supplied, the computer will ignore the excess data as in the READ statement.

An output statement can be used to print the results of an executed program.

PRINT STATEMENT

Syntax

<PRINT statement>



Semantics

The PRINT statement instructs the computer to print 1 to 48 elements.

Examples:

```
10 PRINT 2,5*11,D
20 PRINT A,B(3),SIN(C*D)+F
```

The following program takes an angle A in degrees, converts it to radians A1, and prints the angle in degrees and radians. It also prints its sine.

```
10 LET A=90
20 LET A1=(A*3.14159265)/180
30 PRINT A,A1,SIN(A1)
40 END
```

The result printed would be:

```
90             1.570796325  1
```

If the user wishes to print some message (text), the message need only be enclosed in quotation marks. Messages can either stand alone or be mixed with variables. In fact, any string expression can appear in a print list. For example:

```
100 A = 5
200 PRINT "A = "; A, "AND THAT IS THE TRUTH"
```

would cause the computer to print

```
A =  5          AND THAT IS THE TRUTH
```

The PRINT statement produces a line of output up to 75 characters long to the user's remote terminal (132 characters on the line printer if the program is compiled in batch mode).

Print Formats

Using the two available format types, vertical spacing and horizontal spacing (the TAB function), the programmer has almost complete form control of the output.

Zoned Format

For zoned output, each line is numbered 0 to 74, where positions 0, 15, 30, 45, and 60 mark the beginning of each print zone. When a comma is encountered in a PRINT statement, the computer moves to the next available print zone. If the current line has been filled, or if the input will run over the right margin, the computer will move to the next line before printing. Each number or text string requires a separate zone. If the number or text string is less than 15 characters long, the unused area of the zone is filled with blanks. For numerical values, printing normally starts in the second position of the zone since the first position is used for a minus sign, if applicable. In most cases, the computer will move to the next line after a PRINT statement is completed; however, if the PRINT statement ends with a comma, the next PRINT statement will begin in the next available print zone.

Example:

```
10 PRINT 1, -200, 9.31724673, 27E18, 765677
20 PRINT "ZONED","FORMAT"
30 PRINT 24,"INTEGER", 12.34, "REAL",
40 PRINT "NUMBER"
50 PRINT 1, 2, 3, 4, 5, 6, 7, 8
60 PRINT " THIS WILL RUN OVER THE SCREEN BOUNDARY, SO IT IS MOVED";
70 PRINT " DOWN ONE LINE"
80 END
```

Output produced by this program is:

```
1             -200          9.31724673     2.7E+19        765677
ZONED          FORMAT
24            INTEGER       12.34          REAL           NUMBER
1              2            3               4              5
6              7            8
THIS WILL RUN OVER THE SCREEN BOUNDARY, SO IT IS MOVED DOWN ONE LINE
```

Compressed Format

When two list items in a PRINT statement are separated by a semicolon, they will be printed with minimal spacing. For numeric data this means the number will be printed followed by one blank space. For string data the strings are output as is, without any trailing blanks.

Example:

```
10 PRINT 12; 13; -578; 12.3456789123; -7.6E39
20 PRINT "FIRST STRING"; "SECOND STRING"
30 A=SQR(1023.76)
40 A$ = "ABCD"
50 PRINT A$; -A; A; A$; A
60 END
```

Output produced by this program is:

```
 12  13 -578  12.345678912 -7.6E+39
FIRST STRINGSECOND STRING
ABCD-31.996249780  31.996249780 ABCD 31.996249780
```

If the PRINT statement ends with a semicolon, the next PRINT statement will begin after minimal spacing instead of on a new line.

Vertical Spacing

Vertical spacing can be achieved through use of the PRINT statement consisting of only the word PRINT. If a previous PRINT statement also signaled a new line because it did not end with a comma or semicolon, or because a line was filled by the last item printed, a line will be left blank in the output.

Example:

```
10 PRINT "SALES FOR WEEK ENDING MARCH 8"
20 PRINT
30 PRINT "JONES", "SMITH", "MILLER"
```

Would produce the following output:

```
SALES FOR WEEK ENDING MARCH 8

JONES          SMITH          MILLER
```

Horizontal Spacing (the TAB Function)

The TAB function is used to move the print mechanism to a print position determined by the element in the parentheses, and printing begins in the next print position.

Example:

```
PRINT A;TAB(Z);B;TAB(3*Z);C
```

This will result in the PRINT mechanism being moved to column Z after A has been printed, and to column 3*Z after B has been printed. Terminal print positions are numbered from 0 to 74. If the column number is greater than 74, it is taken modulo 74, with the exception that multiple of 74 will cause a TAB to position 74. If the argument is a real number, it is truncated to the nearest integer. If the

column designated by the TAB function has already been passed, the TAB function is ignored. If the information to be printed after a TAB will run over column 74, the TAB is ignored and the information is printed at the beginning of the next line. If the TAB is to a negative column number, the TAB will be to position zero. If the value of the TAB function is less than or equal to zero the warning "TAB OUT OF RANGE" is generated.

The use of commas and semicolons is not affected by the TAB function. When a variable is followed either by a comma or a semicolon, the print mechanism is moved to the right end of the field for that number (which includes separating spaces) before the next item in the PRINT statement is processed.

Examples:

```
10 PRINT "12345678901234567890123456789O"
20 PRINT TAB(7); "FIRST STRING";
30 PRINT TAB(22); 3; TAB(15); 44
40 PRINT TAB(5),"THE COMMA DID THIS"
50 END
```

Output produced by this program is:

```
12345678901234567890123456789O
      FIRST STRING    3  44
              THE COMMA DID THIS
```

```
100 PRINT TAB(78) "IN LIKE";
200 PRINT TAB(12.5) "A LION";
300 PRINT TAB(8) "*****BACKWARDS DIRECTION*****"
400 PRINT TAB(-8) "OUT LIKE A LAMB"
500 PRINT TAB(148) "MULTIPLE OF 74 AND RAN OVER"
600 END
```

Output produced by this program is:

```
BASIC WARNING***TAB OUT OF RANGE
   IN LIKE A LION*****BACKWARDS DIRECTION*****
OUT LIKE A LAMB

A MULTIPLE OF 74 AND RAN OVER
```

First, the warning is printed because of the TAB(-8). Then "IN LIKE" is printed in column 4 because modulo 78 is 4. Next, "A LION" is printed in column 12 because 12.5 is truncated. The backwards TAB to 8 is ignored and that line is printed right where it is. "OUT LIKE A LAMB" is printed in column 0 because it is a TAB to a negative number. Finally, "A MULTIPLE OF 74 AND RAN OVER" is printed at the beginning of the next line because it ran over column 74.

The TAB function can only be used in a PRINT statement.

String Variables in Print Statements

When a string variable appears in a PRINT statement, the string assigned to it is printed. If nothing has been assigned, it is treated as a null string and nothing is printed. String variables are used in the same way as text enclosed in quotation marks and may therefore appear in zoned or compressed format.

Example:

```
10 LET A$="10 SHOPPIN"
20 LET B$="G DAYS LEFT"
30 PRINT A$;B$;C$
40 END
```

and

```
10 DATA"10 ","SHOPPING DAYS ","LEFT"
20 READ A$, B$, C$
30 PRINT A$; B$; C$
40 END
```

would both be printed as

```
10 SHOPPING DAYS LEFT
```

String variables can be mixed with other items in the print list, as in the following examples:

```
110 PRINT A,B$(2);C,D$
120 PRINT "A=";A,B$;C$,D
```

Output to a Terminal

If a PRINT or WRITE statement is terminated with a comma or a semi-colon, the line of output will be held until another PRINT or WRITE statement is executed. If an INPUT statement is executed before another PRINT or WRITE, the line being held will be displayed before the program will accept the input from the terminal.

Example:

```
10 PRINT "INPUT A VALUE";
20 INPUT A
30 PRINT "THE VALUE WAS"; A
40 END
```

When executed the program will print:

```
INPUT A VALUE?
```

The question mark is printed when the system is ready to accept a value for the INPUT statement. The program will wait for the input with the terminal carriage positioned immediately after the question

mark. So if the user inputs the value "12", from the home position, the terminal output would look like:

```
INPUT A VALUE?
THE VALUE WAS 12
```

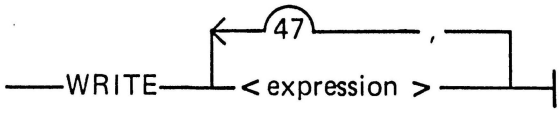
If line 10 was not terminated with a semicolon, the terminal output would appear as:

```
INPUT A VALUE
?
THE VALUE WAS 12
```

WRITE STATEMENT

<WRITE statement>

Syntax



Semantics

The WRITE statement functions similarly to the PRINT statement except that each data item is followed by a comma.

Example:

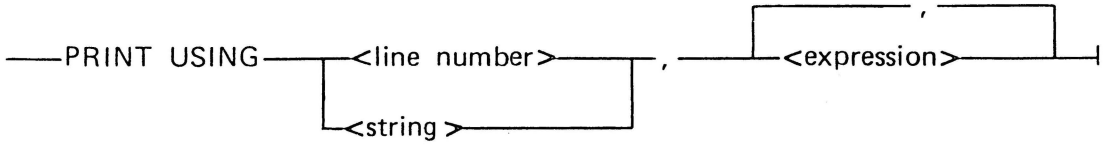
```
100 INPUT A
200 WRITE A
300 END
```

This example shows that you can follow input to a remote file with a write to that file.

PRINT USING STATEMENT

<PRINT USING statement>

Syntax



Semantics

The PRINT USING statement is used to print variables in a format. The line number directs the computer to an image statement. If there are not enough spaces in the image statement to print a numeric data item, an asterisk and the item are printed and then the program continues.

Examples:

```
100 B=0
200 C=3763
300 D=2.95
400 PRINT USING 500, B, C, D
500 :EMPLOYEE 35, $###.##, $###.##, AND $###.##
600 END
```

Execution of this program would produce the following output:

```
EMPLOYEE 35, $   .00, *$3763.00, AND $  2.95
```

```
100 X = .47077344233
200 PRINT USING "#########.#####", X
300 PRINT USING "########.#####", X
400 PRINT USING "#######.#####", X
500 END
```

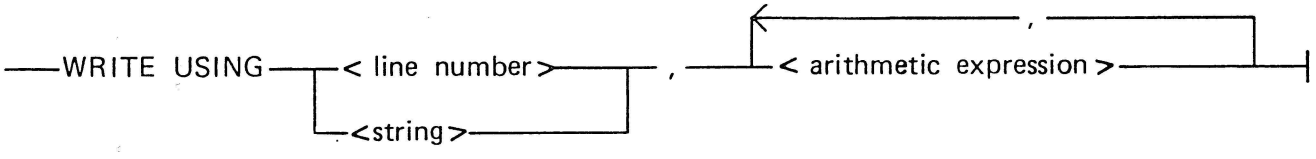
Execution of this program would produce the following output:

```
    .47077
   .47077
  .47077
```

WRITE USING STATEMENT

<WRITE USING statement>

Syntax



Semantics

The WRITE USING statement functions similarly to the PRINT USING statement except that each data item is followed by a comma. The same image statement may be used for both the PRINT USING and the WRITE USING statements.

Example:

```
100 A=5
200 B=8
300 WRITE USING " ### ### ####", A, 4, A*B
400 PRINT USING " ### ### ####", A, 4, A*B
500 END
```

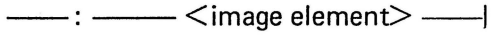
Execution of this program results in the following output:

```
5,   4,   40,
5   4   40
```

IMAGE STATEMENT

Syntax

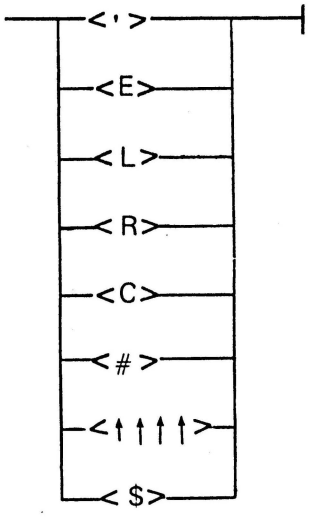
<IMAGE statement>



Semantics

The <line number> is the number of the statement in the program which is referenced in a PRINT USING statement.

<image element>



NOTES

' (Apostrophe). This is a one character field which is filled with the first character of an alphanumeric string.

E This is a continuation character which must always be preceded by an apostrophe (for example, 'EEE...E). This field specifies left justification of an alphanumeric string within the field if the string does not fill the field. If the string is longer than the width of the E-field, the E-field is expanded to the right to accommodate all of the string.

L This is a continuation character which must always be preceded by an apostrophe (for example, 'LLL...L). This field specifies left justification of an alphanumeric string within the field< > if the string does not fill the field. If the string is longer than the width of the L-field, the string will be truncated on the right.

R This is a continuation character which must always be preceded by an apostrophe (for example, 'RRR...R). This field specifies right justification of an alphanumeric string within the field if the string does not fill the field. If the string is longer than the width of the R-field the string will be truncated on the right.

C This is a continuation character which must always be preceded by an apostrophe (for example, 'CCC...C). This field specifies centering of an alphanumeric string within the field if the string does not fill the field. If the string is longer than the width of the C-field, the string is truncated on the right.

(Pound Sign) This is the replacement field for a numeric character. If the number is longer than the #-field, an asterisk is printed and the field is expanded to the right to accommodate the number.

↑↑↑↑ (Four Up Arrows) This field represents scientific notation for a decimal field.

$ (Dollar Sign) This is the replacement field for the dollar sign. When placed at the beginning of a decimal field or integer it will cause a dollar sign to be printed to the left of the numeric data in that position.

Any character other than the ones mentioned above will be treated as a literal field and will be printed exactly as it appears in the image.

When a PRINT USING or a WRITE USING statement is executed it begins with the first element in the associated image, whether or not a previous PRINT USING used only part of the image. If there are more data elements than there are image elements, the excess data elements are printed on a new line following the image format.

Example:

```
100:###### ### ##
200 WRITE USING 100, 1, 2, 3
300 PRINT USING 100, 4, 5
400 PRINT USING 100, 6, 7, 8, 9, 1, 2, 3, 4, 5
```

when executed, would look like this:

```
1,   2,   3,
4   5
6   7  8
9   1  2
3   4  5
```

Note that when the WRITE USING statement is used, each data item placed on the output line is followed immediately by a comma delimiter (unless the delimiter has been changed by a DELIMIT statement). If the WRITE USING is executed to an external EBCDIC file, a five-digit line number is placed at the beginning of each line of the file (the same as with WRITE statements without a USING clause).

When a replacement field is composed only of pound signs, it is referred to as an INTEGER FIELD. The following rules apply to integer fields:

a. An integer field may contain no more than 63 pound signs. If more than 63 pound signs appear adjacent to each other, the first 63 are treated as one integer field and the remainder as a second field.

b. Numbers are placed right justified in an integer field and truncated if they are not integers.

c. Any number equal to, or greater than 2 raised to the 39th power will be converted according to the image #.############↑↑↑↑.

d. The sign of the number is included in the field width.

e. If the number overflows the field, an asterisk will be printed and the field will be expanded to the right.

Example:

```
100: ###### ### ##
110 A$ =" ###### ### ## "
120 WRITE USING 100, 1, -43, 14
130 PRINT USING A$, -12345, 36.0, -1.76
140 PRINT USING 100, 7890, 444, .0
150 PRINT USING 100, 10E16, 12.3456, 0
999 END
```

When the above program is executed, the following output is printed at the user terminal:

```
1, -43, 14,
-12345  36 -1
7890 444  0
* 1.00000000000E+17  12  0
```

Note that the WRITE statement caused delimiters to be written after each data item on the first line of output.

A DECIMAL FIELD is a field of pound signs with a decimal point which may either precede, be imbedded in, or terminate the field. The following rules apply to decimal fields:

a. The total width of the field may be no more than 63 characters. If more than 63 characters appear in the field, the first 63 characters are treated as one field and the remainder as a second field.

b. If the fractional part of the number cannot be expressed exactly in the number of places specified to the right of the decimal point, the number will be rounded and then truncated to the proper number of places.

c. The number is placed right justified in the decimal field.

d. When the number overflows the field, an asterisk is printed and the field is expanded to the right.

Example:

```
100 LET A$ =".#### ###. ###.#### #.##"
110:.#### ###. ###.#### #.##
120 PRINT USING A$, .714698; -47.12; 728; -.00166
130 PRINT USING 110, .12345678, 29.8471, -22.76, 33.6698
999 END
```

When executed the example above will produce the following output:

```
.7147 -47. 728.0000 -.00
.1235  30. -22.7600 *33.67
```

A decimal field followed by four up arrows (↑↑↑↑) is referred to as an EXPONENTIAL FIELD. The following rules apply to exponential fields:

a. The pound signs preceding the decimal point represent the factor by which the exponent will be adjusted.

b. There must be at least one pound sign preceding the decimal point.

c. The leftmost pound sign reserves a position for the sign of the number (minus if negative, blank if positive).

d. If the fractional part of the number cannot be represented exactly in the number of places specified to the right of the decimal point, the number will be rounded and truncated to the proper number of places.

Example:

```
100 : #.####↑↑↑↑ ##.####↑↑↑↑##.↑↑↑↑
110 PRINT USING 100, 178.456, -36.9873, 123E17
120 PRINT USING 100, 41.123456, 4.768E12, 13
999 END
```

When executed, the above program will produce the following output:

```
.1785E+03    -3.6987E+01    1.E+19
.4112E+02     4.7680E+12    1.E+01
```

A field which consists of a string of dollar signs, or a string of dollar signs followed by either a decimal field or an integer field is referred to as a DOLLAR SIGN FIELD. The following rules apply to dollar sign fields:

a. Dollar signs may be placed only to the left of the decimal point.

b. Dollar signs are equivalent to pound signs except that a dollar sign will appear to the left of any numeric data in the field.

c. Only one dollar sign will be printed.

d. When there is more than one dollar sign in the image field the actual dollar sign will be floated as far to the right as possible.

Example:

```
100 : $###.## $$$$$. $$.##
110 PRINT USING 100, 1.36, 1.36, 1.36
120 PRINT USING 100, 11.36, 11.36, 11.36
130 PRINT USING 100, 111.36, 111.36, 111.36
140 PRINT USING 100, 1111.36, 1111.36, 1111.36
999 END
```

When executed, this program will produce the following output:

```
$  1.36    $1. $1.36
$ 11.36   $11. *$11.36
$111.36  $111. *$111.36
*$1111.36 $1111. *$1111.36
```

A field which begins with an apostrophe is referred to as an ALPHANUMERIC FIELD. The following rules apply to alphanumeric fields:

a. Any one of the continuation characters C, E, L or R may be used to continue a field introduced by an apostrophe.

b. The continuation characters C, E, L and R may not be mixed in a single field.

Example:

```
100:'CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
110:'EEEEEEEEEE 'LLLLLLLLL 'RRRRRRRRR
120 A$ = "A12345"
130 B$ = "B23456789B"
140 C$ = "C23456789O123456789C"
150 PRINT USING 100, "TEST OF ALPHANUMERICS"
160 PRINT USING 110, A$, A$, A$
170 PRINT USING 110, B$, B$, B$
180 PRINT USING 110, C$, C$, C$
999 END
```

When executed, this program will produce the following output:

```
          TEST OF ALPHANUMERICS
A12345      A12345          A12345
B23456789B  B23456789B   B23456789B
C23456789O123456789C C2345678901 C2345678901
```

A field which is composed of characters which do not form one of the above mentioned image fields is referred to as a literal field. A literal field will appear on the print line exactly as it appears in the image.

Example:

```
100:AN EXAMPLE OF A LITERAL FIELD AT LINE: ####
110 PRINT USING 100, 110
999 END
```

When executed this program will produce the following output:

```
AN EXAMPLE OF A LITERAL FIELD AT LINE:  110
```

Observe that the letters C, E, L, and R are used without ambiguity since they are not preceded by an apostrophe.

Too many variables and/or image statements in a program can generate an error and abort the compilation of a program. The array in which variables and image statements is stored can hold 1023 words.

When the USING feature is used with MAT PRINT or MAT WRITE, each row of the matrix begins on a new line with the first field in the image.

SECTION 5

FUNCTIONS AND SUBPROGRAMS

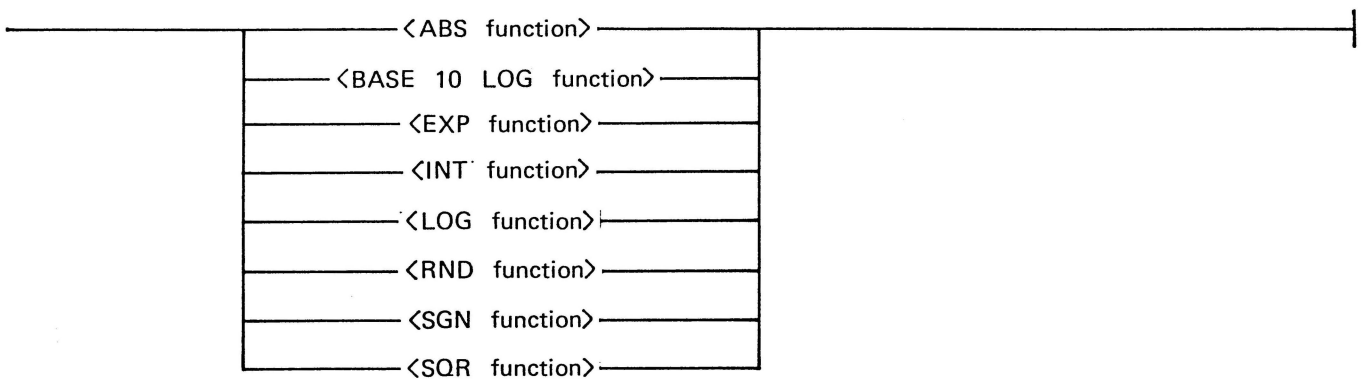
GENERAL DESCRIPTION

Functions are short programs which perform specific tasks. These programs are inherent to the compiler and can be helpful to a programmer. The following arithmetic functions are available in BASIC.

ARITHMETIC FUNCTIONS

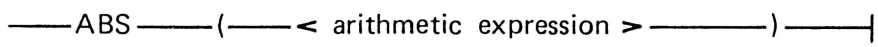
Syntax

<arithmetic function>



ABS FUNCTION

<ABS function>



The ABS (absolute value) function returns the absolute value of an expression. ABS(X) is equal to X if X is greater than or equal to zero. ABS(X) is equal to -X if X is less than zero.

Example:

```
ABS(3) = 3                    ABS(-3) = 3
```

BASE 10 LOG FUNCTION

<BASE 10 LOG function>

Syntax

Semantics

The function LOG10 returns the base 10 logarithm of the arithmetic expression.

Example:

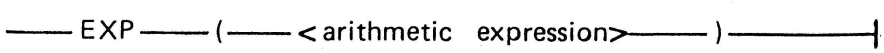
```
100 C=8
200 K=LOG10(C)
300 PRINT C;K
400 END
```

When the above program is executed, the following is output:

```
8   .90308998699
```

EXP FUNCTION

<EXP function>



The EXP (exponential function) finds the exponential of an expression by raising the mathematical constant e to the power of X. (EXP(X) = e**X where e = 2.71828...).

Example:

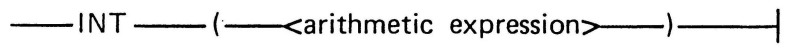
```
100     A = EXP(19)
200     PRINT A
300     END
```

would result in the following output on the terminal

```
178482300.96
```

INT FUNCTION

<INT function>



The INT (integer) function calculates the greatest integer less than or equal to the argument. This is commonly known as the STEP function.

Examples:

```
INT(7.5) = 7                    INT(-7.5) = -8
```

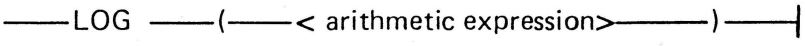
One use of the INT function is to round a number to any number of decimal places. In general INT (X * 10**D + .5)/ 10**D will round X to D decimal places.

Examples:

```
INT (3. 678 + .5)=4
(3.678 rounded to 0 decimal places is 4).

INT (3.678 * 100 + .5)/100 = 3.68
(3.678 rounded to 2 decimal places is 3.68).

INT (-5.678 * 100 + .5)/100 = -5.68
(-5.678 rounded to 2 decimal places is -5.68).
```

LOG FUNCTION

<LOG function>



The LOG (logarithm) function finds the natural logarithm of an expression.

Example:

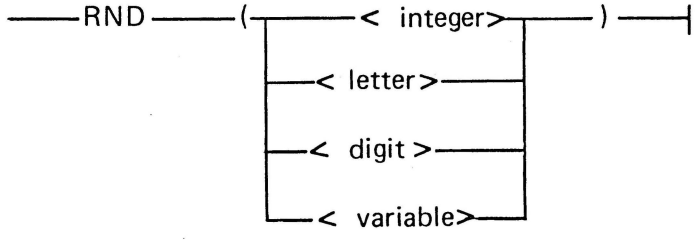
```
100 X = LOG(52365)
200 Z = LOG(5*10**7)
300 PRINT X,Z
400 END
```

would output the following:

```
10.865993700     17.727533563
```

RND FUNCTION

<RND function>



The RND (random) function generates random numbers between 0 and 1.

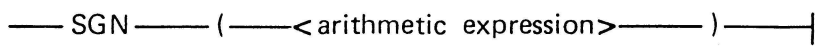
Example:

```
100 FOR I = 1 TO 100
200 PRINT RND(Z)
300 NEXT I
```

Will produce the same sequence of 100 random numbers every time it is executed. The element within the parentheses of a RND function is called a dummy argument. To get a different set of random numbers with each RUN, use a negative number as the dummy argument.

SGN FUNCTION

<SGN function>



The SGN (sign) function returns the sign of the expression or constant in the parentheses. If the value is 0 it returns a 0. If the value is positive, it returns a 1. If the value is negative, it returns a -1.

Example:

Function	Value returned
SGN(0)	0
SGN(-3.69)	-1
SGN(453)	1
SGN(-1.@-46)	-1

SQR FUNCTION

<SQR function>

The SQR (square root) function returns the square root of a constant.

Examples:

```
SQR(4) = 2
SQR(27) = 5.1961524227
SQR(5*5) = 5
```

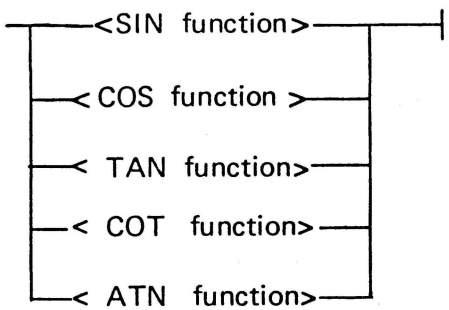
If the arithmetic expression in parentheses is a negative number, then the error message, INVALID SQRT ARGUMENT, is returned.

TRIGONOMETRIC FUNCTIONS

The following trigonometric functions are available in BASIC.

Syntax

<trigonometric function>



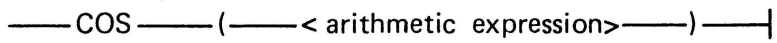
SIN FUNCTION

<SIN function>

The SIN (sine) function returns the sine of the <arithmetic expression>. The <arithmetic expression> is an angle expressed in radians.

COS FUNCTION

<COS function>



The COS (cosine) function returns the cosine of the angle of the <arithmetic expression>. The <arithmetic expression> is an angle expressed in radians.

TAN FUNCTION

<TAN function>

The TAN (tangent function returns the tangent of the angle of the <arithmetic expression>. The <arithmetic expression> is an angle expressed in radians.

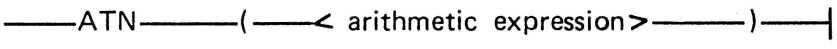
COT FUNCTION

<COT function>

The COT (cotangent) function returns the cotangent of the angle of the <arithmetic expression>. The <arithmetic expression> is an angle expressed in radians.

ATN FUNCTION

<ATN function>

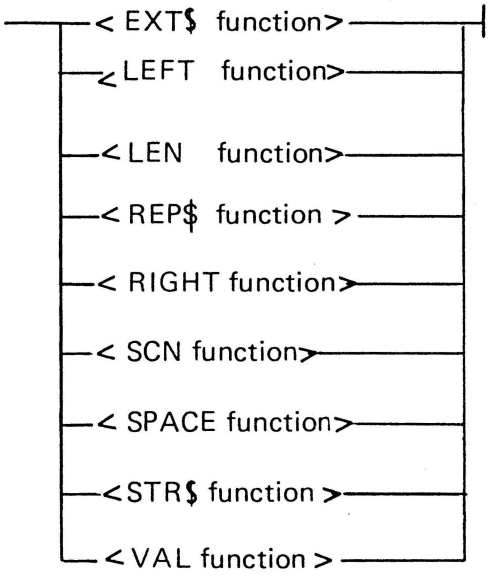


The ATN (arctangent) function returns the arctangent of the angle of the <arithmetic expression>. It is an angle expressed in radians.

STRING FUNCTIONS

Syntax

<STRING functions>



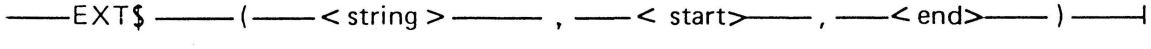
Semantics

BASIC provides the following string functions.

EXT$ FUNCTION

Syntax

<EXT$ function>



Semantics

The EXT$ function extracts a designated segment of a character string. The <start> is the number of the character position in the string at which extraction is to start and <end> is the number of the character position in the string at which the extraction is to end. <start> and <end> may be numeric constants, arithmetic variables or arithmetic expressions. The string may be a string constant, string variable, or string expression.

Example:

```
100  A$ = "1234567"
110  E$ = EXT$(A$,3,6)
120  F$ = EXT$("1234567", 3, 3+3)
130  G$ = EXT$ ("1234" + "567", 3, 6)
140  PRINT E$, F$, G$
150  END
```

When the above program is executed, the following is output:

```
3456          3456          3456
```

If N characters are to be extracted from a string, <end> should be equal to <start> + N-1.

LEFT FUNCTION

Syntax

<LEFT function>

Semantics

The LEFT function returns the specified number of leftmost characters from a string expression. <part of string> is the number of characters to be returned out of the <string>. If <part of string> is greater than <string>, then the length of <string> is returned. The <string> may be a string constant, string variable, or string expression.

Example:

```
100  A$ = "1234567"
110  E$ = LEFT(A$,4)
120  PRINT E$
130  END
```

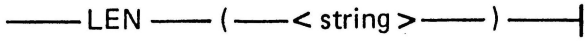
When the above program is executed, the following is output:

```
1234
```

LEN FUNCTION

Syntax

<LEN function>



Semantics

The LEN function returns the number of characters in a specified string. The <string> may be a string constant, string variable, or string expression.

Example:

```
100  A$ = "ABC"
110  L = LEN(A$)
120  PRINT L
130  END
```

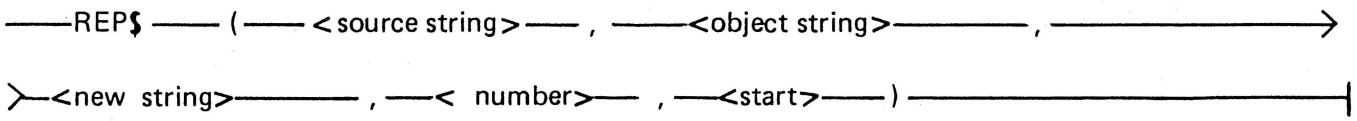
When the above program is executed, the following is output:

```
3
```

REP$ FUNCTION

Syntax

<REP$ function>



Semantics

The REP$ function returns a string which is formed by replacing specified occurrences of a segment of a string with new string segments. The <source string> is the string in which the replacement is to occur. The <object string> is the string which is to be replaced by the <new string>. The <number> specifies how many occurrences of the <object string> are to be replaced and the <start> specifies the number of the character position at which the search for the <object string> is to begin. The strings may be string constants, string variables, or string expressions. <number> and <start> may be numeric constants, arithmetic variables, or arithmetic expressions.

If the <number> is less than zero, then all occurrences of the <object string> starting with the <start> character in the <source string> will be replaced.

If the <number> is equal to zero, no replacements are made and the value returned by REP$ is a string equivalent to the <source string>.

When the <number> is greater than zero, then beginning with the <start> of the <source string>, the <number> of the <object string> is replaced by the <new string>.

Example:

```
100 A$ = "ABBCBBDBBEBB"
110 B$ = "BB"
120 D$ = REP$(A$,B$,"-",1,4)
130 E$ = REP$(A$,B$,"X",-1,7)
140 F$ = REP$(A$,B$,"#",2,1)
150 PRINT D$; E$; F$
999 END
```

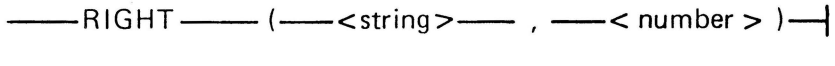
When the above program is executed, the following is output:

```
ABBC    ABBCBBDXEX    A#C#DBBEBB
```

RIGHT FUNCTION

Syntax

<RIGHT function>



Semantics

The RIGHT function returns the specified number of rightmost characters from a string expression. The <number> is the number of rightmost characters to be returned from the <string>. If the <number> is greater than the length of the <string>, the <string> is returned. The <string> may be a string constant, string variable, or string expression.

Example:

```
100  A$="1234567"
110  E$=RIGHT(A$,4)
120  PRINT E$
130  END
```

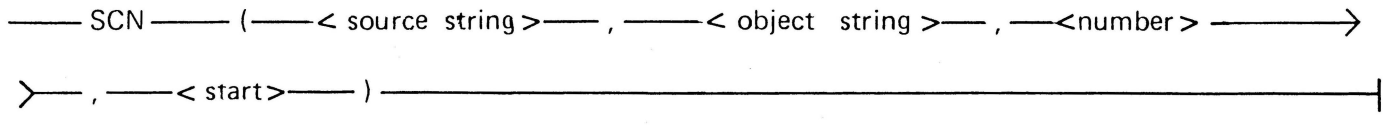
When the above program is executed, the following is output:

```
    4567
```

SCN FUNCTION

Syntax

<SCN function>



Semantics

The SCN function returns the number of the character position at which a specified occurrence of a string segment occurs within another string. The function returns a value of zero if the specified occurrence of the string segment does not exist. The <source string> is the string in which the string segment occurs and the <object string> is the segment which is desired. The <number> is the number of occurrences which are desired. The <start> is the number of the character position in the <source string> at which the scan is to begin. The <source string> and <object string> may be string constants, string variables, or string expressions. <start> and <number> may be numeric constants, arithmetic variables, or expressions.

Example:

```
100 S$ = "ABBCBBDBBEBB"
110 P$ = "BB"
120 A = SCN(S$,P$,1,1)
130 B = SCN(S$,P$,1,4)
140 C = SCN(S$,P$,2,7)
150 D = SCN(S$,P$,5,1)
160 PRINT A; B; C; D
999 END
```

When the above program is executed, the following is output:

```
2  5  11  0
```

SPACE FUNCTION

Syntax

<SPACE function>

```
-- SPACE -- ( --<number>-- ) --!
```

Semantics

The SPACE function returns a specified number of blank spaces. The <number> is the specified number of characters. The SPACE function usually follows a PRINT; it cannot start a line.

Example:

```
100 FOR I = 1 TO 10
200  A$ = SPACE (I)
300  B$ = "I" + A$ + "TALK" + A$ + "BASIC"
400  PRINT USING 500, B$
500  :'CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
600  NEXT I
700  END
```

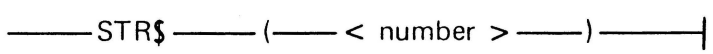
When the above program is executed, the following is output:

```
          I TALK BASIC
         I  TALK  BASIC
        I   TALK   BASIC
       I    TALK    BASIC
      I     TALK     BASIC
     I      TALK      BASIC
    I       TALK       BASIC
   I        TALK        BASIC
  I         TALK         BASIC
 I          TALK          BASIC
```

STR$ FUNCTION

Syntax

<STR$ function>



Semantics

The STR$ function returns a string corresponding to a specified value. The <number> is the value which is to be returned as a string. It may be a constant, arithmetic variable, or arithmetic expression. The string returned by STR$ is the same as if the <number> had been printed by a PRINT statement.

Example:

```
100 S$ = STR$(123)
110 PRINT S$
999 END
```

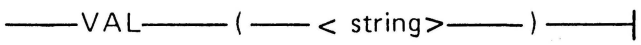
When the above program is executed, the following is output:

```
123
```

VAL FUNCTION

Syntax

<VAL function>



Semantics

The VAL function returns a numeric constant corresponding to a specified string. The <string> is the specified string for which VAL returns a corresponding numeric constant. The <string> may be a string constant, string variable, or a string expression, the characters of which form a valid number.

Example:

```
100 A$ = "123."
110 B$ = "375E+21"
120 N = VAL(A$ + B$)
130 PRINT N
999 END
```

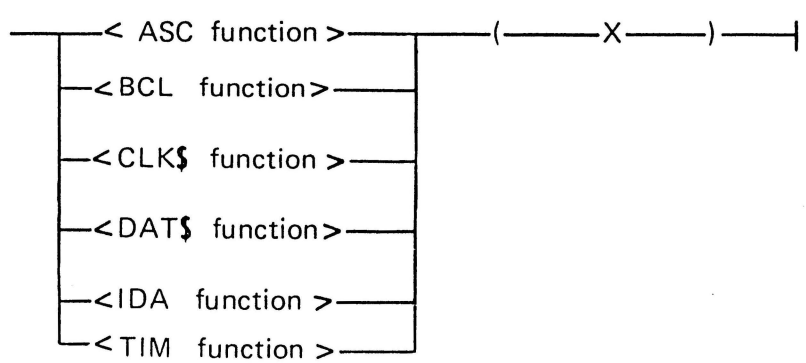
The "+" in line 110 is concatenation and not addition. When the above program is executed, the following is output:

```
1.23375E+23.
```

SYSTEM FUNCTIONS

Syntax

<system functions>



Semantics

The following system functions are available in BASIC.

ASC FUNCTION

<ASC function>

Semantics

The ASC function returns the numeric value of a specified EBCDIC character. Where X is the specified EBCDIC character. If the character cannot be printed, an abbreviation for the character may be used. The abbreviations which the ASC function will accept are NUL, SOH, STX, ETX, HT, DEL, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, BS, CAN, EM, FS, GS, RS, US, LF, ETB, ESC, ENQ, ACK, BEL, SYN, EOT, NAK, SUB, and SP.

Example:

```
100 PRINT "1 =";ASC(1)
110 PRINT "A ="; ASC(A)
120 PRINT "LINE FEED =";ASC(LF)
130 PRINT "NUL =";ASC(NUL)
999 END
```

When the above program is executed, the following is output:

```
1 = 241
A = 193
LINE FEED = 37
NUL = 0
```

BCL FUNCTION

<BCL function>

Semantics

The BCL function returns the time of day, based on a 24-hour clock, in hours and decimal fractions of the hour.

Example:

```
100 A = BCL
999 END
```

At 3:45 p.m., A will contain the value 15.75.

CLK$ FUNCTION

<CLK$ function>

Semantics

The CLK$ function returns the time of day as a string five characters long in the form:

```
HH:MM
```

Where HH is hours and MM is minutes. The time is based on a 24-hour clock.

Example:

```
100 A$ = CLK$
999 END
```

At 4:30 p.m., A$ will contain 16:30.

DAT$ FUNCTION

<DAT$ function>

Semantics

The DAT$ function returns the current date as a string eight characters long in the form:

```
MM/DD/YY
```

Where MM is the current month, DD is the current day, and YY is the current year.

Example:

```
100 PRINT DAT$
999 END
```

On August 11, 1982, the above program would output 08/11/82.

IDA FUNCTION

<IDA function>

Semantics

The IDA function returns the current data as a six-digit integer of the form:

```
YYMMDD
```

Where YY is two digits representing the year, MM is two digits representing the month, and DD is two digits representing the day.

Example:

```
100 PRINT IDA
999 END
```

On March 8, 1982, the above program would output 820308.

TIM FUNCTION

<TIM function>

Semantics

The TIM function returns the elapsed processor time since the job began. This time is expressed in seconds and decimal fractions of seconds.

Example:

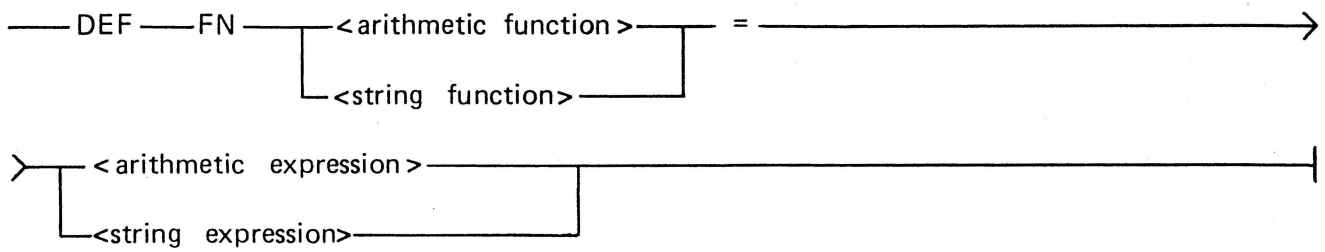
```
100 PRINT TM
999 END
```

If the processor time elapsed had been 1/9 second, the output would be .11111111.

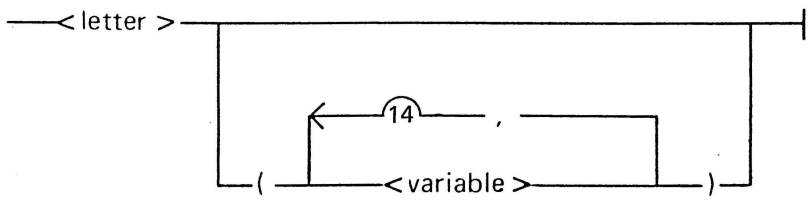
STATEMENT FUNCTIONS AND SUBPROGRAMS

The user can avoid repeated sections of code within a program by separating the code as a function or subprogram. A program then passes control to a function or subprogram when the corresponding section of code is to be executed.

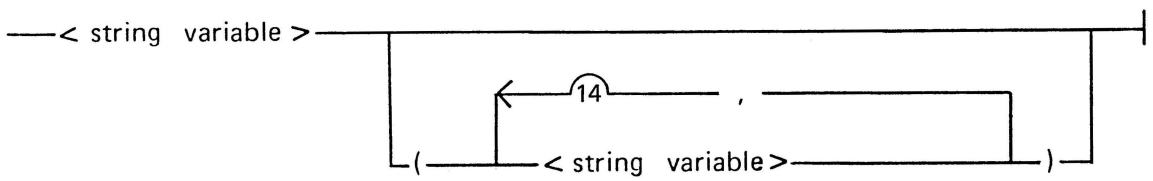
<DEF statement>



<arithmetic function>



<string function>



In addition to inherent functions, the user is able to define other functions which will be used repeatedly. This avoids the need to repeat the particular function each time it is required by the program. The name of the defined function must consist of three letters. The first two of which must be the letters FN, and the third is a user supplied letter. The <variable> can be arithmetic or string. The type (arithmetic or string) must be the same on both sides of the equal sign. A string function returns a string as its value and the <variable> must be written as a string variable ($ <variable>). The user is provided with the ability to define as many as 52 unique functions (26 (A-Z) which return numeric values and another 26 string functions).

Statement functions are defined by the DEF statement. The DEF statement contains a function's name for either a multiple or a single statement function. A single statement function contains the entire function in one DEF statement.

Previously, user-defined functions were not allowed to contain calls on themselves. That is, a DEF statement could not have the same function on the right and the left of the equal sign. If the same function appears on the right and on the left of the equal sign, that is called recursion and it used to result in the syntax error, RECURSIVE FUNCTION CALL NOT PERMITTED.

Functions may now be defined recursively. A recursive function is referenced in the same way as any other function except that, inside the definition of the function itself, parentheses must be included even if the function takes no variables.

Examples:

```
100 DEF FNF
200   X = X - 1
300   IF X = -1 THEN FNF = 1 ELSE FNF = (X + 1) * FNF()
400 FNEND
500
600 X = 5
700 PRINT FNF
800 END
```

NOTE

On line 300, parentheses are included even though FNF takes no variables. The parentheses are not needed on line 700 because the line is outside the function definition (after the FNEND statement).

```
100 DEF FNA
200     FNA = X
300     IF X <=0 THEN 700
400     X = X-1
500     FNA = FNA
600     PRINT "*";
700 FNEND
800 X = 5
900 D = FNA
```

The rightmost FNA on line 500 refers to the local function value. An * is printed when the segment is run. If line 500 is replaced by 500 FNA = FNA (), the rightmost FNA is a recursive call so ***** will be printed. In both cases, the left-most FNA on line 500 refers to the local function value.

Single Line Statement Function

The variable in parentheses on the left side of the equal sign is called a dummy variable. It can be a string variable or an arithmetic variable. If more than one variable is used, they must be separated by commas. The type (numeric or string) of an argument in a function must be the same type as its corresponding dummy variable in the DEF statement. They can be used on either side of the equal sign. They are used to reserve the place for the variable which will be used when the function is called. Wherever the dummy variable appears in the expression, the value of the argument used in the call for the user-defined function is used in evaluating the expression. The expression on the right side of the equal sign may be any formula that can fit on the rest of the statement line. It may include any combination of other functions, including the ones defined by other previously or subsequently defined DEF statements. A statement containing a function call may appear before and/or after the DEF statement for that function.

Example:

```
25 DEF FNA(X) = COS(X*3.14/180)
```

defines FNA as the cosine of X degrees.

It is illegal to have a user-defined function call a second user-defined and then have the second user-defined function call the first one. This results in a stack overflow error.

Example:

```
100 DEF FNA = FNB + 40
200 DEF FNB = X - 10 * FNA
```

the above code is illegal.

The variable is used only to show how the value of the argument is used in the expression. Thus,

```
75 DEF FNA (Z) = COS(Z*3.14/180)
```

defines exactly the same function as the first example. Furthermore, the use of a variable as a dummy does not in any way affect its use elsewhere in the program. Its value does not change when the function is used. For example, consider the following three statements:

```
10 LET X=Y=30
20 LET S1=FNA(40)
30 LET S2=X+Y
```

If FNA is defined by:

```
40 DEF FNA(X)=X+Y**2
```

the value assigned to S1 will be 940 (40+30**2). However, if the function is defined by:

```
40 DEF FNA(Y)=X+Y**2
```

The value assigned to S1 will be 1630 (30+40**2). In either case, 60 will be assigned to S2 since the use of X or Y as a dummy variable does not affect their value.

Sometimes it is convenient to have more than one argument in a function, or no arguments at all. Examples of this type of function are:

```
10 DEF FNA=SQR(X*X+Y*Y)
20 DEF FNC(X,Y)=SQR(X*X+Y+Y)
```

Both of these examples perform the same calculation. The difference is the manner in which values are supplied. For instance, in the first example, both X and Y are variables which must have values assigned to them, while in the second example both X and Y are dummies and the value of their respective arguments is used in the expression. Determining which of these functions is best would depend on the circumstances in which it would be used. If the calculation is needed for several different pairs of variables, the second form should be used. However, if the calculation is done for the same pair in several different places, the first form is better.

Note that the DEF statement is not needed unless the defined function must appear at two or more locations in the program. Therefore, the following program which is used to print a table of values of the sine function in degrees:

```
10 DEF FNF(Z)=SIN(Z*P)
20 LET P=3.14159/180
30 FOR X=0 TO 90
40 PRINT X, FNF(X)
50 NEXT X
60 END
```

can be more efficiently written as:

```
20 LET P=3.14159/180
30 FOR X=0 TO 90
40 PRINT X, SIN(P*X)
50 NEXT X
60 END
```

A user-defined string function may be used wherever a string variable can be used.

Example:

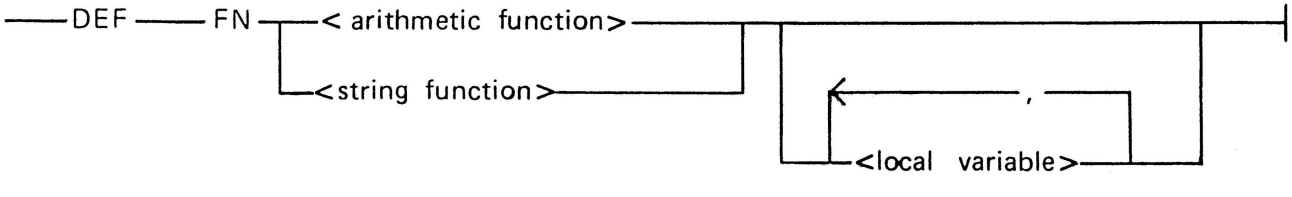
```
10 DEF FNC$(X$,Y$,S) = X$+SPACE(S)+Y$
20 LET A$ = "A VERY"
30 LET B$ = "GOOD BEGINNING"
40 LET F$ = FNC$(A$,B$,10)
50 END
```

After the execution of line 40, F$ will contain the string

"A VERY GOOD BEGINNING".

Multiple Line Statement Function

If a function cannot be contained on one line, a multiple statement function is used. The first statement of a multiple line function has the format:



The list of local variables can be used only by the function. Once again, the variable may be arithmetic or string.

The first statement of the function is followed by those statements which define the function.

FNEND STATEMENT

Syntax

<FNEND statement>

Semantics

This statement indicates the end of the multiple statement function. If there is no FNEND statement, the function is not terminated.

DEF statements may not be nested. Furthermore, functions defined by DEF statements can be executed only by invoking the function with a function call. Specifically, transfer of program control into the range of a function subprogram by any statement other than a function call is not permitted. Also, transfer of program control from the range of a function subprogram is not permitted except by normal return from the subprogram. For example, the execution of a FNEND statement.

For a function to return a value, an assignment to the function name (via LET, READ or INPUT) should appear within the statements which define the function. Inside the function, the function name can be used as a local variable. Upon execution of the FNEND statement, the current value of the function name variable is returned.

Example:

```
10  FOR I = 0 TO 5
20     X = FNF (I)
30     PRINT I;"FACTORIAL= ";X
40  NEXT I
50  DEF FNF (N) I, X
60  IF N <> 0 THEN 90
70  FNF = 1
80  GOTO 140
90  X = 1
100 FOR I = 1 TO N
110    X = X * I
120 NEXT I
130 FNF = X
140 FNEND
150 END
```

The example will produce the output:

```
0 FACTORIAL=  1
1 FACTORIAL=  1
2 FACTORIAL=  2
3 FACTORIAL=  6
4 FACTORIAL=  24
5 FACTORIAL=  120
```

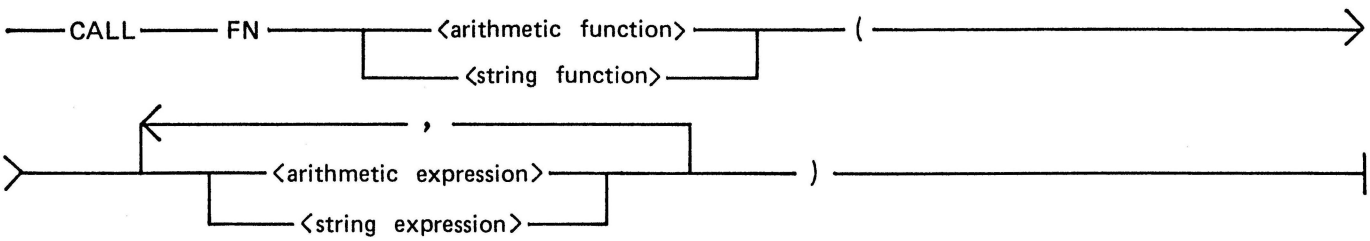
Line 50 defines the function F to have one argument N and two local variables I and X. By defining I and X to be local, the use of these variables in the function will have no effect on the global variables I and X used in lines 10 through 40. Lines 70 and 130 assign values to the function F.

For a multiple line function which is defined as a string function, (as an example, 100 DEF FNA$(B$,X,C$)), the function value assignment must specify the function name followed by "$". For example, 200 FNA$ = "THIS IS THE FUNCTION VALUE".

CALL STATEMENT

<CALL statement>

Syntax



Semantics

The CALL statement causes the defined arithmetic or string function to be invoked as a subroutine. The value it returns is then discarded. (When functions are invoked, rather than subroutines, a value is returned).

Example:

```
100 DEF FNC$(X$,Y$,S) = X$+SPACE(S)+Y$
200 A$="QUALITY"
300 B$="COUNTS"
400 PRINT FNC$(A$,B$,5)
500 END
```

When the above program is executed, the following is output:

```
QUALITY     COUNTS
```

But if the lines 350 FNEND and 375 CALL FNC$(A$,B$,5) were added to the program, there would be no output.

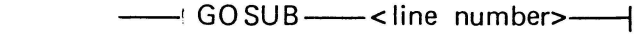
SUBPROGRAMS

A subprogram is a section of the program which is used in a way similar to the way a called function is used. When a called function appears, the computer performs the calculations needed to evaluate the function, and then continues to execute the statement. Similarly, the GOSUB statement tells the computer to go execute the subprogram. The RETURN statement then tells the computer that the subprogram is over and it should go back to the statement following the GOSUB statement.

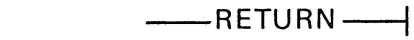
GOSUB AND RETURN STATEMENTS

Syntax

<GOSUB statement>



<RETURN statement>



Semantics

Example:

```
110  GOSUB  400
120  ----------
 .   ----------
 .   ----------
 .   ----------

400  ----------
 .   ----------
440  RETURN
```

would cause the computer to branch to line 400 when it had reached line 110. After executing lines 400 through 430, it would be directed by line 440 to return to line 120.

In effect, the GOSUB acts like a GO TO except that the computer stacks the GOSUBs that have been executed. When a RETURN is encountered, the program returns to the statement that immediately follows the GOSUB last executed. This makes the concept of subprograms very powerful. For instance, subprograms may be nested; that is, subprograms may

include GOSUBs calling other subprograms. A nested subprogram may call a subprogram B which has previously called A, or a subprogram may call itself. This is known as recursion. Furthermore, a GOSUB may cause a transfer to any line in the program, and the program will act just as if a GO TO had been executed. Then, when it encounters a RETURN at a time when the GOSUB is at the top of the stack, it will transfer to the statement following the GOSUB.

Example:

```
100  INPUT X, Y
110  GOSUB 300
120  PRINT A,B
200  INPUT Z
210  GOSUB 410
220  PRINT A
230  STOP
300  IF X  LT 0 THEN 350
310  GOSUB 400
320  LET B=A/X
330  RETURN
350  LET A=B=0
360  RETURN
400  LET Z=SQR(X)
410  LET U=Y-Z
420  LET A=SQR(U*U+1)
430  RETURN
440  END
```

The GOSUB at line 110 sends the program to line 300. If X is less than zero, it goes to line 350, sets A and B to 0, and returns from line 360 to line 120. If X is greater than or equal to 0, the statements in the nested subprogram at line 400 are executed and then the program returns from line 430 to line 320 and then to line 120. The GOSUB at line 210 causes statements 410 and 420 to be executed before the program continues at line 220. The STOP at line 230 tells the computer not to execute any of the statements that follow. None of the GOSUB statements will be evaluated again.

In short, the only thing that cannot be done when using GOSUB and RETURN is to execute more RETURN statements than GOSUB statements. If an attempt is made to execute a RETURN when there are no unsatisfied GOSUBs, the program will be aborted.

Example:

```
100  INPUT F
200  A=F
300  X=1
400  GOSUB 800
500  PRINT F; "FACTORIAL IS: ";X
600  GO TO 100
700  STOP
800  X=X*A
900  A=A-1
1000 IF A. NE. 0 THEN GOSUB 800
1100 RETURN
1200 END
```

The above program is an example of a recursive GOSUB. Lines 800-1100 are a subprogram. Line 1000 is a conditional GOSUB. If A is not equal to 0 then a recursive call is made to line 800. This will happen until A is equal to zero or until the subprogram is executed 3960 times, at which point the user will receive a stack overflow error. To avoid the error, ensure that the condition which will take the user out of the subprogram eventually is met.

SECTION 6

MATRICES

GENERAL DESCRIPTION

The ability to perform matrix operations on arrays gives the user of BASIC a very valuable tool. In BASIC, any two-dimensional array may be used as a matrix. One-dimensional arrays are treated as row vectors. Column vectors can be used by creating a matrix with only one column. Therefore,

```
100 DIM A(5,10), B(5,1), X(10)
```

introduces a 5x10 matrix, a 5-element column vector, and a 10-element row vector.

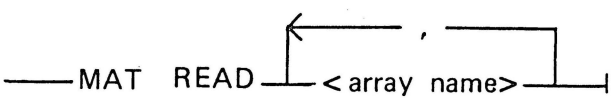
All statements specifying matrix operations have the word MAT immediately following the line number. Matrix statements may be mixed freely with other BASIC statements, which in turn may refer to specific elements of the arrays. The various matrix statements are described in the paragraphs that follow.

MATRIX I/O STATEMENTS

MAT READ STATEMENT

Syntax

<MAT READ statement>



Semantics

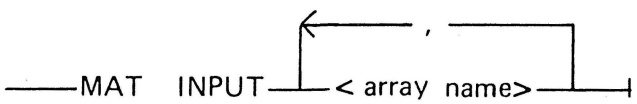
The MAT READ statement is used to read information into a matrix by using DATA statements. The matrix dimensions may optionally follow the array name but may not be changed to read a partial matrix. If these dimensions are not present, the dimensions specified by the DIM statement are used.

The MAT READ statement will cause information to be read into the matrix in row order, for example, A(1,1), A(1,2), A(2,1), A(2,2), A(3,1), and A(3,2).

MAT INPUT STATEMENT

Syntax

<MAT INPUT statement>



Semantics

The MAT INPUT statement reads data into a matrix from a remote terminal or an external file. When a MAT INPUT statement is executed, the matrix is filled in row order until the end of the data list is reached or the matrix is filled. The data list can be one line of input from an external file or remote device. When it is not possible to put all the data on one line of input (from remote devices), the use of the ampersand as the last data element will cause the program to continue looking for data. This will not work from an external file.

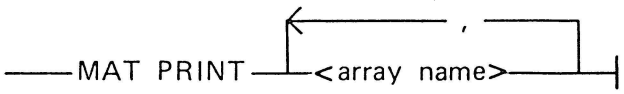
The data list may consist of numeric or unquoted string data separated by commas.

When using MAT INPUT with one-dimensional arrays (vectors), if the number of data elements input is less than the size of the vector, then the vector will be resized to be equal in size to the number of elements input. This will not occur with two-dimensional arrays.

MAT PRINT STATEMENT

Syntax

<MAT PRINT statement>



Semantics

The MAT PRINT statement is used to write information from a matrix. Reference to the individual elements in the matrix is not necessary. An entire matrix may be printed using the MAT PRINT statement. The last matrix name may be followed by a blank and the requested matrix will be printed in zoned format. The effects of the commas and semicolons are the same as for the PRINT statement. Each new row of the matrix is always begun on a new line.

Example:

```
100 DIM A(3,3)
200 MAT READ A
300 DATA 1,2,3,4,5,6,7,8,9
400 MAT PRINT A;
500 END
```

The resulting output is:

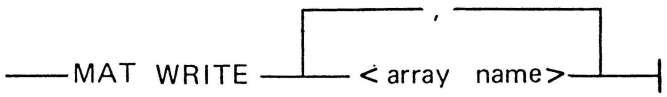
```
 1  2  3
 4  5  6
 7  8  9
```

The elements of a matrix are written in row order with each row of the matrix beginning on a new line.

MAT WRITE STATEMENT

Syntax

<MAT WRITE statement>



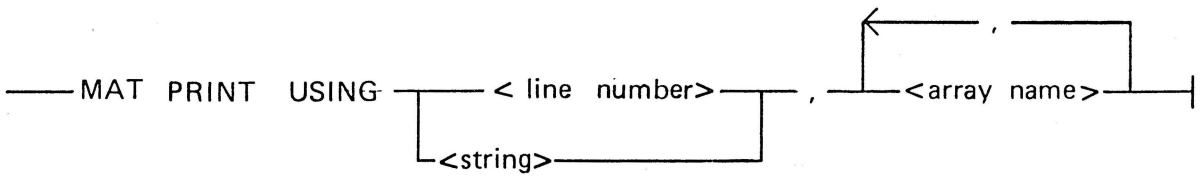
Semantics

The MAT WRITE statement functions similarly to the MAT PRINT statement except that the written data is followed by a comma.

MAT PRINT USING STATEMENT

Syntax

<MAT PRINT USING statement>



Semantics

The MAT PRINT USING statement is used to print formatted output of matrices. The line number or string expression is used to format each row of the matrix. Separating semicolons and colons have no effect on the output. Each matrix row begins formatting with the line number or string expression.

If the line number or string expression is less than the number of elements in a matrix row, the excess matrix elements are printed on a new line and the line number or string expression is reused beginning with the first field.

Example:

```
100 DIM A(3,2)
200 MAT INPUT A
300 MAT PRINT USING 400, A
400 : $###   $###
500 END
```

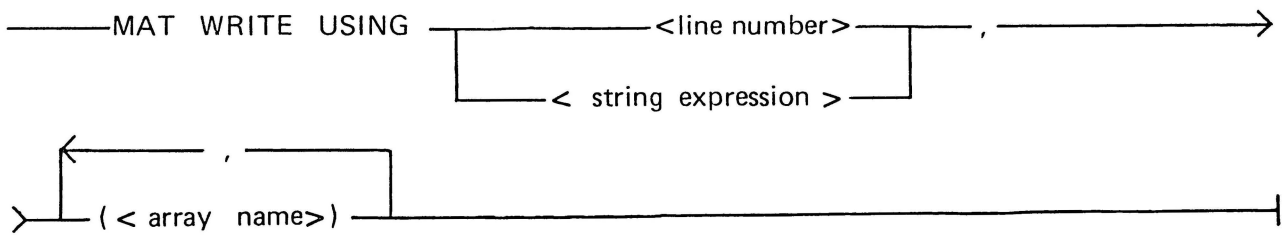
The resulting output is:

```
$ 12   $ 36
$440   $  0
*$5673 $ 47
```

MAT WRITE USING STATEMENT

Syntax

<MAT WRITE USING statement>



Semantics

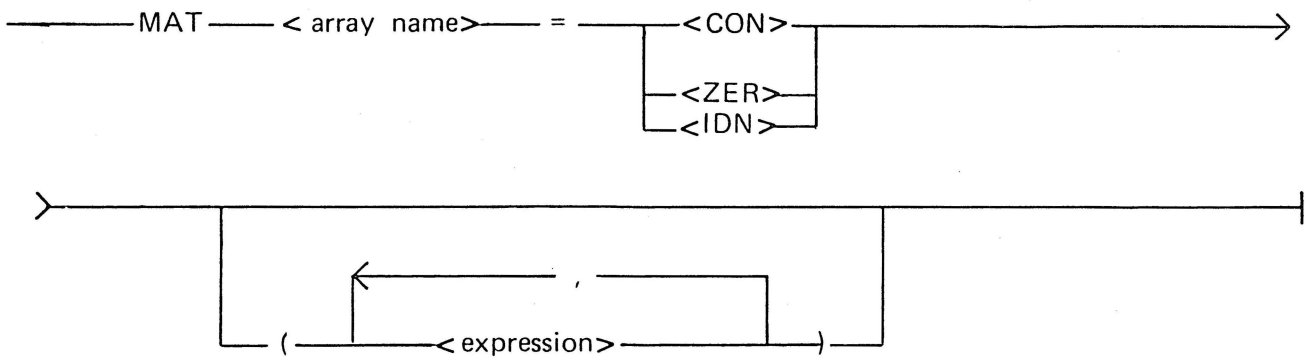
The MAT WRITE USING statement functions similiar to the MAT PRINT USING statement except that the written data is followed by a comma.

MATRIX FUNCTIONS

CON, ZER, AND IDN STATEMENTS

Syntax

<CON, ZER, and IDN statements>



Semantics

In addition to MAT READ and MAT INPUT, values may be assigned to matrices using CON, ZER, and IDN.

When CON (constant) is used, all elements of the specified matrix are assigned to value 1. ZER (zero) assigns 0 to all elements, and IDN (identity) assigns 1's to the elements along the diagonal and 0's elsewhere. IDN can only be used with square matrices.

If an optional expression is used, it signifies dimension information for the array name at execution time. (Refer to dimensioning).

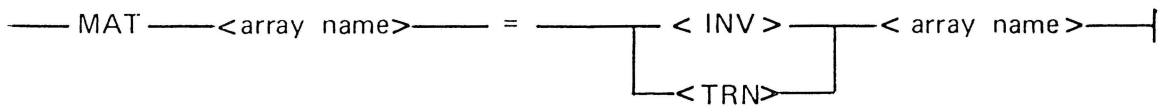
Example:

```
100 MAT C=CON
200 MAT D=IDN(2*N+1,2*N+1)
300 MAT S=ZER(2*N,M)
400 MAT T=CON(3*M)
```

INV AND TRN FUNCTIONS

Syntax

<INV and TRN functions>



Semantics

Matrices can be inverted or transposed using the INV (inversion) and TRN (transposition) functions.

Example:

```
10 MAT B=TRN(A)
20 MAT P=INV(B)
30 MAT P=INV(P)
```

The transposition function can be performed only on square matrices. For MAT A = TRN(B), both A and B must be of dimension (M,M). The inverse function can be performed only on matrices such that if MAT A = INV (B) and A is of dimension (M,N) then B must be of dimension (N,M). Also the syntax error, NUMBER OF DIMENSIONS NOT CONSISTENT, will occur if an attempt is made to invert a singular matrix.

Example:

```
100 DIM A(2,2), B(2,2), C(2,2)
200 MAT INPUT A
300 MAT B = TRN(A)
400 MAT C = INV(A)
500 MAT PRINT A
600 MAT PRINT B
700 MAT PRINT C
800 END
```

If the input was 1, 2, 3, 4 the output would be as follows:

```
1              2
3              4
1              3
2              4
-2             1
1.5            -.5
```

DET FUNCTION

Semantics

The DET function returns the value of the determinant of the matrix which was last inverted via the INV function. DET has no arguments and may be used in an arithmetic expression or be referred to directly by a PRINT statement.

Example:

```
100 DIM A(3,3), B(3,3)
200 MAT INPUT A
300 MAT B = INV(A)
400 MAT PRINT A
500 PRINT DET
600 END
```

If the input was 1, .5, .333333, .5, .888888, .25, .222222, .88, .2 the output would be as follows:

```
1               .5              .333333
.5             .8888888        .25
.222222        .88             .2

4.62924074-4
```

NUM FUNCTION

<NUM function>

The NUM (number) function returns the number of data elements entered into the last array which was filled by a MAT INPUT statement.

If the array has only one dimension (a vector), and the number of data elements input to that vector via the MAT INPUT statement is less than the size of the vector, the vector will be resized to be as large as the number of data elements which were input. NUM will contain the number of data elements input, which also will be the new size of the vector.

Example:

```
100 DIM A(15)
200 MAT INPUT A
300 PRINT "NUM =";NUM
400 A(7) = 10
500 END
```

When the above program is executed, the statement in line 200 will wait until the data elements for the array A have been supplied. If the data consists of values 1, 2, 3, 4 and 5, then A(1) through A(5) will be assigned the values one through five, and A will be resized to be only five elements long. Line 300 will then produce the output:

```
NUM = 5
```

After this operation, line 400 causes an invalid index error because A is only five elements long and line 400 attempted to index the seventh element of A.

NOTE

The programmer can resize vector A back to 15 elements later in the program by using the statement:

```
350 MAT INPUT A(15)
```

and then, when line 350 is executed, 15 data elements need to be supplied.

For two-dimensional arrays filled by a MAT INPUT statement, NUM will contain the number of data elements input to the array. The array will not be resized if the number of data elements input is less than the size of the array.

For both two-dimensional and one-dimensional arrays, if the number of data elements input is more than the size of the array, then the MAT INPUT statement will ignore the excess data and NUM will equal the size of the array when it was dimensioned.

DIMENSIONING

Before a matrix variable can be used in a matrix calculation, it must be dimensioned. There are three ways to assign dimensions to matrices:

a. The DIM statement.

b. Using the variable as a subscripted variable.

c. IDN, CON, ZER, and the MAT READ statements.

The DIM statement is used to assign the indicated dimensions to a variable. When variables which are not included in a DIM statement are used with subscripts in ordinary BASIC statements, they are assigned the dimension 10 if they are one dimensional, and 10x10 if they are two dimensional. If a variable is not dimensioned by either of these methods, the first MAT statement in which it appears must be a MAT READ, CON, ZER, or IDN.

MAT READ, CON, ZER, and IDN are used to dimension matrices during program execution and, therefore, take precedence over the other two methods. The expressions which may follow a variable in a MAT READ statement specify the dimensions for that variable. With CON, ZER, and IDN, the expressions specify dimensions for the matrix variable on the left of the equal sign. The expressions are evaluated when the statement is executed, and the values are used to dimension the associated matrix variable, whether or not the variable was already dimensioned. Therefore, in addition to specifying dimensions, these statements can be used to change the dimensions of a matrix during execution. The new dimensions are used in all subsequent calculations until the matrix is redimensioned once again. There are no restrictions on dimensions assigned in this manner.

Example:

```
100 DIM A(2,2),E(8,10)
110 LET D(2,3)=5
120 MAT READ A,B(2,2)
130 MAT C=ZER(4,6)
140 READ M,N
150 MAT A=CON(5,7)
160 MAT READ B(M,2*N+1),C,D,E(4,6)
```

The DIM statement established a 2x2 matrix called A, and an 8x10 matrix called E. The use of D in the LET statement causes it to be dimensioned 10x10. The READ statement at line 120 reads four numbers into A, and then four numbers into B (which is specified as a 2x2 matrix). Statement 130 illustrates the use of ZER to dimension C and to set its elements to zero.

In line 150, A is redimensioned to a 5x7 matrix. In the READ statement on line 160, the dimensions of B are set to the values of the expressions (which may be larger or smaller than the old dimensions). Data is read into B and E using their new dimensions, and into C and D using the old dimensions as specified at line 130 and line 110, respectively.

Note that all statements which redimension a matrix also cause new values to be assigned to its elements. As a result, all data originally in a redimensioned matrix is lost.

Since the regular BASIC instructions in combination with the MAT instructions provide the user with a very powerful tool, special care must be taken when working with dimensions. When an element of an array is used in an ordinary BASIC statement, the subscripts refer to the most recently executed dimensions. For instance, in the above example, the use of E(5,1) after line 160 would cause an indexing error.

Also, be careful in MAT statements to ensure that the resulting matrix is large enough to handle the results of the operation performed. For example, the statement MAT X=Y*Z may be illegal because either:

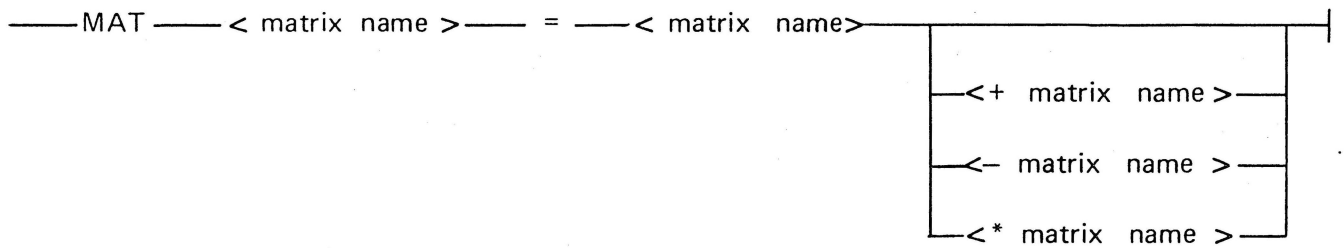
a. The dimensions of Y and Z are such that it is not possible to define the product, or

b. X may not be dimensioned sufficiently to handle the product.

Either case causes an indexing error.

Matrix Arithmetic Statements

<Matrix arithmetic operations>



Examples:

```
100 MAT A=B
110 MAT A=B+C
120 MAT M=M-N
130 MAT B=D*F
140 MAT B=A*A
```

In the first line, the values of matrix B are assigned to matrix A. In a statement like this, the matrices must have the same dimensions. In each of the other lines, the indicated operation is performed and the result is assigned to the matrix on the left. For the addition or subtraction of matrices, the matrix on the left and the two matrices on the right of the equal sign must all have the same dimensions. Matrix multiplication of the form MAT A=A*D or MAT A=B*A is illegal. (The same <matrix name> cannot appear on the right and left of the equal sign).

Example:

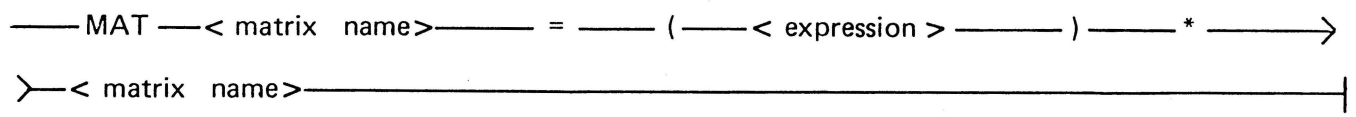
```
100  DIM A(4,3), B(4,3), C(4,3), D(4,2), F(2,3)
200  MAT INPUT B,C,D,F
300  MAT A=B+C
400  PRINT "MATRIX B"
500  MAT PRINT B
600  PRINT "MATRIX C"
700  MAT PRINT C
800  PRINT "MATRIX A=B+C"
900  MAT PRINT A
1000 MAT B=D*F
1100 PRINT "MATRIX D"
1200 MAT PRINT D
1300 PRINT "MATRIX F"
1400 MAT PRINT F
1500 PRINT "MATRIX B=D*F"
1600 MAT PRINT B
1700 END
```

When the above program is executed with the input 1,2,3,4,5,6,7,8,9,10, 11,and 12 for Matrix B; 0,1,2,3,4,5,6,7,8,9,10, and 11 for Matrix C; 2,-1,0,2,3,-2,0, and 1 for Matrix D; and 1,2,3,1,2, and 3 for Matrix F, the following is output:

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

MATRIX C
0              1              2
3              4              5
6              7              8
9              10             11

MATRIX A=B+C
1              3              5
7              9              11
13             15             17
19             21             23

MATRIX D
2             -1
0              2
3             -2
0              1

MATRIX F
1              2              3
1              2              3

MATRIX B=D*F
1              2              3
2              4              6
1              2              3
1              2              3
```

Matrix scalar multiplication

<matrix scalar multiplication>



Semantics

The parentheses must always be present since they indicate scalar multiplication rather than matrix multiplication. The expression can be any arithmetic expression.

Examples:

```
10 MAT A=(3)*B
20 MAT R=(N)*R
30 MAT T=(6.3+T(1,3))*Q
```

In the final statement, the expression is evaluated before multiplication takes place so that the original value of T(1,3) is used throughout the scalar multiplication.

Each statement can contain only one matrix operation.

Sample Program 1:

The sample program below illustrates some of the capabilities of matrix operations in BASIC. First, two matrices (in this particular example, vectors) are dimensioned at line 100. Then the value of J is read in. This value is used to dimension and zero out another matrix at line 300, and to dimension and read in two matrices at lines 400 and 700. Immediately after being read in, these matrices are printed out at lines 600 and 900 as an echo-check to verify the input. Then one matrix is subtracted from the other at line 1100 and the result is printed out at line 1200. Then two vectors are read in at line 1300 and printed out at lines 1500 and 1700, after which a matrix is zeroed out and dimensioned at line 1900. The two vectors are multiplied together and the result placed in the newly dimensioned matrix at line 2000, after which it is printed out.

```
100  DIM P(5,1), Q(1,5)
200  READ J
300  MAT W=ZER(J,J)
400  MAT READ X(J,J)
500  PRINT "FIRST MATRIX"
600  MAT PRINT X
700  MAT READ Y(J,J)
800  PRINT "SECOND MATRIX"
900  MAT PRINT Y;
1000  PRINT "FIRST MINUS SECOND"
1100  MAT W=X-Y
1200  MAT PRINT W;
1300  MAT READ P,Q
1400  PRINT "COL. VECTOR"
1500  MAT PRINT P;
1600  PRINT "ROW VECTOR"
1700  MAT PRINT Q;
1800  PRINT "COL. TIMES ROW"
1900  MAT R=ZER(5,5)
2000  MAT R=P*Q
2100  MAT PRINT R;
2150  DATA 3
2200  DATA 1,0,0,0,1,0,0,0,1,-4,-3,-2,-1,0,-1,-2,-3,-4
2300  DATA -2,-1,0,1,2,2,1,0,-1,-2
2400  END
```

The following illustrates the output results of the sample program.

```
FIRST MATRIX

 1              0              0
 0              1              0
 0              0              1

SECOND MATRIX

  -4 -3 -2
  -1 -0 -1
  -2 -3 -4

FIRST MINUS SECOND

  5  3  2
  1  1  1
  2  3  5

COL. VECTOR

  -2
  -1
   0
   1
   2

ROW VECTOR

  2  1  0 -1 -2

COL. TIMES ROW

  -4 -2  0  2  4
  -2 -1  0  1  2
   0  0  0  0  0
   2  1  0 -1 -2
   4  2  0 -2 -4
```

Sample Program 2:

```
    100  DIM A(3,3), B(3,3), C(3,3)
    200  MAT INPUT A, B
    300  PRINT "MATRIX A"
    400  MAT PRINT A
    500  PRINT "MATRIX B"
    600  MAT PRINT B
    700  MAT A=(3*A(1,2))*B
    800  MAT PRINT "MATRIX A=(3*A(1,2))*B"
    900  MAT PRINT A
    1000 END
```

When the above program is executed with the input 1,2,3,4,5,6,7,8, and 9 for Matrix A and 0,1,2,3,4,5,6,7, and 8 for Matrix B, the following is output:

```
MATRIX A
1               2               3
4               5               6
7               8               9

MATRIX B
0               1               2
3               4               5
6               7               8

MATRIX A=(3*A(1,2))*B
0               6               12
18              24              30
36              42              48
```

SECTION 7

EXTERNAL DATA FILES AND FUNCTIONS

GENERAL DESCRIPTION

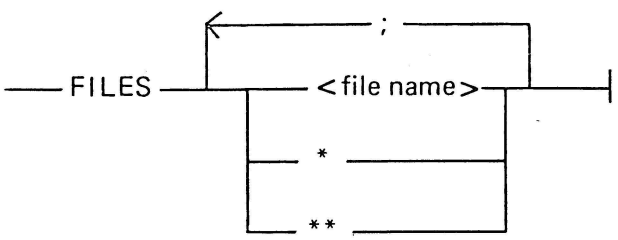
In many cases it is inconvenient to have all input and output go through the remote console. One feature of BASIC is the ability to read or write disk data files under program control. All of the rules of file security are applicable to BASIC files. Only those files that the user can access from the console can be read by his program. Files written by a user's program are treated as if they had been saved and, therefore, become a protected part of his files. File names must contain six or fewer characters, no special characters, and must begin with a letter.

DECLARING AND ACCESSING FILES

FILES STATEMENT

<FILES statement>

Syntax



All files to be used in a BASIC program must be explicitly mentioned in a FILES statement before the first use of the file. A program may contain more than one FILES statement. All files declared in the FILES statements are accumulated in one area of the program, which is referred to as the FILES LIST. The files can then be referenced by their position in this list.

Example:

```
100 FILES A;B
200 FILES *;D

File A can be referenced as File #1.
File B can be referenced as File #2.
File * can be referenced as File #3.
File D can be referenced as File #4.
```

The maximum number of files which can be declared is 16. More than 16 files may be accessed by a BASIC program, as will be discussed later, but at any one time only 16 files may be open.

The INTNAME (internal name) attribute for files declared in a BASIC program is of the form FILE N, where N is an integer between 1 and 16. When a label equation is necessary, this INTNAME is used to refer to files in the program.

A <filename> must begin with an alphabetic character and contain no more than six alphanumeric characters.

A single asterisk is used to reserve a spot in the FILES list for a file which will be specified at a later time. Before a file declared with an asterisk can be used, it must be explicitly opened with a FILE statement which specifies that file's <filename>.

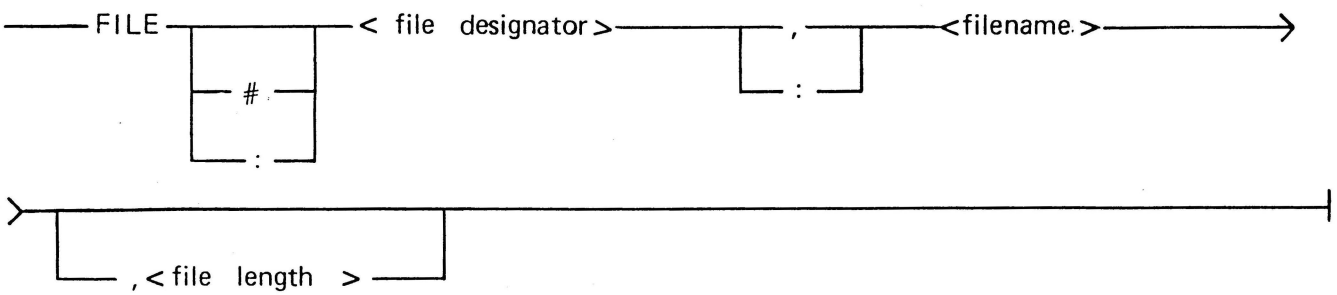
The FILES statement, when followed by a <filename> or a single asterisk, is strictly declarative. It does not cause files to be opened or closed.

Two asterisks are used to specify a scratch file and to explicitly open that file. Unlike other files, scratch files are opened in the WRITE mode, which is more fitting considering their use. A scratch file is used only during the running of the program in which it is declared, and is purged when the program is completed or if the file is explicitly closed with a FILE statement. It is not necessary to specify a name for a scratch file. (Examples are given in the discussion of the FILE statement).

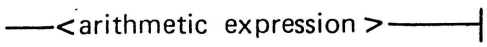
FILE STATEMENT

Syntax

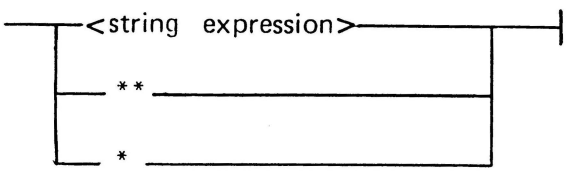
<FILE statement>



<file designator>



<filename>



Semantics

The FILE statement is used to explicitly open and close external files.

When the <filename> is a string, it must begin with an alphabetic character and contain no more than six alphanumeric characters.

When the <filename> form is used, the name of the file is changed to the name specified by the <filename>. If the file was closed, it is now opened in the READ mode. It is an error to use a FILE statement of this form if the file is already open and any statement other than a FILES or FILE statement has already referenced the file. If the <filename> is in quotes, that is, FILE #1, "F", the file named F is opened for reading or writing depending on the mode of file #1.

When the <filename> is an asterisk enclosed in quotation marks, the file specified by the <file designator> is closed. If the specified

file was a scratch file, then it will be purged. Otherwise the file will be saved.

If the <filename> is two asterisks enclosed in quotation marks, the designated file is typed scratch. If the file was closed,it is now opened in the WRITE mode.

EBCDIC AND BINARY FILES

External files can be either EBCDIC or binary. In an EBCDIC file the data is represented by a set of EBCDIC character codes. When accessing data from an external EBCDIC file, the data must be converted from EBCDIC characters to a binary representation to be used by the computer. The reverse conversion, from binary to EBCDIC, is necessary when doing output to an EBCDIC file. However, with binary files all data is stored in binary representation and thus, no conversion of data is needed when inputting or outputting to a binary file. It is for this reason that input and output to external binary files is quicker than input and output to external EBCDIC files. Since the data in a binary file is in binary representation, the file cannot be edited by CANDE editing commands. Specifically, binary files can be read or written only by a BASIC program.

External binary files may be accessed either sequentially or randomly. If a file is to be created as a random access binary file, the <file length> must be specified. If a binary file is declared without a <file length> , the file is created as a sequential access file. An EBCDIC file can only be accessed sequentially.

If a particular statement can be used with either EBCDIC or binary files, the only distinction between the two statements is the use of the "#" (for EBCDIC files) and the ":" (for binary files).

Further references to a file are made by specifying a <file designator> which represents the position of that file in the program's FILES list. If the value of the <file designator> is not an integer, it is truncated to an integer.

Example:

```
10 FILES FILE1; FILE2; FILE3
20 FILE :1,200
30 READ #2, X, Y, Z
40 READ :3, D, E, F
```

In the example above, line 10 specifies three files. The FILE statement at line 20 references FILE1 with a colon file designator and specifies a file length of 200 words. Thus, FILE1 will now be treated as a binary random access file 200 words long. The statements at line 30 and 40 are the first references to files FILE2 and FILE3 respectively. Since no file length has been specified for either, FILE2 will now be treated as an EBCDIC sequential file and FILE3 will now be treated as a binary sequential file.

BASIC puts mode restrictions on sequential files. A sequential file which is being processed by a BASIC program is defined as in either the READ or the WRITE mode. When a file is first opened it is in the READ mode. Before any output can be performed on that file, it must be placed in the WRITE mode. A file can be placed in the WRITE mode by executing either a SCRATCH statement or an APPEND statement. Similarly, a file which is in the WRITE mode must be placed in the READ mode before any input can be obtained from that file. A file which is in the WRITE mode can be placed in the READ mode by executing either a RESTORE statement or a BACKSPACE statement. These mode restrictions do not apply to random access files.

Example:

```
10 FILES A
20 PRINT #1, 12, 26, 71.9
30 END
```

If the above program was run, it would be aborted with the run time error ILLEGAL I/O COMMAND because line 10 opens file A in READ mode by default and line 20 attempts to write to this file. The error could be corrected by inserting this statement:

```
15 SCRATCH #1
```

Which would purge all information in file A and place the file in the WRITE mode.

Example:

```
100 FILES FILEA;*
110 FILES *; **; FILEF; **; FILE2
120 READ #1, A, B, C
130 FILE #2, "FILEB"
140 FILE :4, 200
150 PRINT #6, A, B, C
160 FILE :3, "FILED"
```

Lines 100 and 110, in the program segment above, specify seven files. After these two lines have been executed, the following statements can be made about the seven files.

File one is named FILEA; it is open and in the READ mode.

File two has not been specified and thus is closed.

File three has not been specified and thus is closed.

File four is a scratch file; it is open and in the WRITE mode.

File five is named FILEF; it is open and in the READ mode.

File six is a scratch file; it is open and in the WRITE mode.

File seven is named FILE2; it is open and in the READ mode.

After the execution of lines 120 through 160 the following information is known:

File one is a sequential access EBCDIC file.

File two is a sequential access EBCDIC file named FILEB; it is open and in the READ mode.

File three is a binary file named FILED; it is open and in the READ mode.

File four is a random access binary file 200 words long. This is known because line 140 referenced the file with a ":" file designator (binary) and specified a file length of 200 words.

File six is a sequential access EBCDIC scratch file.

Example:

```
100 FILES FILEA; *; *; FILED
110 SCRATCH #4
120 READ #1, A, B, C
130 F$ = "FILEC"
140 FILE :3, F$
150 FILE #1, "*"
160 WRITE #4, A, B, C
170 FILE :1, "FILEA2"
180 SCRATCH :1
190 FILE #2, "**"
200 FILE :1, "*"
210 FILE #1, "FILEA"
```

FILEA and FILED are opened by their appearance in the FILES statement at line 100. Line 140 opens the third file in the files list as a binary sequential file with the name FILEC. At this point line 100 has become in effect:

```
100 FILES FILEA; *; FILEC; FILED
```

Line 150 then closes FILEA. With the first file in the files list now closed, line 170 opens a new binary file, FILEA2. Line 100 is now in effect:

```
100 FILES FILEA2; *; FILEC; FILED
```

Line 190 makes the second file in the files list a scratch file. Lines 200 and 210 close the binary file FILEA2 and reopen the FILEA. (When FILEA is reopened, it is in the READ mode and the first READ on the file will begin reading with the first data item in the file.) Line 100 has now become in effect:

```
100 FILES FILEA; **; FILEC; FILED
```

As mentioned earlier, the maximum number of files that a BASIC program can have declared is 16; however, more than 16 files can be used by one BASIC program. File one in the example above shows that a file may be closed with a FILE statement and another file may be opened. The new file can have the same value for its <file designator> that the closed file had.

Another use of the FILE statement is to classify a given binary file as being a random access file and to specify the length of that file in words. The syntax of a FILE statement which specifies a FILE as being random access binary is the same as the syntax for FILE statements already discussed with one addition. Following the <filename> is an arithmetic expression which specifies the length of the file and is separated from the <filename> by a comma. If the file is already open, then only the length of the file need be specified, to create it as a random file. Any FILE statement which references a binary file and specifies a length for that file will make that file a random access binary file.

NOTE

The number of words specified for the file in the FILE statement will all be assigned the value zero.

Example:

```
100 FILES *; B2; B3; *; **
110 FILE :1, "B1", 300
120 A = 100
130 FILE :2, A
140 FILE :3, "**", 300
150 FILE :4, "**", A*3
160 FILE :5, 300
170 WRITE :5, A, B, C
180 FILE :5, "*"
```

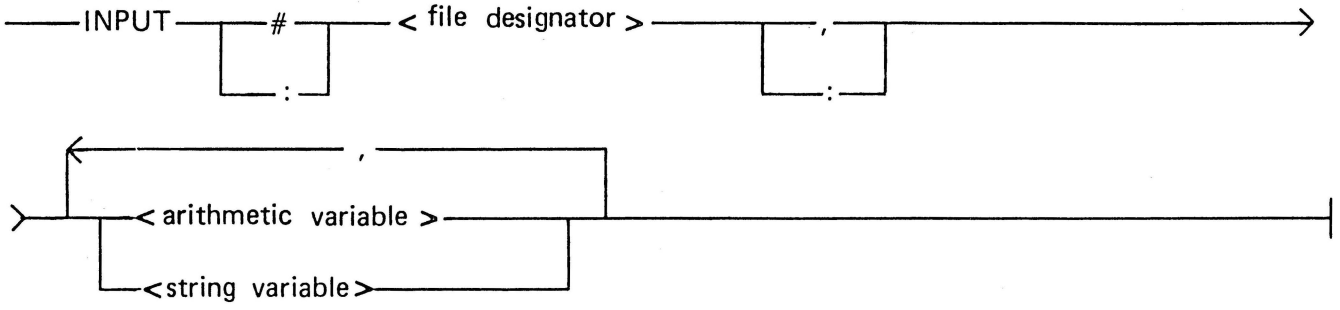
In the program segment above, five files are declared. After the execution of line 160, all five files have been classified as random access binary files. Line 110 opens file one with the name B1 and specifies the length of the file to be 300 words. Since file two, B2, was opened in the FILES statement in line 100, all that is necessary to make it a random access binary file is to reference it with a ":" <file designator> and to specify the length of the file (both of which are done in line 130). Lines 140 and 150 make files B3 and file four respectively, random access binary scratch files 300 words long. Line 160 makes file five, which is already a scratch file, a random access binary file. When closing a binary random file (line 180) it is not necessary to specify the length of the file.

FILE I/O STATEMENTS FOR EBCDIC FILES

FILE INPUT STATEMENT

Syntax

<FILE INPUT statement>



Semantics

Input from external files is accomplished through the use of the FILE INPUT statement and the FILE READ statement.

If the value of the <file designator> is zero or the <file designator> is not present, the data will be input from the user terminal. As indicated, the INPUT statement may be used with binary files. The discussion in this section, however, will be concerned with EBCDIC files. Binary files will be discussed later.

The INPUT statement, as used with external EBCDIC files, is designed to input from files which have been created without line numbers at the beginning of each line. (Files created with the PRINT statement or files created through CANDE as type CDATA files.) If the file does have line numbers, the INPUT statement will treat them as numeric data.

When doing INPUTS from external files the type of the variable in the INPUT list must be the same as the type of the data item in the file. If the type of variable in the INPUT list is not the same type as the data item in the file, the program is aborted. Also, when reading from external files, the INPUT statement does not recognize when it has run out of data. When the data has been exhausted on an external file, then all further INPUTS on that file will cause the value of zero to be assigned to numeric variables and the value of null string to be assigned to all string variables. End of data can be detected for external files by using the IF END and IF MORE statements (discussed later).

Suppose that the following code appeared in a program.

```
10 FILES DATA1
20 INPUT #1, N$(1),N$(2)
30 FOR X=1 TO 5
40 INPUT #1, A(X),B(X)
50 PRINT N$(1),N$(2),A(X),B(X)
60 NEXT X
70 END
```

To generate a data file for this code, the following could be done at the terminal before the BASIC program containing the code was run.

```
MAKE DATA1 CDATA

10 "STRING1", "STRING2"
20 1, 1.5
30 2, 2.5
40 3, 3.5
50 4, 4.5
60 5, 5.5

SAVE
```

When the program code is run, the output will now be:

```
STRING1        STRING2        1              1.5
STRING1        STRING2        2              2.5
STRING1        STRING2        3              3.5
STRING1        STRING2        4              4.5
STRING1        STRING2        5              5.5
```

Each time an INPUT statement is executed a new line of data is read from the file, even if all the data from the previous line has not been referenced.

```
5  FILES DATA1,DATA2
10 FOR I = 1 TO 3
20 INPUT #2, A(I),B(I)
25 PRINT A(I),B(I)
30 NEXT I
40 END
```

Where file #2 contains the following three lines of data

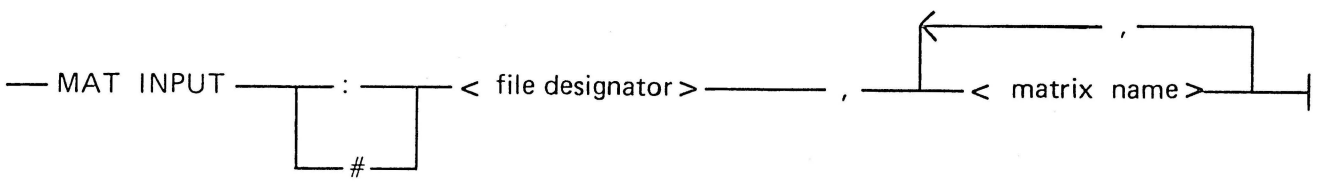
```
10  1 , 2 , 3 , 4
20  5 , 6 , 7 , 8 , 9 , 10
30  11 , 12 , 13
```

The results are:

```
A(1) = 1     B(1) = 2
A(2) = 5     B(2) = 6
A(3) = 11    B(3) = 12
```

FILE MAT INPUT STATEMENT

<FILE MAT INPUT statement>



The MAT INPUT statement can be used to read data from a file. For each MAT INPUT statement in a program there must be a corresponding DATA statement. This DATA statement must contain all of the data needed for the matrix. Any extra data elements will be set to zero. It is illegal to use an "@" to signal the computer to use the next DATA statement. (It was legal when using MAT INPUT from a remote terminal).

Example:

```
100 FILES WBDT, WBDT2, WBDT3
200 DIM A(3,2), B(3,2), C(3,2)
300 MAT INPUT #1, A
400 MAT PRINT A
500 MAT INPUT #2, B
600 MAT PRINT B
700 MAT INPUT #3, C
800 MAT PRINT C
900 END
```

If the files contained the input:

FILE WBDT

```
100 1, 2, 3, 4, 5, 6
```

FILE WBDT2

```
100 1, 2
200 3, 4
300 5, 6
```

FILE WBDT3

```
100 1, 2, *
200 3, 4, *
300 5, 6
```

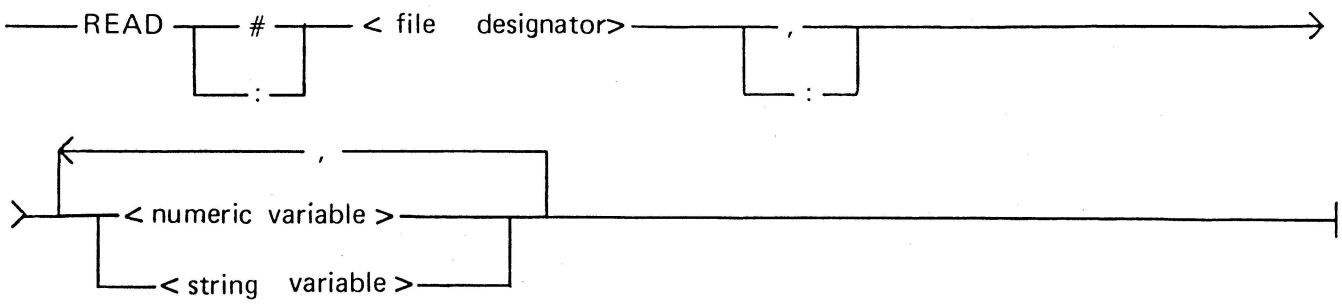
the following would be output:

```
1               2
3               4
5               6

1               2
0               0
0               0

BASIC ERROR ***** NUMBER IS BADLY FORMED
```

FILE READ STATEMENT

Syntax

<FILE READ statement>



Semantics

As indicated, the READ statement may be used with binary files. The discussion in this section, however, will be concerned with EBCDIC files. Binary files will be discussed later.

The READ statement, as used with external EBCDIC files, is designed to read files which have been created with line numbers at the beginning of each line. (Files created with the WRITE statement or files created through CANDE as type CSEQDATA files.) When a READ statement is executed on an external file, the first number on the input line of data is considered to be a line number and its value is not assigned to variables in the READ list.

When doing READs from external files, the type of the variable in the READ list must be the same as the type of the data item in the file. If the type of variable in the READ list is not the same type as the data item in the file, the program is aborted. Also, when reading from external files, the READ statement does not recognize when it has run out of data. When the data has been exhausted on an external file, then all further READs on that file cause the value of zero to be assigned to numeric variables and the value of null string to be assigned to all string variables. End of data can be detected for external files by using the IF END and IF MORE statements (discussed later).

Example:

```
100 FILES A
110 SCRATCH #1
120 WRITE #1, 10; 20; 30
130 WRITE #1, 40; 50; 60
140 RESTORE #1
150 READ #1, A, B, C
160 PRINT A; B; C
170 READ #1, A, B, C
180 PRINT A; B; C
190 READ #1, A, B, C
200 PRINT A; B; C
999 END
```

When this program is executed, lines 100 through 130 will create file A which contains:

```
100    10  ,  20  ,  30  ,
110    40  ,  50  ,  60  ,
```

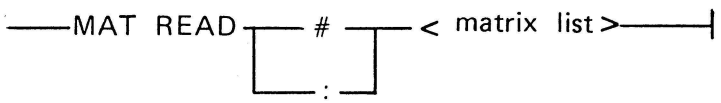
Line 140 then RESTOREs the input data pointer to the beginning of the file and places the file in READ mode. Lines 150 through 200 produce the following output:

```
10  20  30
40  50  60
0   0   0
```

The READ statements at lines 150 and 170 treated the first number on each line as line numbers (as they should) and did not assign these values to variables in the READ list. When the READ statement at line 190 was executed, the data in the file had been exhausted, so the value of zero was assigned to the variables A, B, and C.

FILE MAT READ STATEMENT

<FILE MAT READ statement>



The MAT READ statement also may be used to read information from a file. Unlike the MAT INPUT statement, the DATA statement does not need to contain all the data for a matrix on one line. The "@" is still illegal.

If the MAT INPUT example was run using MAT READ, the following would be output:

```
1             2
3             4
5             6

1             2
3             4
5             6

BASIC ERROR ***** NUMBER IS BADLY FORMED
```

Note the difference between using MAT INPUT or MAT READ to run the same program. Unlike INPUT and READ statements, MAT INPUT and MAT READ can use the same files.

Comparison of the READ and INPUT statements

When no file designator is specified for a READ statement the data is read from DATA statements. For INPUT statements, if no file designator is specified the data is input from the user terminal. Therefore, a clear distinction can be seen between the source of data for the READ and INPUT statements when the data is not located in external files.

Both statements can be used to retrieve data from external EBCDIC files and can in fact be used on the same file. It is the manner in which each statement retrieves the data that makes the statements different.

The INPUT statement is line-oriented. When an INPUT statement is executed on an external EBCDIC file, data elements are read from the file until the input list has been satisfied. When this occurs, the data pointer is positioned at the beginning of the next line, whether or not more data is contained in the line from which the last item was read. The INPUT statement also assumes that the file from which it is reading does not contain line numbers. Thus, if the file actually does contain line numbers, the INPUT statement will treat them as numeric data. The INPUT statement expects all data elements in the file to be separated from each other by commas (unless the delimiter has been changed by a DELIMIT statement).

The READ statement is data-element-oriented. When a READ statement is executed on an external EBCDIC file, data elements are read from the file until the READ list has been satisfied. When this occurs, the data pointer is positioned at the data element immediately following the last data element read. Thus, all data elements on the line of a file can be read by using one or more READ statements. The READ statement also assumes that the file from which it is reading contains line numbers. These line numbers are not assigned to variables in the READ list. The READ statement expects all data elements in the file to be separated from each other by commas (unless the delimiter has been changed by a DELIMIT statement).

Both the READ and the INPUT statement do not recognize when they have exhausted the data on an external EBCDIC file. If either statement reads past the last data elements on the file, then subsequent INPUTs (or READs) from that file will cause the value of zero to be assigned to all numeric variables and the value of null string to be assigned to all string variables.

Example:

```
100 FILES A
110 READ #1, A, B, C
120 PRINT "READ TEST"
130 PRINT A; B; C
140 READ #1, A, B, C
150 PRINT A; B; C
160 RESTORE #1
170 PRINT "INPUT TEST"
180 INPUT #1, A, B, C
190 PRINT A; B; C
200 INPUT #1, A, B, C
210 PRINT A; B; C
999 END
```

If file A was created through CANDE as a CSEQDATA file with the following data,

```
100   1, 2, 3, 4, 5
110  11, 22, 33, 44, 55
```

then the program above would produce the following output:

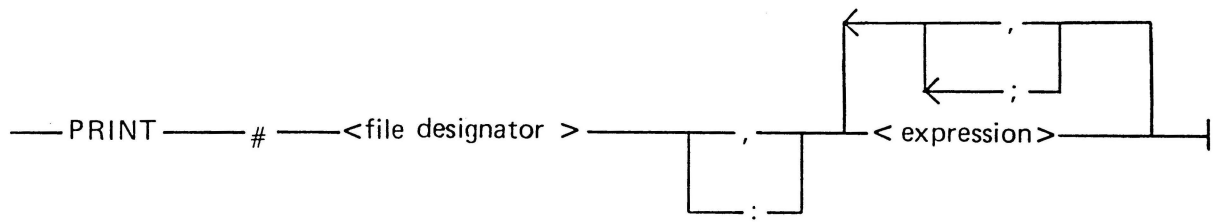
```
READ TEST
1  2  3
4  5  11
INPUT TEST
1001  2  3
11011  22  33
```

Observe that the INPUT statement treated the line numbers as numeric data, since INPUT does not expect to be reading from a line numbered file.

FILE PRINT STATEMENT

Syntax

<FILE PRINT statement>



Semantics

Output of data to external files is done through the FILE PRINT and FILE WRITE statements. It is not legal to use the PRINT statement when outputting data from binary files.

The TAB function, the comma, and the semicolon, when used as delimiters for print list items, control the spacing of data of an external file, as they do for PRINT statements to the user terminal. If the PRINT statement to an external file is terminated with a comma or a semicolon, the next PRINT statement to that file will not begin on a new line, but will continue printing after the last data item printed on the previous line.

Example:

```
100 FILES ONE; TWO
110 SCRATCH #1
120 SCRATCH #2
140 PRINT #0, "THE PROGRAM HAS BEGUN"
150 PRINT
160 PRINT #1, 12, 13, 14, 15
170 PRINT #1, 16,
180 PRINT #1, "END #1"
190 PRINT #2, 1; 2; 3
200 PRINT #2, 4;
210 PRINT #2, TAB(18); "END #2"
220 PRINT #0, "THE PROGRAM IS COMPLETED"
999 END
```

This program will produce output in the following form:

```
THE PROGRAM HAS BEGUN

THE PROGRAM IS COMPLETED
```

Note that PRINT statements with file designators of zero, and those with no file designator at all, refer to the terminal. Also, no SCRATCH statement was necessary to print to the user terminal.

File one now contains:

```
12              13              14              15
16            END #1
```

File two now contains:

```
1  2  3
4               END#2
```

If file one and file two were listed at the terminal with the CANDE LIST command they would appear like this:

```
100  12              13              14              15
200  16            END #1
```

and

```
100  1  2  3
200  4                 END #2
```

The line numbers that appear are supplied by CANDE as a convenience. They can be used to edit the file but do not actually appear in the file.

External EBCDIC files are created by the compiler with the attributes:

```
KIND=DISK, UNITS=CHARACTERS, MAXRECSIZE=81, BLOCKSIZE=1620
INTMODE=EBCDIC, BUFFERS=2, FILETYPE=0, SAVEFACTOR=15
```

To create a file through CANDE which is compatible with the BASIC external files, it is necessary to make the file type CDATA. Files created as type CDATA are unsequenced character mode files compatible to those created by the file PRINT statement.

Example:

```
100 FILES ONE
110 SCRATCH #1
120 PRINT #1, 1; 2; 3
130 END
```

The above program would produce a file ONE, which would be the same as the file CONE created with the following CANDE commands.

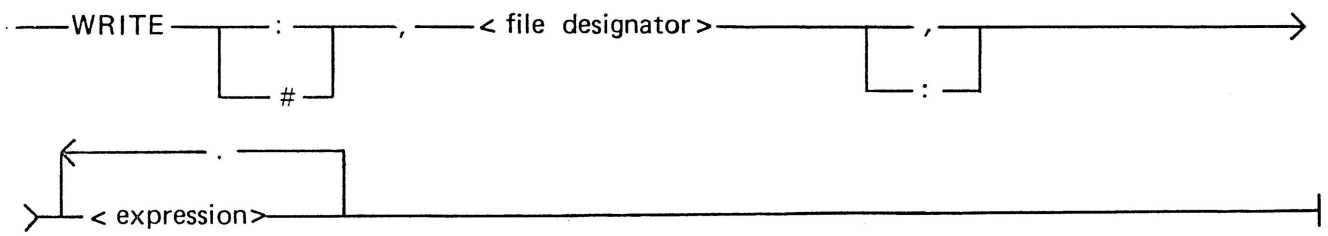
```
MAKE CONE CDATA

100  1  2  3
SAVE
```

The sequence number is furnished so that CANDE will not interpret the first data value 1 as a sequence number. The number 100 is not actually stored in the file CONE. When creating these files through CANDE, it is not necessary to enclose string data in quotation marks.

FILE WRITE STATEMENT

Syntax

<FILE WRITE statement>



Semantics

If the <file designator> has a value of zero or if it is not present, then the <expression>s are output to the user's terminal. As indicated in the syntax, the WRITE statement may be used with binary files. The discussion in this section, however, will be concerned with EBCDIC files. Binary files will be discussed later.

The FILE WRITE statement is similar to the FILE PRINT statement in its use. Any item in a WRITE list must be separated from any following item by a comma or a semicolon. The effects of the comma and semicolon on spacing between data items on the output line is the same as for the PRINT statement. A WRITE statement will cause data in the output list to be printed and the system to advance to the next line on the output device. If the WRITE statement is terminated by a comma or semicolon, the current line will be held and the next output to the file will begin printing where the last output stopped.

Unlike PRINT statements, any data written to a file by a WRITE statement is always followed by a delimiter. The standard delimiter is the comma, but this may be changed with a DELIMIT statement (discussed later). When WRITE statements are to external EBCDIC files, a five-digit line number is generated at the beginning of each output line. These line numbers begin at 100 and are incremented by 10.

Example:

```
100 FILES FILEA; FILEB
110 SCRATCH #1
120 SCRATCH #2
130 WRITE "SAMPLE OF WRITE STATEMENT"
140 A = 36
150 B$ = "ABC"
160 WRITE #1, 1, 22, 333, 4444
170 WRITE #1, A, SQR(A)
180 WRITE #1, A; -A; 4+A; -47;
190 WRITE #1, 99.9
200 WRITE #2, "STRING ITEM", "EXTRA LONG STRING"
210 WRITE #2, B$ + B$, "XYZ"
220 WRITE #2, B$; "ANOTHER STRING"; "END FILEB"
230 WRITE A; A + A; -A; 76; "FINISH"
999 END
```

Execution of this program will produce the following output at the user terminal:

```
 SAMPLE OF WRITE STATEMENT,
36 , 72 ,-36 , 76 ,FINISH,
```

The important thing to notice about the output is that each data item is followed by a comma delimiter and that no line numbers are generated when the WRITE statements at lines 130 and 230 are executed. Also, a SCRATCH statement is not necessary to WRITE to the terminal.

After the program has been executed, FILEA contains:

```
100  1 ,            22 ,          333 ,          4444 ,
110  36 ,            6 ,
120  36 ,-36 , 40 ,-47 , 99.9 ,
```

And FILEB contains:

```
100 STRING ITEM,   EXTRA LONG STRING,
110 ABCABC,        XYZ,
120 ABC,ANOTHER STRING,END FILEB,
```

All line numbers are separated from the first data on the line by one blank space. The line numbers are not counted as part of the 75 characters of data that are printed on all output lines. Nor are the line numbers and following blank included when counting character positions in using the TAB function.

One restriction is placed on the use of the WRITE statement. Once a PRINT statement has been executed to a file, WRITE statements can no longer be used with that file.

Example:

```
100 FILES A
110 SCRATCH #1
120 WRITE #1, 10, 11
130 PRINT #1, 12, 13
140 WRITE #1, 14, 15
999 END
```

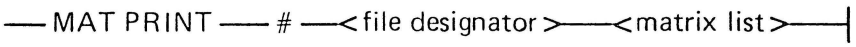
If the above program were executed, line 140 would cause the program to be aborted and the error message ILLEGAL I/O COMMAND to be displayed at the user terminal.

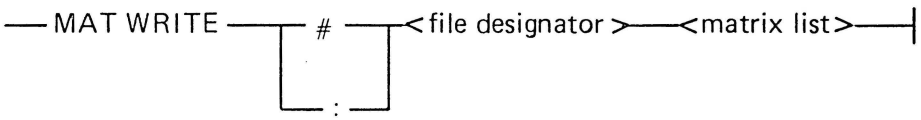
Files similar to those produced by the WRITE statement also can be created through CANDE at the user terminal. To do this the user should type his file CSEQDATA. This produces character mode files, and the sequence numbers (which are used when creating the file) are stored in the first five character positions, with the sixth position blank. To be compatible with files produced by the WRITE statement, string data is not enclosed in quotation marks, and a comma should separate each data item in the file.

FILE MAT PRINT AND MAT WRITE STATEMENTS

<FILE MAT PRINT statement>

Syntax





Semantics

These statements are similiar to the usual MAT PRINT and MAT WRITE statements. It is illegal to use MAT PRINT with binary files.

Example:

```
100 FILES WBDT
200 SCRATCH #1
300 DIM A(2,2), B(3,3), C(1,2)
400 MAT INPUT A, B, C
500 MAT PRINT #1, A, B
600 MAT PRINT #1, C;
700 END
```

If the input was 1,1,1,1,2,2,2,2,2,2,2,2,2,3,3 then a listing of file WBDT would look like this:

```
100  1              1
200  1              1
300  2              2              2
400  2              2              2
500  2              2              2
600  3  3
```

If the MAT PRINT in lines 500 and 600 was changed to MAT WRITE, then a listing of file WBDT would look like this:

```
100  1 ,             1 ,
200  1 ,             1 ,
300  2 ,             2 ,            2 ,
400  2 ,             2 ,            2 ,
500  2 ,             2 ,            2 ,
600  3 , 3 ,
```

The line numbers that appear are supplied by CANDE but do not actually appear in the files.

It is illegal to have MAT PRINT and MAT WRITE statements directed to the same file.

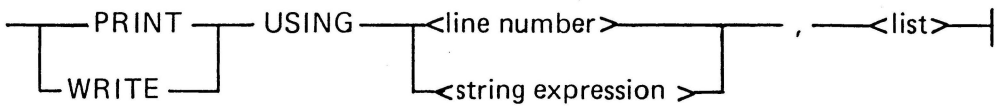
Illegal example:

```
100  FILES WBDT
200  SCRATCH #1
300  DIM A(2,2), B(3,3), C(1,2)
400  MAT INPUT A, B, C
500  MAT WRITE #1, A, B
600  MAT PRINT #1, C
```

FILE PRINT USING AND WRITE USING STATEMENTS

<FILE PRINT USING and WRITE USING statements>

Syntax



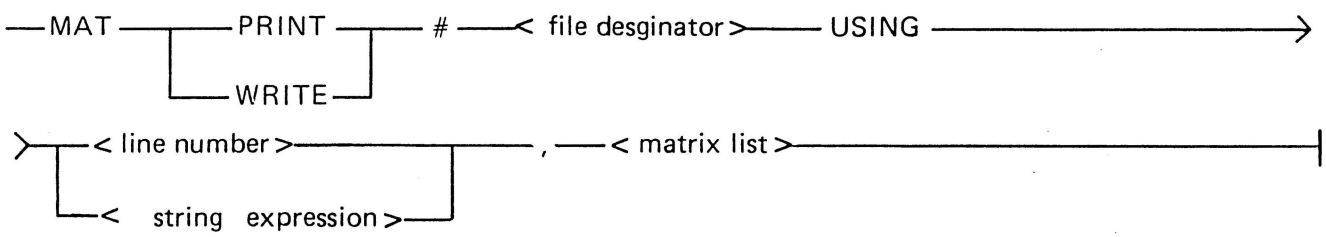
Semantics

BASIC enables the user to format the output from a BASIC program. The discussion which follows concerns the PRINT statement; however, the WRITE statement is identical, except for the use of the keywords PRINT and WRITE. If the <file designator> has a value of zero or if it is not present, the output will be returned to the user terminal. The <list> is the same as for the normal PRINT (and WRITE) statement. The data elements in the <list> must be separated by commas or semicolons, but these delimiters have no effect on the spacing between data items on the output line. All horizontal spacing is controlled by the image (format) associated with the PRINT USING statement. A PRINT USING statement which is terminated with a comma or semicolon will cause the system to hold the current line of output. The next PRINT to that file will begin printing data where the last PRINT USING stopped.

When the <line number> is used, it must be the number of a line in the BASIC program which contains the image to be used with the data items in the print list. When the single asterisk is used, the characters of the <string expression> are used as the image to be associated with the data items in the print list. It is illegal to use the PRINT USING and WRITE USING statements with binary files.

FILE MAT PRINT USING AND MAT WRITE USING STATEMENTS

<FILE MAT PRINT USING and MAT WRITE USING statements>



Matrices also may be printed using formatted output to a file. The MAT WRITE USING statement functions similiar to the MAT PRINT USING statement except that the written data is followed by a comma.

Example:

```
100 FILES TEMP4
200 SCRATCH #1
300 MAT A=IDN(3,3)
400 :###.# ####  #.####
500 MAT WRITE #1 USING 400,A
600 END
```

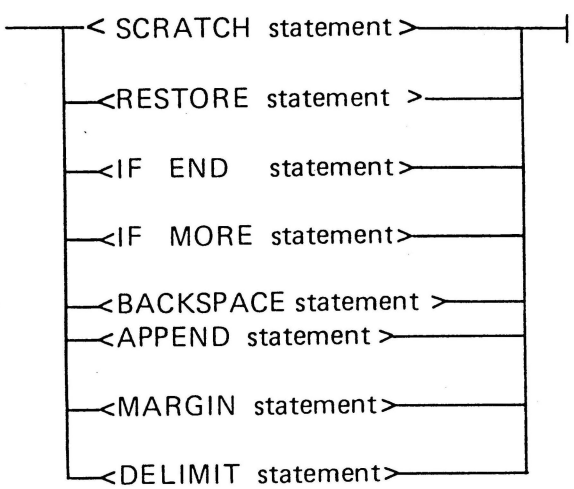
File TEMP4 contains:

```
100   1.0,    0,    .0000,
110    .0,    1,    .0000,
120    .0,    0,  1.0000,
```

FILE MANIPULATION STATEMENTS

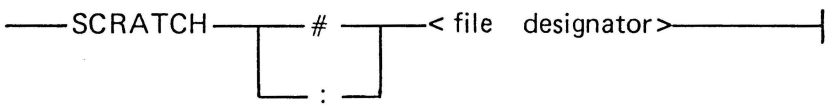
Syntax

<file manipulation statements>



SCRATCH STATEMENT

<SCRATCH statement>



Semantics

The SCRATCH statement allows the user to explicitly purge an external file. The <file designator> must be supplied and it cannot be a reference to file zero (the user terminal). The SCRATCH statement positions the data pointer to the designated file at the beginning of the file. If the file is a sequential file, all information in the file will be erased and the file will be placed in the WRITE mode. For random files the SCRATCH statement will cause all data elements in the file to be replaced with the value binary zero.

Examples:

```
100 FILES FILEA
110 WRITE #1, 10, 20, 30
120 RESTORE #1
130 READ #1, X, Y, Z
140 PRINT X; Y; Z;
999 END
```

When executed, the above program will be aborted with the error ILLEGAL I/O COMMAND, because line 110 attempts to write on FILEA while it was in the READ mode. By inserting the line

```
105 SCRATCH #1
```

The program will run and produce the following output at the user's terminal:

```
10  20  30
```

```
100 FILES FILEA
110 SCRATCH #1
120 WRITE #1, 10, 10, 10
130 WRITE #1, 20, 20, 20
140 WRITE #1, 30, 30, 30
150 RESTORE #1
160 READ #1, X, Y, Z
170 IF END #1 THEN 200
180 PRINT X, Y, Z
190 GOTO 160
200 END
```

When executed, the following will be output:

```
10      10      10
20      20      20
30      30      30
```

```
100 FILES FILEA
110 SCRATCH #1
120 WRITE #1, 40, 40, 40
130 RESTORE #1
140 READ #1, X, Y, Z
150 IF END #1 THEN 180
160 PRINT X, Y, Z
170 GOTO 140
180 END
```

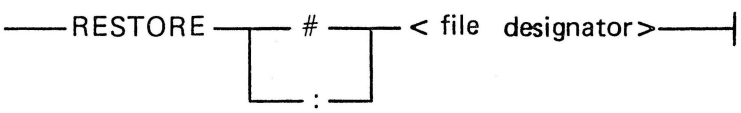
When executed, the following will be output:

```
40      40      40
```

RESTORE STATEMENT

Syntax

<RESTORE statement>



Semantics

This statement moves the file pointer so the next line uses the beginning of the file. The RESTORE statement also puts the file into the READ mode. If the file is at its beginning when the RESTORE statement is executed, the statement will be ignored and the program will continue in sequence. Files are automatically restored before the start of a program.

Example:

```
100 FILES ONE
110 SCRATCH #1
120 FOR I = 1 TO 10
130 WRITE #1, I, I*I
140 NEXT I
150 RESTORE #1
160 READ #1, A, B
170 PRINT A, B
180 END
```

A file is in the READ mode when it is first referenced in a program. To write to the file, it must be placed in the WRITE mode. This can be done by using a SCRATCH statement.

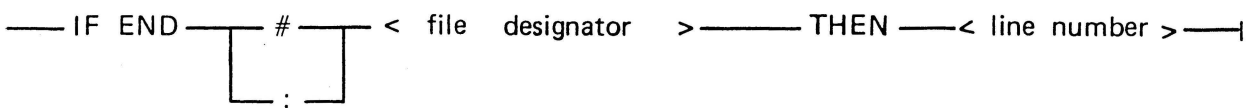
After a file has been written, it must be placed in the READ mode. The RESTORE statement in line 150 accomplishes this. The first data item in the file can now be read.

The WRITE statement places an EOF on the file, so all data previously on the file is lost as soon as the file is written. To make modifications to an existing file, the file must be copied onto another file up to the modification point. The modification can then be written onto the second file.

IF END STATEMENT

Syntax

<IF END statement>



Semantics

The IF END statement allows the user to check an external file to see if the data has been exhausted, when READ and INPUT statements are being executed. The <file designator> cannot be a reference to file zero (the user terminal). The <line number> is the line number of a statement in the program to which program control is transferred if the designated file is out of data.

The IF END statement tests whether or not a file is out of data by determining if the last data read was a valid data element. For this reason it is best to use the IF END statement after the READ or INPUT statement and before using the input data in the program.

Example:

```
100 FILES ONE
110 SCRATCH #1
120 WRITE #1, 1, 2, 3
130 WRITE #1, 4, 5, 6
140 RESTORE #1
150 READ #1, A, B, C
160 IF END #1 THEN 999
170 PRINT A; B; C
180 READ #1, A, B, C
190 IF END #1 THEN 999
200 PRINT A; B; C
210 READ #1, A, B, C
220 IF END #1 THEN 999
230 PRINT A; B; C
240 PRINT "THIS LINE SHOULD NEVER BE PRINTED"
999 END
```

Now file ONE contains the following data:

```
100  1 ,            2 ,            3,
110  4 ,            5 ,            6,
```

When executed, the program will produce the following output at the user terminal:

```
1  2  3
4  5  6
```

The READ statement at line 180 reads the last data in the file. The IF END statement at line 220 determined that the data read by line 210 was not valid (already exhausted) and program control was transferred to line 999, the end of the program. If lines 220 and 230 were interchanged in the above example, then the program would produce the following output:

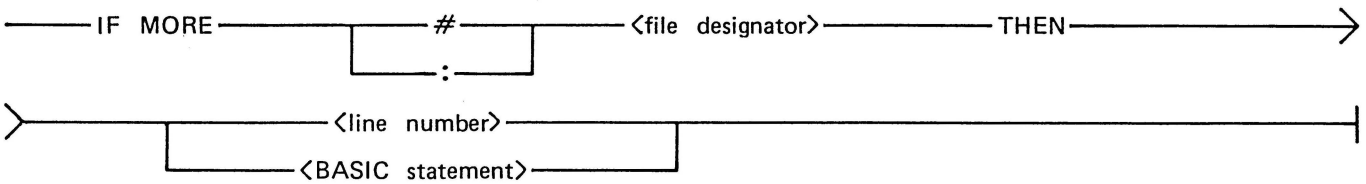
```
1  2  3
4  5  6
0  0  0
```

The line of zeroes was produced because the system assigns the value of zero to numeric variables and the value of null string to string variables when the data list is exhausted. Thus, line 210 assigned A, B, and C the value of zero and line 220 printed these values. The IF END statement, now at line 230, transferred control to line 999, thus avoiding the execution of line 240.

IF MORE STATEMENT

Syntax

<IF MORE statement>



Semantics

The IF MORE statement allows the user to check an external file to see if there is more data in the file. It is usually used with the READ and INPUT statements. The <file designator> cannot be a reference to file zero (the user terminal). The <line number> is the line number of a statement in the program to which program control will be transferred if there is more data in the designated file.

Examples:

```
100 FILES ONE
110 SCRATCH #1
120 WRITE #1, 1, 2, 3
130 WRITE #1, 4, 5, 6
140 RESTORE #1
150 READ #1, A, B, C
160 PRINT A; B; C
170 IF MORE #1 THEN 190
180 GO TO 999
190 READ #1, A, B, C
200 PRINT A; B; C
210 IF MORE #1 THEN 230
220 GO TO 999
230 READ #1, A, B, C
240 PRINT A; B; C
999 END
```

Now file ONE contains:

```
100  1 ,          2 ,          3
110  4 ,          5 ,          6
```

Then the program above produces the following output:

```
1  2  3
4  5  6
```

```
100  FILES **
200  FILE #1, "F"
300  SCRATCH #1
400  WRITE #1,100,200,300,400,500,600,700
500  WRITE #1,800,900,1000,1100,1200,1300,1400
600  RESTORE #1
700  READ #1, A, B, C
800  PRINT A, B, C
900  IF MORE #1 THEN 1100
1000 GOTO 1700
1100 READ #1, A, B, C
1200 PRINT A, B, C
1300 IF MORE #1 THEN 1500
1400 GO TO 1700
1500 READ #1, A, B, C
1600 PRINT A, B, C
1700 END
```

Now File F contains:

```
100  100,          200,          300,          400,          500,
110  600,          700,
120  800,          900,          1000,         1100,         1200,
130  1300,         1400,
```

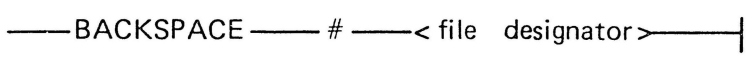
When executed, the above program outputs:

```
100          200          300
400          500          600
700          800          900
```

BACKSPACE STATEMENT

Syntax

<BACKSPACE statement>



Semantics

The BACKSPACE statement provides the user with the ability to move the data pointer backwards through the file. It also places the file in the READ mode. The BACKSPACE statement is valid only for external EBCDIC files.

The BACKSPACE statement depends on the last input or output statement which was executed to the file before it. If it was a READ or WRITE statement, then each BACKSPACE statement moves the pointer back to the last delimiter and the line number, if present, to the previous data element. If the BACKSPACE statement causes the pointer to be moved back past the beginning of the file, the first line of the file will be reused.

Example:

```
100 FILES BKSPR
110 READ #1, A, B, C, D, E
120 PRINT A; B; C; D; E
130 BACKSPACE #1
140 READ #1, A, B, C, D, E
145 PRINT A; B; C; D; E
150 FOR I = 1 TO 3
160 BACKSPACE #1
170 NEXT I
180 READ #1, A, B, C, D, E
190 PRINT A; B; C; D; E
200 FOR I = 1 TO 11
210 BACKSPACE #1
220 NEXT I
230 REM AT THIS POINT THE DATA POINTER IS AT
240 REM THE FIRST VALUE, 1.  ONE MORE BACKSPACE
250 REM WILL CAUSE THE FIRST ROW TO BE REUSED.
260 BACKSPACE #1
270 READ #1, A, B, C, D, E
280 PRINT A; B; C; D; E
999 END
```

If file BKSPR contains the following data:

```
100  1,  2,  3,  4,  5,  6,  7
110  11,  22,  33,  44,  55,
```

The example program will produce the following output:

```
1  2  3  4  5
5  6  7  11  22
7  11  22  33  44
7  11  22  33  44
```

Examination of the program and output should reveal how the BACKSPACE statement works when READ statements have been executed prior to the BACKSPACE statement. (It works the same for WRITE statements.) At line 260, the pointer is at the first line and goes back over this line from right to left. The pointer finds the last delimiter which is the comma following the 6, and so rests at the 7. Line 270 causes the 7 and the following four data items to be read.

When a file PRINT or INPUT statement was the last I/O statement executed, then the BACKSPACE statement causes the pointer to be moved back to the beginning of the current line. If the pointer is at the beginning of a line, then the BACKSPACE statement moves the pointer back one line and positions the pointer at the beginning of that line. If the BACKSPACE statement causes the pointer to be moved back past the beginning of the file, the first line of the file will be reused.

Example:

```
100 FILES BKSPI
110 INPUT #1, A, B, C, D
120 PRINT A; B; C; D
130 INPUT #1, A, B, C, D
140 PRINT A; B; C; D
150 BACKSPACE #1
160 BACKSPACE #1
170 INPUT #1, A, B, C, D
180 PRINT A; B; C; D
190 BACKSPACE #1
200 REM AT THIS POINT THE DATA POINTER IS AT
210 REM THE FIRST LINE OF THE FILE.  ONE MORE BACKSPACE
220 REM WILL CAUSE THE FIRST ROW TO BE REUSED.
230 BACKSPACE #1
240 INPUT #1, A, B, C, D
250 PRINT A; B; C; D
999 END
```

If file BKSPI contains the following data:

```
  1,  2,  3,  4,  5,  6
 11, 22, 33, 44, 55, 66
 77, 88, 99
```

Then the execution of the example program would produce the following output:

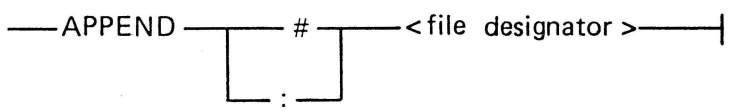
```
 1  2  3  4
 11  22  33  44
 1  2  3  4
 1  2  3  4
```

A careful study of the example will reveal how the BACKSPACE statement works when file INPUT and PRINT statements have been executed, prior to the execution of a BACKSPACE statement. Remember that after the execution of an INPUT statement, the data pointer is positioned at the beginning of the next line of the file. Especially note lines 200 through 250 which demonstrate backspacing past the beginning of the file.

APPEND STATEMENT

Syntax

<APPEND statement>



Semantics

The APPEND statement should allow the user to write information to an existing file without losing data already in the file. The APPEND statement should position the pointer right behind the last data item in the file and place the file in WRITE mode. The APPEND statement was never fully implemented at Burroughs. Only records within the last block of a file can be added in. If a program attempts to add records past the last block, an end-of-file condition may occur. Also, added data may not appear on the file if the file was previously opened input. The designated file must be an existing sequential file. (There must be at least one data item in the file.)

Example:

```
100 FILES APPEND
110 APPEND #1
120 WRITE #1, 44; 55; 66
130 WRITE #1, 21; 32; 43
999 END
```

If before the program was executed, the file APPEND contained:

```
100  1, 2, 3, 4
110  5, 6, 7, 8
```

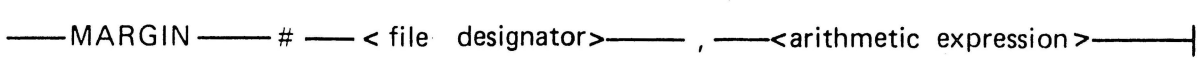
then after the execution of the example program file APPEND would contain:

```
100  1, 2, 3, 4
110  5, 6, 7, 8
120  44 , 55 , 66 ,
130  21 , 32 , 43 ,
```

MARGIN STATEMENT

Syntax

<MARGIN statement>



Semantics

The MARGIN statement allows the user to specify the character position to be used as the right margin on external EBCDIC files only. The left margin is always defined to be character position zero. A <file designator> must be specified, and it can have a value of zero (the user terminal). The <arithmetic expression> specifies the character position to be used as the right margin. If the value of the <arithmetic expression> is not an integer, it will be truncated to an integer.

If no margin value is explicitly set for a file with a MARGIN statement, the default value used by the system is 75. When WRITE statements are executed to a file, the line number and following blank space are not included as part of the margin specification. (That is, a margin of 75 means there are really 81 characters on a line of output produced by the WRITE statement.) If the user desires a margin greater than 75, he must specify the maximum margin value in a MARGIN statement before the execution of any I/O statements on that file. Then, once an I/O statement has been executed on the file, the margin may be varied to any size less than or equal to this maximum value.

If the margin has been explicitly set, the execution of a SCRATCH statement on that file will not reset the margin to its default value. If the file is explicitly closed and later reopened, the value of the margin will be the default value when the file is reopened.

Example:

```
100 FILES A
110 MARGIN #1, 18
120 SCRATCH #1
130 WRITE #1, 1; 2; 3; 4; 5; 6; 7; 8; 9;
140 WRITE #1, 10; 11; 12
150 FOR I=1 TO 12
160 WRITE #1, "Z";
170 NEXT I
180 MARGIN #0, 20
190 PRINT #0, 1; 2; 3; 4; 5; 6; 7; 8; 9;
200 PRINT #0, 10; 11; 12
210 FOR I = 1 TO 25
220 PRINT #0, "Z";
230 NEXT I
999 END
```

When this program is executed, the following output will be produced:

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

Notice that the letter Z was written 19 times on one line. When string data is being output, the whole line of output is filled. However, when numeric data is being output a check is made before printing the number to see if the number and one blank space can fit on the line of output. If they cannot, they are printed on a new line. This is why 7 was printed on the second line of output though there was room for it on the first line. Thus, if the margin is set to N, there are effectively only N-1 character positions on the line if the last item which could be written on the line is a numeric value.

After execution of the above program, file A contains:

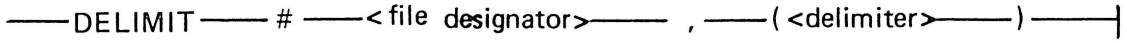
```
100   1 , 2 , 3 , 4 ,
110   5 , 6 , 7 , 8 ,
120   9 , 10 , 11 , 12 ,
130  Z,Z,Z,Z,Z,Z,Z,Z,Z,
140  Z,Z,Z,
```

Observe that the MARGIN statement has significance only for output statements. Input statements do not need to know the value of the margin since they will read data from the input line until there is no more data on the line, whether there was only one character on the line, or 255 characters.

DELIMIT STATEMENT

Syntax

<DELIMIT statement>



Semantics

The DELIMIT statement allows the user to specify a delimiter, other than the standard delimiter (the comma), to be used between data items which are written to external EBCDIC files only. The <file designator> must be present and may reference files zero (the user terminal).

If a file has been created with a delimiter other than a comma, that delimiter must be specified in a DELIMIT statement before any READ or INPUT statements to that file are used. If a file is to be written with a delimiter other than the comma, that delimiter must be specified in a DELIMIT statement before any WRITE statement to that file is used. The DELIMIT statement has no effect on the execution of PRINT statements since PRINT statements do not place delimiters between data on the output line.

Example:

```
100  FILES FILE1
200  DELIMIT #1, (%)
300  SCRATCH #1
400  WRITE #1, 1; 2; 3; 4; 5
500  END
```

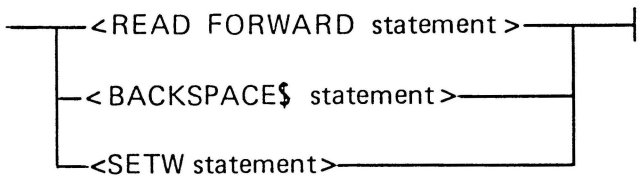
After execution, FILE1 will contain:

```
100  1%  2%  3%  4%  5%
```

MANIPULATION OF BINARY FILES

Syntax

<binary file statements>



Semantics

The user has the option to use binary external files. Binary files can be accessed randomly as well as sequentially. Also, binary files do not require code conversion as do EBCDIC files. The following paragraphs explain binary files, how they are created, and how they are used.

Storage of Data on Binary Files

For the user to make the most efficient use of binary files, it is necessary to understand how data is actually stored on these files.

In a binary file, each numeric data item is stored in one word of the binary file. (This means the Large Systems word of 48 information bits and 3 tag bits.) A number is stored in a word in its binary representation. If 100 numeric data items are written to a binary file, then there are 100 words of valid information in the file. If the user were randomly accessing the file and he wanted to read the 61st data item in the file, he would set the value of the data pointer to 61 (with a SETW statement) and then read one word from the file.

String data, on the other hand, is stored differently. String data may contain 0 to 255 characters so it is not possible to store this data as one word per string item. When a string data item is written to a binary file, a STRING CONTROL WORD is written preceding the actual characters of the string. A string control word contains the number of characters in the string. Following the string control word is the actual string of characters, stored as a binary representation of EBCDIC character codes. The string is stored six characters per word of the file. If the last characters of a string do not fill a complete word, the remainder of the last word is filled with blanks. For instance, the string "WHERE THERE IS BUSINESS, THERE IS BURROUGHS" would occupy nine words in a binary file. The first word is the string control word indicating that there are 43 characters in the string. The next seven words, two through eight, contain the first 42 characters of the string, six characters per word. And the ninth word contains the 43rd character, "S", and five blank spaces.

Processing a binary sequential file is basically no different than processing an EBCDIC sequential file. Since binary files of any kind cannot be created through CANDE, the user must create his binary files with a program. If the user reads the data from the file in the same sequence that he wrote the data to the file he will have no problems.

Processing binary random files requires the user to know how the data is stored in the file. This is because the user can gain access to any word in the file. If he does not understand how the data has been stored, he may find himself reading six characters of a string when he expects to be reading a numeric value.

CREATING BINARY FILES

Making either binary sequential or binary random files was discussed earlier. The following paragraphs will mention that subject in an occasional explanation of some examples.

INPUT AND OUTPUT STATEMENTS FOR BINARY FILES

The following input and output is available for binary files.

READ and WRITE statements

The READ statement is used to read values from binary files and load those values into associated variables.

The WRITE statement is used to place values in binary files.

Examples:

```
100 FILES BIN
110 SCRATCH :1
120 A = B = C = 15
130 WRITE :1, A, B*2, C*3
140 A$ = "STRING ITEM"
150 WRITE :1, A$, A$ + " " + A$
160 RESTORE :1
170 READ :1, X, Y, Z
180 PRINT X; Y; Z
190 READ :1, X$, Y$
200 PRINT X$; Y$
999 END
```

In the example above, file BIN is opened by its appearance in the FILES statement at line 100. It is a binary file because it is referenced with a <:>. Since no length has been specified for the file, it is understood to be sequential. When executed, the program will produce the following output:

```
   15  30  45
 STRING ITEMSTRING ITEM STRING ITEM
```

The program:

```
100 FILES F
200 SCRATCH :1
300 FOR I = 1 TO 50
400 WRITE :1, I
500 NEXT I
600 END
```

Creates a File F, and then the program:

```
100 FILES F
200 FOR I = 1 TO 50
300 READ :1, J
400 PRINT J;
500 NEXT I
600 END
```

Outputs File F in three lines: 1-21, 22-40, and 41-50.

The program:

```
100 FILES F1
200 SCRATCH :1
250 A$ = "ABCDEF"
300 FOR I = 1 TO 10
400 WRITE :1, A$
500 NEXT I
550 FILE :1, "*"
600 END
```

Creates a File F1 and then the program:

```
100 FILES F1
200 FOR I = 1 TO 10
300 READ :1, A$
400 PRINT I, A$
500 NEXT I
600 END
```

Outputs File F1 as follows:

```
1             ABCDEF
2             ABCDEF
3             ABCDEF
4             ABCDEF
5             ABCDEF
6             ABCDEF
7             ABCDEF
8             ABCDEF
9             ABCDEF
10            ABCDEF
```

MAT READ and MAT WRITE statements for binary files are the same form as for EBCDIC files, except that the <#> is replaced by a <:>.

Example:

```
100  FILES WBDT
200  SCRATCH :1
300  DIM A(3,4)
400  MAT INPUT A
500  MAT WRITE :1, A
600  RESTORE :1
700  MAT READ :1, A
800  MAT PRINT A
900  END
```

When the above program is run with the input 1,2,3,4,5,6,7,8,9,10,11, and 12, the following is output:

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

READ FORWARD STATEMENT

Syntax

<READ FORWARD statement>

Semantics

The READ FORWARD statement allows the user to read selectively from a binary file. The variable can be numeric or string.

When a READ FORWARD statement encounters a numeric variable in its associated list, it will skip over string data in the file and input the next numeric item from the file. If there is no numeric data remaining in the file, the READ FORWARD statement will stop at the end-of-file and the numeric variable in the list will be assigned the value zero.

When the READ FORWARD statement encounters a string variable in its associated list, it will skip over numeric data in the file and input the next string from the file. If there is no string data remaining in the file, the READ FORWARD statement will stop at the end-of-file and the string variable in the list will be assigned the value of null string.

Example:

```
100 FILES BIN
110 SCRATCH :1
120 WRITE :1, "STRING A", 1, 2, "STRING B"
130 RESTORE :1
140 READ FORWARD :1, X, A$
150 PRINT X; A$
999 END
```

When executed, the program prints:

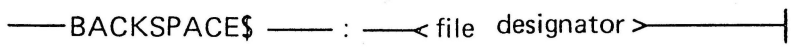
```
1 STRING B
```

When the READ FORWARD statement at line 140 was executed, it skipped over STRING A and assigned X the value 1. It then skipped over 2 and assigned A$ the value STRING B.

BACKSPACE$ STATEMENT

Syntax

<BACKSPACE$ statement>



Semantics

The BACKSPACE$ statement is used to move the data pointer in a binary file back to the last string control word in the file. If no string control word is found, the data pointer is left pointing to the beginning of the file. For binary sequential files the BACKSPACE$ statement places the file in the READ mode.

Example:

```
100 FILES BIN
110 SCRATCH :1
120 WRITE :1, "STRING A", 1, 2, "STRING B"
130 WRITE :1, 3, 4, "STRING C"
135 RESTORE :1
140 READ FORWARD :1, A$, B$, C$
150 PRINT A$; B$; C$
160 BACKSPACE$ :1
170 BACKSPACE$ :1
180 READ :1, X$
190 PRINT X$
999 END
```

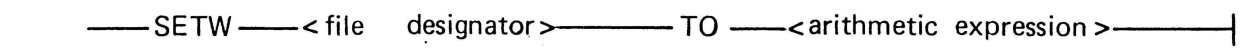
When executed, this program will produce the following output:

```
STRING ASTRING BSTRING C
STRING B
```

SETW STATEMENT

Syntax

<SETW statement>



Semantics

The SETW statement allows the user to move the data pointer, in a binary random file, to a specific word in the file. The value of the element following the keyword TO, is the position of the word in the binary random file to which the data pointer will be set. If this value is not an integer, it will be truncated to an integer. If this value is less than one, or greater than the number of words in the binary random file, the execution of the program is aborted and the error message INVALID SETW ARGUMENT will be displayed at the user terminal.

The SETW statement also is used to classify an already existing binary file as a random access file. For instance, if a BASIC program is using a binary file created by some other BASIC program, all that is necessary to use it as a random access file is to execute a SETW statement on that file prior to any read or write operations. The SETW statement will cause the mode restrictions to be lifted and hence the file will be treated as a random file. Note: When using an already existing binary file in another BASIC program, the user should not try to classify it in the second program as a random access file by using a FILE statement of the form FILE :1, BINARY, 300 as this would cause the 300 words in the existing file to be zeroed out.

Example:

```
100 FILES BINRAN
110 FILE :1, 100
120 SETW 1 TO 1
130 WRITE :1, 10, 20, 30, 40, 50
140 FOR X = 5 TO 1 STEP -1
150 SETW 1 TO X
160 READ :1, V
170 PRINT V
180 NEXT X
190 PRINT "PROGRAM COMPLETED"
999 END
```

The above example creates the file BINRAN, which is classified as a random access binary file when line 110 specifies the file to be 100 words long (each word set to zero). Line 120 positions the data pointer to this file at the first word. Notice that a SCRATCH

statement was not necessary before line 130 wrote to the file, since there are no mode restrictions on random access files. The loop in lines 140 through 180 then uses the SETW statement to read the file backwards and produce the following output:

```
50
40
30
20
10
PROGRAM COMPLETED
```

The handling of random binary files is not difficult if the program uses only numeric data. String data is more complicated. One way to handle string data on random files is to calculate how many words of the file the longest string will require and then set the data pointer ahead that many words after each string is written. In the following example the longest string is "ABCDEFGHIJKLMNOPQRSTUVWXYZ", which is 26 characters long. This string will require six words in the binary file. (One for the control word, four for the first 24 characters, and one for the final two characters). After each string is written, the data pointer will be moved ahead six words.

Examples:

```
100 FILES *
110 FILE :1, "A", 100
120 X = 1
130 WRITE :1, "ABCD"
140 X = X + 6
150 SETW 1 TO X
160 WRITE :1, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
170 X = X + 6
180 SETW 1 TO X
190 WRITE :1, "ABCDEFGHIJKL"
200 X = X + 6
210 SETW 1 TO X
220 WRITE :1, "THE LAST STRING"
230 PRINT "THE STRINGS IN REVERSE ORDER ARE"
240 FOR W = 4 TO 1 STEP -1
250 SETW 1 TO FNW(W)
260 READ :1, X$
270 PRINT X$
280 NEXT W
290 DEF FNW(X) = ((X - 1) *6) +1
999 END
```

This example will produce the following output:

```
THE STRINGS IN REVERSE ORDER ARE
THE LAST STRING
ABCDEFGHIJKL
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCD
```

The following program creates a file:

```
100 FILES *
200 FILE :1, "F", 200
300 SETW 1 TO 1
400 FOR I = 1 TO 200
500 WRITE :1, I
600 NEXT I
700 FILE :1, "*"
800 END
```

The following program writes to and then reads from the file:

```
100  FILES F
200  SETW 1 TO 1
300  FOR I = 1 TO 70
400  WRITE :1, 1000
500  NEXT I
600  FOR I = 1 TO 20
700  READ :1, J
800  NEXT I
900  FILE :1, "*"
1000 END
```

The following program prints the file:

```
100 FILES *
200 FILE :1, "F"
300 FOR I = 1 TO 200
400 READ :1, A
500 PRINT A;
600 NEXT I
700 END
```

SUMMARY OF STATEMENTS AVAILABLE TO EXTERNAL FILES

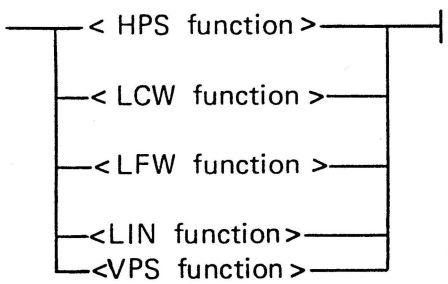
The following list describes the availability of BASIC statements for use with external EBCDIC sequential files, external binary sequential files, and external binary random access files. An X indicates that the statement is valid for that file.

STATEMENT	EBCDIC	BIN SEQ	BIN RAN
PRINT	X		
PRINT USING	X		
MAT PRINT	X		
MAT PRINT USING	X		
WRITE	X	X	X
WRITE USING	X		
MAT WRITE	X	X	X
MAT WRITE USING	X		
INPUT	X		
MAT INPUT	X		
READ	X	X	X
MAT READ	X	X	X
READ FORWARD		X	X
SCRATCH	X	X	X
RESTORE	X	X	X
IF END	X	X	X
IF MORE	X	X	X
DELIMIT	X		
MARGIN	X		
BACKSPACE	X		
APPEND	X	X	
BACKSPACE$		X	X
SETW			X

FILE FUNCTIONS

Syntax

<FILE functions>



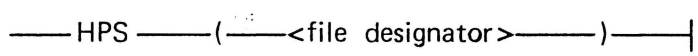
Semantics

BASIC provides the following file functions.

HPS FUNCTION

Syntax

<HPS function>



Semantics

The HPS function returns the value of the character position in the current line of the EBCDIC file which is being read from or written to.

Example:

```
100 MARGIN #0, 25
110 FOR X=1 TO 12
120 PRINT X;
130 NEXT X
140 X = HPS(0)
150 PRINT "CHARACTER POSITION IS";X
999 END
```

The above program will produce the following output:

```
1  2  3  4  5  6  7  8
9  10  11  12
CHARACTER POSITION IS 15
```

The HPS function in line 140 refers to file zero, the user terminal. The character position is 15 because the line starts at 0. "9" is in the zero position; "10" is in the 3-4 position; "11 is in the 7-8 position; and "12" is in the 11-12 position. Allowing 2 spaces (13-14), the pointer will now be at position 15. If the semi-colon was removed from the end of line 120, the program would have returned a character position of 0 because it would print at the start of a new line.

This function is valid only for EBCDIC files.

LCW FUNCTION

Syntax

<LCW function>

Semantics

The LCW function returns the value of the current location of the data pointer in a binary random access or sequential file.

Example:

```
100 FILES BINRAN
110 FILE :1,100
120 SETW 1 TO 67
130 PRINT "DATA POINTER IS AT WORD"; LCW(1)
140 SETW 1 TO (35+3)
150 PRINT "DATA POINTER IS AT WORD"; LCW(1)
999 END
```

When executed, this program will produce the following output:

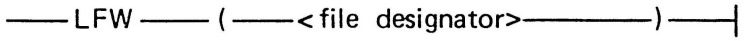
```
DATA POINTER IS AT WORD 67
DATA POINTER IS AT WORD 38
```

This function is valid on binary files only.

LFW FUNCTION

Syntax

<LFW function>



Semantics

The LFW function returns the value, in words, of the current size of a binary random access file. This size is established by the FILE statement which specifies the length of the file.

Example:

```
100 FILES BINRAN
110 X = 71
120 FILE :1, (X * 2) + 4
130 PRINT "THE SIZE OF THE FILE IS"; LFW(1)
999 END
```

Which, when executed will produce the following output:

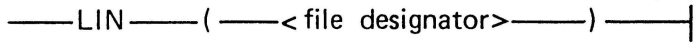
```
THE SIZE OF THE FILE IS 146
```

This function is valid on binary random access files only.

LIN FUNCTION

Syntax

<LIN function>



Semantics

The LIN function returns the value of the last line number encountered by the system when reading an EBCDIC sequential file, or the last line number created by the system when an EBCDIC sequential file is being written.

Example:

```
    100 FILES A
    110 SCRATCH #1
    120 FOR I=1 TO 15
    130 WRITE #1, I
    140 NEXT I
    150 PRINT LIN(1)
    999 END
```

When the above program is executed, the following is output:

```
    240
```

By issuing the CANDE command "LIST A" it can be seen that the last line number created for the file was indeed 240.

```
100  1 ,
110  2 ,
120  3 ,
130  4 ,
140  5 ,
150  6 ,
160  7 ,
170  8 ,
180  9 ,
190  10 ,
200  11 ,
210  12 ,
220  13 ,
230  14 ,
240  15 ,
```

VPS FUNCTION

Syntax

<VPS function>

Semantics

The VPS function returns the value of the number of lines which have been read from or written to an EBCDIC sequential file.

Example:

```
100 FILES A
110 SCRATCH #1
120 FOR I = 1 TO 15
130 WRITE #1, I
140 NEXT I
150 PRINT VPS(1)
999 END
```

When the above program is executed, the following is output:

```
15
```

APPENDIX A

COMPILER OPTIONS

The following compiler options are available in BASIC using the standard dollar sign form: $SET <option>

ANSI78 — When the ANSI78 compiler option is set, or a compiler built using the ANSI78COMP flag is used, the compiler sets a bit to tell the intrinsics that ANSI78 format is to be expected. The user program is compiled according to the ANSI 1978 Standard for minimal BASIC instead of the Burroughs implementation of BASIC. Also, the proper interface between BASIC and BASICSUPPORT is provided for the processing of user input as specified in the ANSI standard. $ RESET ANSI78 causes the user program to be compiled according to the Burroughs implementation. The ANSI78 option statement can appear only once and must precede all non-dollar statements. For more information see Appendix C.

CHECK — Sequence numbers on input are checked and errors listed.

CODE — The output listing will contain the object code generated by the compiler.

ERRLIST — All syntax error messages are issued to a print file or to a remote terminal.

LIMIT<integer> — The BASIC compiler will terminate compilation of a program if the number of syntax errors in the program equals or exceeds <integer>. If the LIMIT option is not specified the default value is 100 (10 if called from CANDE).

LINEINFO — A line directory is placed in the object code file of the program. The line directory is used to locate a BASIC statement (its 4-digit statement number found in character positions 1 through 4) for a given object code location.

LIST — Source language statements are printed on the line printer. If LIST is reset, only errors and error messages are printed.

MERGE — Primary input file named CARD is merged with secondary input file TAPE on the basis of sequence numbers. If sequence numbers match, the primary input overrides. If MERGE is reset, the secondary input is completely ignored.

NEW — A new source language file called NEWTAPE is created. This file may later be used as input to the BASIC compiler as the file tape.

PAGE — If the LIST option is set when compiling a BASIC program, then every occurrence of the option PAGE will cause the compiler to skip to the top of the next page of the output listing.

SEGMENT — Allows the user to control segmentation of the BASIC code file. If the user does not wish to explicitly control segmentation of his program, then the BASIC compiler automatically begins a new segment of code each time it has accumulated approximately 500 words of code in one segment. Five hundred words is the default size for code segments generated by the BASIC compiler; however, the user may change this size specification by using the $ SEGMENT = <integer> option.

Example:

```
100 $ SEGMENT = 256
200 LET P1=3, 1415927

  .
  .
  .
```

In the example, line 100 directs the compiler to create code segments consisting of approximately 256 words. When the compiler has generated 256 words of code for the program, then that segment (containing approximately 256 words) of code is written to disk, and a new segment begins. The program is thus divided into consecutive segments of about 256 words each. The size specification is an approximation because the compiler checks the size of the segment when it begins compiling each statement. Therefore, the exact size of each segment in a program may differ but they will all contain about the same number of words. The

<integer> following the equal sign in the SEGMENT option must be a positive integer in the range 1 through 8,196. Note that as the segment size decreases, the chances become greater that statements such as GO TO, GO SUB, ON <expression> GO TO, and IF <relational expression> THEN <line number> will be transferring control of the program control to another code segment which is slower than transferring control to a destination within the same segment.

If the segment size specification is exceeded while a multi-lined DEF function is being compiled, the compiler will not create a new segment before it starts to compile the next segment. Instead, the new segment will be started immediately after the compiler has found the FNEND statement for the function.

In addition to the ability to specify the size of code segments, the user may explicitly cause a new segment to be started by using the compiler control option SEGMENT (without the equal sign and integer). When the SEGMENT option is used in this manner it can be SET and POPed only. Any attempt to RESET this option will result in a compiler warning message and no action will be taken.

When the BASIC compiler encounters a $SET SEGMENT, it begins generating code for the next statement of the program in a new segment. If the compiler later encounters a $POP SEGMENT, it will end the current segment, write it to disk, and continue generating code at the nest statement to be placed in the code segment accumulated before the last $SET SEGMENT was encountered.

Example:

```
100 A=1
200 $ SET SEGMENT
300 FOR I = 1 TO 3
400     PRINT I
500 NEXT I
600 $POP SEGMENT
700 GO TO 900
800 INPUT I
900 END
```

In the example above, the code for statements 100 and 700-900 is written to one code segment and the code for statements 300-500 is written into another separate segment. The following example of a skeleton BASIC program illustrates how the SEGMENT option can be used to nest segments within other segments. The alphabetic characters that appear between the statements are used to show portions of the program that are stored in common code segments.

Example:

```
100 REM   SOME EXPLICIT SEGMENTATION
          .
          .
          A
          .
          .
200 $SET SEGMENT
                .
                .
                B
                .
                .
300 $SET SEGMENT
                   .
                   .
                   C
                   .
                   .
400 $POP SEGMENT
                .
                .
                B
                .
                .
500 $SET SEGMENT
                         .
                         .
                         D
                         .
                         .
600 $POP SEGMENT
                .
                .
                B
                .
                .
```

```
700 $POP SEGMENT
      .
      .
      A
      .
      .
800 END
```

When using the SEGMENT option in this way, segments may be nested 47 deep (that is, 47 consecutive $ SET SEGMENT statements without an intervening $ POP SEGMENT).

A $SET SEGMENT or $POP SEGMENT may not appear in the definition of a multi-line DEF function.

If a BASIC program contains a $SET segment option, and the compiler reaches the end of the program without encountering a corresponding $POP SEGMENT, then the compiler will implicitly POP the segment before generating any wrap up or stack building code.

When a program sets the size specification with a $SET SEGMENT = <integer>, or if the default segment size specification is used, then this size specification applies to code generated between $SET SEGMENT and $POP SEGMENT options.

SEQ — The listing or new file will contain new sequence numbers. The new sequence number will be the base, initially. Then the current sequence base is incremented by the sequence increment for a new sequence base. If the sequence base and increment are not specified, the value 1000 is used.

SEQERR — Sequence errors are flagged as compilation errors if CHECK is set.

SINGLE — Listing is single-spaced; if SINGLE is reset, listing is double-spaced.

STACK — The listing will contain relative stack addresses in the form of address couples for all program variables.

VOID — All input from TAPE and CARD is ignored except dollar cards, until VOID is reset.

VOIDT	All input from secondary input source is ignored except dollar cards, until VOIDT is reset.
XREF	A cross reference will be generated and is added to the print file. The LIST option need not be set.
$	Dollar cards appear on the listing.

All options are initially off, or reset, except LIST, and SINGLE.

Options can be SET, RESET, or POPed on the dollar card. SET caused the options listed to be turned on. RESET causes them to be turned off. POP restores options in the succeeding list to their prior values.

An option list with no option action causes all options which appear in the option list to be turned on and all other options to be turned off.

Several actions may be put on the same dollar card when the card reader is not being used. Each action may be followed by one or more options in a list separated by at least one space. A list with no option action may appear alone on a card or before an option action on the card.

Examples of valid dollar($) cards:

```
$ MERGE NEW SEQ 100+100
$ SET LIST VOIDT RESET NEW
$ POP VOID SET SEQ
$ NEW SET LIST SEQ MERGE
```

When the card reader is used, each card is taken as a different line and therefore must contain exactly one statement. The end-of-message character should not be used at the end of a statement, since it will be treated as an illegal operator. The INPUT statement causes data to be read from file INPUT, which is the card reader. The PRINT statement writes data on file PRINT, which is a printer back-up file on disk.

APPENDIX B

USING BASIC FROM A REMOTE TERMINAL

New users of Burroughs B 5000/B 6000/B 7000 Command and Edit (CANDE) Language who wish to know enough to write and run a BASIC program from a remote terminal should consult the CANDE Reference Manual. To properly use some of the features in BASIC, such as the data file capability, the user needs a more thorough knowledge of the system than can be found here.

The remote terminal is used by the programmer to type lines of information to the computer, either supplying it with data (such as a BASIC program) or issuing commands telling it what to do. Every line typed by the user must be ended with an end-of-message character. This signals the end of the line and causes the system to respond with a carriage return and a line feed, thus positioning the terminal at the beginning of a new line. The user should not perform either of these actions himself. The standard for end-of-message is defined in NDL.

RECORD SIZE

CANDE cannot write or update character mode files which have a MAXRECSIZE that is an odd number. Files created by PRINT and WRITE statements in a BASIC program are created as character mode files with a MAXRECSIZE of 81, and thus cannot be modified by CANDE. Therefore, if the user creates files with BASIC PRINT and WRITE statements, which he plans to manipulate through CANDE, he must create the files with an even number of characters for the record size. This can be done in the BASIC program by using the MARGIN statement.

Example:

```
MARGIN #1,82
```

For purposes of label equation, the INTNAME of internal file number zero is BCARD for input and BLINE for output.

APPENDIX C

THE ANSI 1978 STANDARD BASIC COMPILER OPTION

Several ways in which the ANSI78 compiler is different from the Burroughs implementation of BASIC are discussed in the following paragraphs.

Only one data list is formed instead of two. It consists of string and numeric constants in the order that they appeared in the DATA statements. (See DATA statements in section 4.)

A RANDOMIZE statement has been implemented which can be used only when the ANSI78 compiler option is set. When the RANDOMIZE statement is used, different random numbers are generated with each execution. If the statement is not included, the RND function will generate the same numbers for each execution.

The RESTORE statement is only valid in its simplest form, that is, RESTORE.

An unquoted null string is any keyboard symbol that does not begin with a quote. Blanks following an unquoted string are ignored. Trailing blanks in unquoted strings, read from either DATA statements, the terminal, or external files, are no longer retained as a part of the string.

Unquoted null strings (unquoted strings consisting of only blanks) are not permitted in DATA statements. They will cause a compilation error. This affects READ, INPUT, MAT READ, and MAT INPUT statements. For INPUT and MAT INPUT, a bit is set indicating if ANSI78 is set or not.

When the TAB function contains a real number as the argument, the number is rounded to an integer. (With ANSI78 Real, the number will have been truncated). Also, if the TAB is in a backwards direction, or would overwrite the current column, a line feed is generated followed by a tab to the specified column. The argument to the TAB must be positive. If the argument is greater than 75, it is taken modulo 75, with the exception that a multiple of 75 will cause a tab to position 75. If the print position designated by the TAB function already has been passed, the output will begin on the next line at the column specified by the TAB argument. If the value for the argument is less than 1, the TAB will be to position 1.

For zoned format, each line is numbered 1 to 75, where positions 1, 16, 31, 46 and 61 will mark the beginning of each print zone.

The entire input reply is validated before any values are assigned to the variables in the variable-list of the corresponding INPUT statement. There must be the same number of items input as there are in the variable-list of the corresponding INPUT statement. If any errors in the input-reply are detected, the user must resupply the input-reply. Errors that may occur include (1) too much or insufficient data in the input-list; (2) the type of data item does not match the type of the variable to which it is to be assigned; and (3) one or more input items are syntactically incorrect. If no data appears between two commas in the input reply, the data is considered to be null. In this case, string variables will be assigned the null string and numeric variables will be assigned zero. If a comma appears as the first item in the input-reply, then the first variable will be assigned its appropriate null value. If a comma appears at the end of the input-reply with one or more spaces before the end-of-input, then this will be considered a null input item, and the corresponding variable will be assigned its appropriate null value. If a comma appears at the end of the input reply with no spaces before the end-of-input, then the comma will be ignored.

Examples:

```
100 INPUT A, B, C
```

In the following input replies, "X" marks the end-of-input.

(1) The input reply:

```
,1,2X
```

will cause A to be assigned 0 (null),
B to be assigned 1, and
C to be assigned 2.

(2) The input reply:

```
1,2,X
```

will be considered to contain 2 items, since the last comma is ignored. An error message reporting that more input items are needed will be generated.

(3) The input reply:

```
1, ,2
```

will cause A to be assigned 1,
B to be assigned 0 (null), and
C to be assigned 2.

(4) The input reply:

```
1,2,   X
```

will cause A to be assigned 1,
B to be assigned 2, and
C to be assigned o (null).

In an ON statement, if the arithmetic expression is less than one or greater than the number of line numbers provided, then a run-time error is generated and execution is terminated. (With Burroughs implementation, control would have passed to the next statement). Trailing blanks in unquoted strings are not retained. This is true whether the string has been read from DATA statements, the terminal, or external files. In addition, DATA statements containing unquoted null strings, or unquoted strings consisting of only blanks will cause a compilation error to be generated.

APPENDIX D
ERROR MESSAGES

NUMBER	MEANING
1	STATEMENT NUMBER MISSING OR NOT BETWEEN 1 AND 9999
2	THIS IS NOT A RECOGNIZABLE STATEMENT STARTER
3	THIS IS AN ILLEGAL RELATIONAL OPERATOR
4	AN ARITHMETIC VARIABLE NAME WAS EXPECTED
5	<THEN> <GO TO> IS MISSING FROM 'THEN PART' IN IF STATEMENT
6	MISSING RIGHT PAREN ON A SUBSCRIPTED VARIABLE
7	COMPILER ERROR --> DURING SEGMENTATION
8	BAD GO TO - BRANCH CANNOT BE INTO OR OUT OF DEF FUNCTION
9	MISSING LEFT PAREN ON AN INTRINSIC CALL
10	A MISSING <TO> IN THE FOR STATEMENT
11	UNMATCHED 'FOR' STATEMENT(S) IGNORED
12	BAD GO TO - STATEMENT CANNOT BRANCH TO ITSELF
13	SEGMENTATION NOT ALLOWED IN DEF FUNCTIONS
14	
15	SEGMENT SIZE SPECIFICATION OUT OF RANGE (1 TO 8192)
16	FOR STATEMENTS ARE NESTED TOO DEEPLY
17	NESTING STACK OVERFLOW
18	A STATEMENT NUMBER WAS EXPECTED FOR THIS TRANSFER
19	UNMATCHED 'NEXT' STATEMENT IGNORED
20	NUMBER EXPECTED FOR SEQUENCE INCREMENT ON DOLLARCARD
21	UNRECOGNIZED DOLLAR CARD OPTION
22	SOURCE FILE HAS A SEQUENCE ERROR
23	NUMBER EXCEEDS ALLOWABLE RANGE
24	DEF FUNCTION NAME MUST BE A LETTER OR A LETTER FOLLOWED BY $
25	ASSIGNMENT OPERATOR MISSING
26	COMMA OR RIGHT PAREN EXPECTED ON SUBSCRIPTED VARIABLE
27	LEFT PAREN OR ASSIGNMENT OPERATOR EXPECTED
28	RIGHT PARENTHESIS EXPECTED FOR INTRINSIC CALL
29	RIGHT PARENTHESIS MISSING FOR ARITHMETIC PRIMARY
30	THIS IS AN UNRECOGNIZABLE PRIMARY IN AN EXPRESSION
31	END OF FOR STATEMENT OR <STEP> EXPECTED
32	ARRAY NAME MISSING FROM DIM STATEMENT
33	LEFT PARENTHESIS MISSING FROM DIM STATEMENT
34	AN ARRAY DIMENSION MUST BE AN INTEGER
35	RIGHT PARENTHESIS MISSING FROM DIM STATEMENT
36	COMPILER ERROR IN A DIM STATEMENT
37	THIS FUNCTION WAS PREVIOUSLY DEFINED
38	PARAMETER IN A DEF STATEMENT MUST BE A VARIABLE
39	MISSING RIGHT PARENTHESIS IN THE DEF STATEMENT
40	DEF STATEMENT IMPROPERLY TERMINATED
41	NUMBER OF PARAMETERS DOES NOT AGREE FOR THIS DEF FUNCTION
42	EXPRESSION IN DEF STATEMENT IS INCORRECTLY TERMINATED
43	<FN> IS MISSING FROM FUNCTION NAME IN DEF STATEMENT
44	'SEGMENT' COMPILER CONTROL OPTION CAN NOT BE RESET
45	NUMBER OF DATA LIST ITEMS EXCEEDED PROGRAM LIMIT

46 EXTRA '$ POP SEGMENT' IGNORED
47 READ LIST ELEMENT MUST BE ARITHMETIC OR STRING VARIABLE
48 STRING PRIMARY EXPECTED IN EXPRESSION
49 NUMBER OF DIMENSIONS NOT CONSISTENT
50 <RANDOM> PARAMETER MUST BE A NUMBER OR ARITHMETIC VARIABLE
51 VARIABLE APPEARED AS BOTH A SUBSCRIPTED VAR AND SIMPLE VAR
52 STATEMENT NOS MUST BE ASCENDING AND NONZERO
53 MISSING ASSIGNMENT OPERATOR IN MATRIX STATEMENT
54 MISSING COMMA
55 ILLEGAL ARITHMETIC OPERATOR IN MATRIX STATEMENT
56 <GOTO> WAS EXPECTED
57 END OF MATRIX STATEMENT WAS EXPECTED
58 VARIABLE EXPECTED
59 MISSING RIGHT PAREN IN SCALAR MATRIX EXPRESSION
60 MISSING RIGHT PAREN ON DEF FUNCTION DESIGNATOR
61 MATRIX IDENTIFIER MISSING OR WRONG TYPE IN SCALAR OPERATION
62 LEFT PAREN EXPECTED ON THIS MATRIX FUNCTION
63 RIGHT PAREN MISSING ON DIMENSION PART OF THIS MATRIX FUNCTION
64 MATRIX IDENTIFIER EXPECTED OR WRONG TYPE FOR THIS FUNCTION
65 MISSING RIGHT PAREN IN MATRIX READ DIMENSION PART
66 SEGMENTS NESTED TOO DEEPLY - NESTING LIMIT IS 47
67 TOO MANY DIMENSIONS FOR THIS MATRIX(TYPE STRING)
68 SEGMENT LIMIT MUST BE AN UNSIGNED INTEGER BETWEEN 1 AND 8192
69 UNRECOGNIZABLE MATRIX FUNCTION
70 MATRIX IDENTIFIER EXPECTED
71 STRING ARRAY MUST BE ONE DIMENSIONAL
72 MUST BE AN UNSUBSCRIPTED VARIABLE
73 PRINT/WRITE USING NOT ALLOWED ON BINARY FILES
74 ONLY A FILE IDENTIFIER CAN BE GIVEN AN EXTERNAL NAME
75 END STATEMENT SHOULD BE LAST STATEMENT IN A PROGRAM
76 FILE IDENTIFIER IS MORE THAN SIX CHARACTERS
77 END OF STATEMENT WAS EXPECTED
78 NO DATA STATEMENTS WERE SPECIFIED FOR 1 OR MORE READ STMTS.
79 FILE ALREADY DECLARED
80 COMMA OR COLON EXPECTED AFTER FILE DESIGNATOR
81 ILLEGAL LINE NUMBER REFERENCE
82 FILE NAME MUST BEGIN WITH A LETTER
83 MISSING QUOTE FOR A STRING
84 MISSING COMMA IN STRING FUNCTION
85 ARITHMETIC PRIMARY EXPECTED IN EXPRESSION
86 EXTERNAL FILE NAME MUST BE A STRING
87 EXTERNAL FILE NAME TOO LONG
88 THE LINESIZE DOLLAR OPTION IS NO LONGER SUPPORTED
89 THE OLDBASIC DOLLAR OPTION IS NO LONGER SUPPORTED
90 THE $-OPTION 'SEQ' ONLY ALTERED COLUMNS 73 - 80
91 MNEMONIC 2-LETTER RELATIONAL OPERATOR WAS EXPECTED
92 DEF FN FUNCTIONS MAY NOT BE NESTED
93 LOCAL ITEM IN A DEF STATEMENT MUST BE A VARIABLE
94 INCORRECT FN FUNCTION NAME
95 NUMBER BADLY FORMED
96 NUMBER OF PARAMETERS NOT CONSISTENT
97 INVALID VAL FUNCTION ARGUMENT
98 MISSING COMMA IN PARAMETER LIST FOR SYSTEM FUNCTION
99 INVALID REP$ FUNCTION ARGUMENT

```
100     INVALID ASC FUNCTION ARGUMENT
101     UNKNOWN CHARACTER
102     LIMIT OPTION MUST BE FOLLOWED BY AN INTEGER
103     ILLEGAL TARGET VARIABLE NAME IN LET STATEMENT
104     USERCODE MAY NOT BE SPECIFIED FOR SCRATCH FILES
105     DEF FUNCTION TOO BIG FOR THIS SEGMENT
106     USING CLAUSE NOT ALLOWED IN THIS CONTEXT
107     PRINT TO BINARY FILE NOT DEFINED IN BASIC LANGUAGE
108     FILE DESIGNATOR EXPECTED
109     MISSING LEFT PAREN
110     MISSING RIGHT PAREN
111     MISSING <TO> IN SETW STATEMENT
112     EBCDIC FILE TYPE EXPECTED
113     BINARY FILE TYPE EXPECTED
114     TOO MANY FILES SPECIFIED (LIMIT IS 16)
115     INVALID 'FILES' STATEMENT SYNTAX
116     RECURSIVE FUNCTION CALL NOT PERMITTED
117     MISSING FNEND STATEMENT
118     TOO MANY VARIABLES AND/OR IMAGE STATEMENTS DECLARED
119     CONSTANT TOO LARGE -- OVERFLOW ERROR WILL OCCUR AT RUN TIME
120     MAGNITUDE OF CONSTANT TOO SMALL.  ZERO WILL BE USED.
121
122     RANDOMIZE STATEMENT VALID ONLY WITH ANSI 78 STANDARD
123     NESTED FOR STATEMENT USES PREVIOUS CONTROL VARIABLE
124     TYPE OF PARAMETERS DOES NOT AGREE FOR THIS DEF FUNCTION
125     TYPE OF PARAMETERS NOT CONSISTENT
126     TOO MANY PARAMETERS IN THIS FUNCTION CALL
127     TOO MANY PARAMETERS IN THIS FUNCTION DEFINITION
128     ANSI78 OPTION STATEMENT MAY OCCUR AT MOST ONCE
129     ANSI78 OPTION STMT MUST PRECEDE ALL NON-DOLLAR STATEMENTS
130     UNQUOTED NULL STRINGS NOT ALLOWED IN DATA STATEMENT
131     ILLEGAL ASSIGNMENT IN SWAP STMT - ONLY VARIABLES ARE ALLOWED
132     BOTH VARIABLES IN SWAP STMT MUST BE THE SAME TYPE
133     OPTION STMT MUST PRECEDE DIM STMTS AND ARRAY REFERENCES
134     THERE MAY BE AT MOST ONE OPTION STATEMENT
135     BASE VALUE IS INVALID OR MISSING IN OPTION STATEMENT
136     INVALID BASE SPECIFIED IN OPTION STATEMENT - MUST BE 0 OR 1
137     KEYWORD 'BASE' EXPECTED IN OPTION STATEMENT
138     ARRAY BASE IS ONE - ARRAYS CANNOT HAVE UPPER BOUND OF ZERO
139     THEN/ELSE CLAUSE MAY BE NEITHER BLOCK NOR DECLARATIVE STMT
140     'WEND' STATEMENT EXPECTED
141     UNMATCHED 'WEND' STATEMENT
142     END OF STATEMENT OR COMMA EXPECTED
143     COMMA FOUND WHERE VARIABLE WAS EXPECTED
```

INDEX

ABS Function 5-1
APPEND Statement 7-33
Arithmetic Expression . . 2-5, 3-1, 3-11, 3-15, 4-6, 4-13, 5-1 thru 5-9
. . . . . 5-26, 7-3, 7-34, 7-43
Arithmetic Functions. 5-1, 5-23, 5-26
Arithmetic Operator 2-5
Assignment Lines. 3-1
ASC Function 5-16
ATN Function 5-7

BACKSPACE Statement 7-5, 7-31, 7-37, 7-46,
BACKSPACE$ Statement. 7-37, 7-42
BASE 10 LOG Function. 5-1, 5-2
BASIC Rules 2-13
BASIC Symbol. 2-1
BCL Function. 5-15, 5-16
Binary Files. 7-4, 7-6, 7-8, 7-13, 7-25, 7-37, 7-38, 7-43
Boolean Expressions 2-5, 2-11

CALL Statement 5-26
CLK$ Function 5-15, 5-17
Compiler Options A-1, C-1
CON Statement 6-6, 6-10
Constants 2-2, 2-3, 2-5, 2-9, 4-1, 4-2
. . . . . 5-2, 5-8, 5-9, 5-10, 5-14, 5-15
Control Lines. 3-1, 3-7
COS Function 5-6
COT Function 5-6, 5-7

DAT$ Function 5-15, 5-17
DATA Statement 4-1
DEF Statement 5-14, 5-21, 5-22, 5-33, D-1, D-2
DELIMIT Statement 4-16, 7-24, 7-36
Delimiter. 1-1, 2-1, 2-2, 4-16, 7-19, 7-20
. . . . . 7-31, 7-32, 7-40
DET Function 6-8
Digit. 2-1, 2-2, 2-3, 3-3, 5-4
Dimensioning 6-10
DIM Statement 3-1, 3-3, 6-10, D-1

EBCDIC Files 4-16, 7-4, 7-8, 7-12, 7-15, 7-19, 7-34, 7-37
. . . . . 7-38, 7-40, 7-46, 7-47, 7-50, 7-51, D-3
END Statement 3-7, 3-18, D-2
Error Messages D-1
EXP Function 5-1, 5-2
Expressions. 2-1, 2-5, 4-11, 6-6, 6-14, 7-17, 7-19, D-1, D-2
EXT$ FUNCTION 5-8

File Designator. 7-3, 7-4, 7-7, 7-8, 7-10, 7-12, 7-15, 7-17
. . . . . 7-18, 7-19, 7-21, 7-23, 7-25, 7-26, 7-27
. . . . . 7-29, 7-31, 7-33, 7-34, 7-36, 7-41, 7-42
. . . . . 7-43, 7-47, 7-48, 7-49, 7-50, 7-51, D-2, D-3

INDEX (CONT.)

File Functions 7-47
File Length 7-3, 7-4, 7-6
FILE MAT INPUT Statement 7-10
File Manipulation Statements 7-24
File Mat PRINT Statement 7-21
FILE MAT READ Statement 7-14
FILE MAT WRITE Statement 7-21
FILE PRINT Statement 7-17, 7-18
FILE READ Statement 7-8, 7-12
FILE Statement 7-1, 7-3, 7-8, 7-47
FILE WRITE Statement 7-19
Filename 7-1, 7-2, 7-3, 7-4, 7-7
FILES Statement. 7-1
FNEND Statement 5-24
FOR Statement 3-13

GO TO Statement 3-8
GOSUB Statement 5-27

HPS Function 7-47

IDA Function 5-15, 5-18
IDN STATEMENT 6-6
IF Statement 3-7, 3-8
IF END Statement 7-27
IF MORE Statement 7-29
IF THEN Statement 3-7, 3-9
IF-THEN-ELSE Statement 3-7, 3-10
IMAGE Statement 4-14
INT Function 5-1, 5-3
INV Function 6-7
INPUT Statement. 4-5

Keyboard Symbol. 2-4

Language Components. 2-1
LCW Function 7-48
LEFT Function 5-8, 5-19
LEN Function 5-8, 5-10
LET Statement 3-1
Letter 2-3, 2-4, 3-3, 5-4
LFW Function 7-49
LIN Function 7-50
Line Number 2-13, 3-8, 3-9, 3-10, 3-11, 4-13,
. . . . . 5-27, 6-4, 6-5, 7-23, 7-27, 7-29, A-3
LOG Function 5-1, 5-4
Loops. 3-12

MARGIN Statement 7-34
MAT INPUT Statement 6-32
Matrices 6-1
Matrix Functions 6-6

INDEX (CONT.)

Matrix I/O Statements. 6-1
MAT PRINT Statement 6-3
MAT PRINT USING Statement 6-4
MAT READ Statement 6-1
MAT WRITE Statement 6-4
MAT WRITE USING Statement 6-5
<Matrix Scalar Multiplication> 6-14

New String 5-11
NEXT Statement 3-13
Numeric Variables 2-3
NUM Function 6-18

ON Statement 3-7, 3-11
OPTION Statement 3-1, 3-5

PRINT Formats 4-7
PRINT Statement 4-6
PRINT USING Statement 4-11
Program Loops 3-12

Railroad Programs. 1-1, 1-2
Read and Data Statements 4-1
READ FORWARD Statement 7-41
READ mode 7-5
READ Statement 4-1
Recursive Function 5-20
Relational Expression 2-8, A-3
Relational Operator 2-8
REMARK 2-1, 2-12
REP$ Function 5-8, 5-10
RESTORE Statement 4-3, 7-26, 7-27
RETURN Statement 5-27, 5-28
RIGHT Function 5-8, 5-11
RND Function 5-1, 5-4

SCN Function 5-8, 5-12
SCRATCH Statement 7-25
SETW Statement 7-43
SGN Function 5-1, 5-5
SIN Function 5-6
SPACE Function 5-8, 5-13
SQR Function 5-1, 5-5
Statement Functions, Multiple Line 5-23
Statement Functions/Subprograms 5-23
STEP 3-13, D-1
STOP Statement 3-7, 3-18
STR$ Function 5-8, 5-14
Strings 2-4
String Functions . . . 2-4, 4-1, 4-10, 5-8, 5-14, 5-15, 5-17, 5-21, 5-23
. 6-4

INDEX (CONT.)

String expression 2-9, 3-1, 4-6, 5-19, 6-5, 7-23
String Variables 4-10
SWAP Statement 3-1, 3-6
System Functions 5-15
Subprograms 5-27

TAB 4-7, 4-9, 7-20, C-1
TAN Function 5-7
TIM Function 5-15, 5-18
Trigonometric Functions 5-6
TRN Function 6-7

VAL Function 5-8, 5-15
VPS Function 7-51

WEND Statement 3-7, 3-17
WHILE Statement 3-7, 3-17
WRITE Statement 4-11
WRITE Mode 7-5
WRITE USING Statement 4-13

ZER Statement 6-6

Documentation Evaluation Form

Title: B 5000/B 6000/B 7000 Systems BASIC Reference Manual

Form No: 1160371

Date: July, 1983

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Documentation Dept, - East
Burroughs Corporation
Box CB7
Malvern, PA 19355

2" BINDER

B5000/B6000/B7000 Systems BASIC

REFERENCE MANUAL

1160371

Printed in U.S.A.

1" BINDER

1½" BINDER

Printed in U.S.A.

July 1983

1160371