Burroughs

Reference Manual

B 20 Systems BASIC Compiler

(Relative to Release Level 4.0)

Distribution Code SA

Priced Item
Printed in U.S.A.
April 1985

1180148

Burroughs

Reference Manual

B 20 Systems BASIC Compiler

(Relative to Release Level 4.0)

)istribution Code SA

Priced Item
Printed in U.S.A.
April 1985

1180148

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

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 from time to time to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded, using the Documentation Evaluation Form at the back of the manual, or remarks may be addressed directly to Burroughs Corporation, Corporate Product Information East, 209 W. Lancaster Ave., Paoli, PA 19301, U.S.A.

LIST OF EFFECTIVE PAGES

Page	Issue
iii	Original
iv	Blank
v thru xi	Original
xii	Blank
1-1 thru 1-3	Original
1-4	Blank
2-1 thru 2-18	Original
3-1 thru 3-112	Original
4-1 thru 4-20	Original
5-1 thru 5-41	Original
5-42	Blank
A-1 thru A-9	Original
A-10	Blank
B-1 thru B-10	Original
C-1	Original
C-2	Blank
D-1	Original
D-2	Blank
E-1 thru E-10	Original
F-1	Original
F-2	Blank
G-1 thru G-12	Original
H-1 thru H-5	Original
H-6	Blank
I-1, I-2	Original
J-1 thru J-6	Original
K-1, K-2	Original
1 thru 10	Original

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

LIST OF ILLUSTRATIONS

LIST OF TABLES

HOW TO USE THIS MANUAL

The BASIC Compiler Reference Manual is designed for users who have a working knowledge of the BASIC language, and is not intended to teach a user how to write a program in BASIC. This manual contains all the syntax of the BASIC language with respect to the compiler, so that a user can easily create a BASIC Compiler source program. It also provides all the instructions necessary to install and link these source programs into an executable run file.

ORGANIZATION OF THE MANUAL

This manual contains five sections, several appendixes, a glossary, and an index.

Section 1 provides an overview of the compiler, as well as a discussion of memory organization.

Section 2 contains an explanation of the BASIC language syntax and the structure of a program.

Section 3 lists the BASIC commands and functions in alphabetical order for quick reference. An example and an explanation are given for each command.

Section 4 discusses installing, compiling, and debugging a BASIC program. It also describes how to invoke the Compiler, Linker, Run File, and New Command commands from the Executive.

Section 5 contains a complete listing of BASIC run-time, compile-time, and fatal compiler errors.

The appendixes contain additional information, such as control of the video display, disk input and output, and printing with the printer spooler. Also included are brief discussions on configuring memory usage, non-BASIC procedures, and how to convert programs to B 20 BASIC.

SECTION 1
INTRODUCTION

The BASIC compiler translates BASIC programs into B 20 object module format. The BASIC compiler:

- accepts the most extensive implementation of BASIC available for the 8086 microprocessor.
- separately compiles modules that comprise a BASIC program.
- supports the Public Code options, which allow transfers between separately compiled modules using GOTO, GOSUB, CHAIN, and user-defined functions.
- supports the use of the virtual code segment management facility for overlaying modules of a BASIC program.
- supports the Common DATA option, which permits references to a variable in two or more modules to address the same storage.
- extends the logical maximum string length to 32K bytes.

A typical BASIC program can consist of a main module that displays a menu, accepts input, and then enters one or more subsystems (using CHAIN) based on the menu input. The subsystems (in other modules) execute specific tasks and then return to the main program. The main module also contains common user-defined functions, subroutines, and shared variables.

During execution, calls are made from the subsystems to common routines in the main module and references are made to shared variables. After exiting the subsystems, control is transferred back to the main module (using GOTO).

A program is broken into subsystems when the number and size of the subsystems are such that the entire application cannot be accommodated in available memory.

When a program consists of two or more modules, each module is compiled separately. The main module is compiled with the Common DATA option, and the Public Code option. The subsystems are compiled with the Common DATA option, Public Code option, and Suppress main option.

The BASIC compiler creates a list file displaying information such as the version number of the compiler, source file name, date and time of compilation, and the list file and object file names. It also returns information about memory allocation, symbol table usage, and the number of statements compiled.

When the Linker is invoked, the main module is made resident and the subsystems are placed in overlays, producing an executable run file.

Once linked, the application is invoked with the Run File ccmmand, or with a custom command form created with the New Command command.

Figure 1-1 illustrates an example of a typical compiled BASIC program.

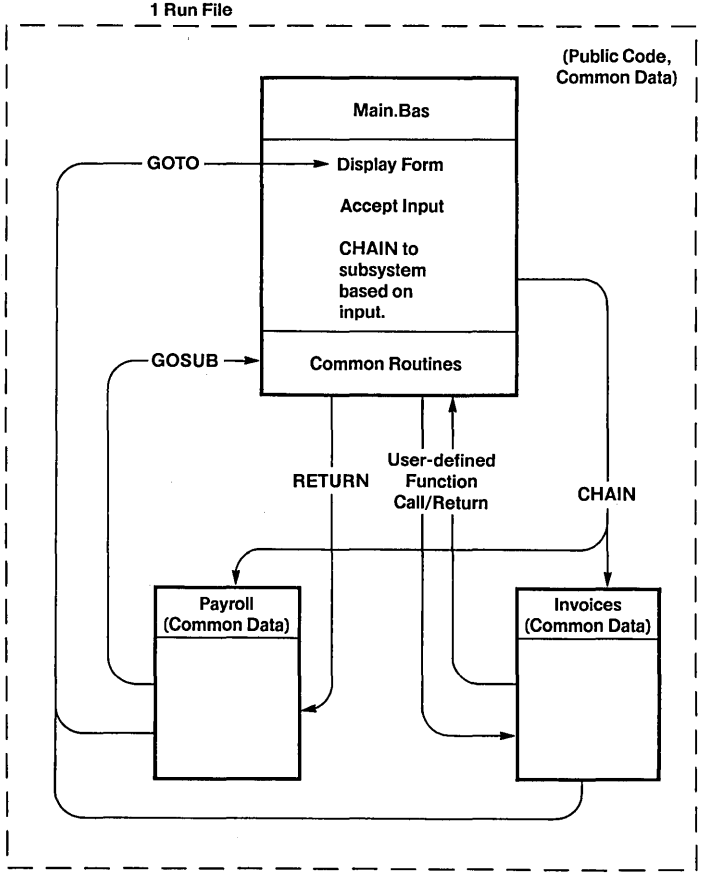

Figure 1-1. A Typical Compiled BASIC Program

A BASIC program consists of several segments. (A segment is a contiguous area of memory not larger than 64K bytes.) These segments are:

- the code in each BASIC module
- the code in each run-time environment module
- BASIC program variables and constants (except string constants), run-time environment variables and constants, and the stack
- the data portion of string constants
- the data contained in DATA statements
- the information used to support the ERL variable, RESUME, and CHAIN

When a BASIC program starts executing, additional memory is required for string variables, file buffers, an overlay zone, and user memory. This memory is dynamically allocated based on configuration parameters.

SECTION 2
THE BASIC LANGUAGE SYNTAX

INTRODUCTION

B 20 BASIC is the most extensive implementation of BASIC available for the 8086 microprocessor. The language includes an extensive formatted printing capability (with the PRINT USING command), full control of the video display, and access to other Burroughs software products, such as Forms, Sort/Merge and ISAM utilities.

LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional items):

```
[nnnnn]  BASIC statement  [:BASIC statement...]
```

The line number indicates the order of execution and is also used for reference when branching. In the BASIC compiler, line numbers are optional. If line numbers are used, the maximum line number supported by the compiler is 65535.

At the programmer's option, more than one BASIC statement can be placed on a line, but each statement of a file must be separated from the last by a colon.

The number of characters in a BASIC program line is unrestricted by the BASIC compiler.

It is possible to extend a logical line over more than one physical line, by adding a continuation character to the end of a line. The BASIC compiler accepts either the MARK or the ampersand (&) as continuation characters. By adding an ampersand or a MARK character to the end of a line, before pressing RETURN, the next physical line will be read as a continuation of the same line. MARK is 02h and can be inserted into the source file by the Editor or Word Processor.

CHARACTER SET

The BASIC character set is comprised of alphabetic, numeric, and special characters.

The alphabetic characters are the uppercase and lowercase letters of the alphabet.

The numeric characters are the digits 0 through 9.

The following special characters and keys are recognized by the BASIC compiler:

CHARACTER OF KEY	NAME
	Space
;	Semicolon
=	Equal sign or assignment symbol
+	Plus sign or string concatenation
-	Minus sign or hyphen
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent sign or integer type declaration character
#	Number (or pound) sign or double-precision declaration character
$	Dollar sign or string type declaration character
!	Exclamation point or single-precision type declaration character
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe) or remark delimiter symbol
"	Double quotation mark or string delimiter symbol
:	Colon or multiple statement separator
&	Ampersand or continuation symbol
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
BACKSPACE	Deletes last character typed

CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

String Constants

A string constant is a sequence of up to 32,767 alphanumeric characters enclosed in double quotation marks. The BASIC compiler interprets two consecutive double-quote characters as a single double-quote character that is contained within a string. For example, the following statement will print the constant enclosed in quotation marks.

```
10 PRINT ""FILENAME""
```

The line feed character can also be contained in a string constant.

Numeric Constants

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants: integer, fixed-point, floating-point, hexadecimal, and octal.

Integer constants are whole numbers between -32768 and +32767. Integer constants do not have decimal points.

Fixed-point constants are positive or negative real numbers (i.e., numbers which contain decimal points).

Floating-point constants (similar to scientific notation) consist of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is 10-38 to 10+38. Examples of floating-point constants are:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
```

Hexadecimal constants are hexadecimal numbers with the prefix &H. Examples of hexadecimal constants are:

```
&H76
&H32F
```

Octal constants are octal numbers with the prefix of &O or &. Examples of octal constants are:

```
&0347
&1234
```

Single- and Double-Precision Constants

Numeric constants can be either single- or double-precision numbers. Single-precision numbers are stored with seven digits of precision, and printed with up to six digits. Double-precision numbers are stored with 16 digits of precision, and print with up to 16 digits.

A single-precision constant is a numeric constant that has one of the following:

- seven or fewer digits
- exponential form using E
- a trailing exclamation point (!)

Listed below are a few examples of single-precision constants:

```
46.8
-1.09E06
3489.0
22.5!
```

A double-precision constant is a numeric constant that has one of the following:

- eight or more digits
- exponential form using D
- a trailing number sign (#)

Examples of double-precision constants are:

```
345692811
-1.09432D-06
3489.0#
.8571428571428571     returned in D# as a double-precision
                      value.
```

VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable can be assigned explicitly by the programmer, or it can be assigned a value, its value is assumed to be 0. Before a string variable is assigned a value, its value is assumed to be null.

Names and Declaration Characters

BASIC variable names can be any length; however, only the first 40 characters are significant. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character must be a letter. Special type declaration characters are also allowed.

A variable name cannot be a reserved word. If a variable begins with FN, then it is assumed to be a call to a user-defined function. Reserved words include all BASIC commands, statements, function names, and operator names.

Variables can represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example, A$ = "SALESREPORT". The dollar sign is a variable type declaration character, that is, it declares that the variable represents a string.

Numeric variable names can declare integer, single-, or double-precision values. Variables without a declaration character are assumed to be single-precision. The type declaration characters for numeric variable names are as follows:

%	Integer variable
!	Single-precision variable
#	Double-precision variable

Example of variable names:

PI#	declares a double-precision value
MINIMUM!	declares a single-precision value
LIMIT%	declares an integer value
N$	declares a string value
ABC	represents a single-precision value

In addition, the BASIC commands DEFINT, DEFSNG, DEFSTR, and DEFDBL can be used to declare the types of variable names. The DEF command and variable types are described in detail in Section 3.

Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array name has as many subscripts as there are dimensions in the array. For example, V[10] references a value in a one-dimensional array, T[1,4] references a value in a two-dimensional array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

Brackets and Parentheses

Although you can use parentheses to delimit array subscripts, you should use brackets. This permits BASIC to easily distinguish between arrays and value-returning non-BASIC procedures. Using brackets improves the readability and performance of BASIC programs.

Pointer Variables

BASIC stores pointers as single-precision numbers, since both pointers and single-precision numbers require four bytes of memory. Pointer values are created by the BASIC PTR function (and by non-BASIC procedures). The only meaningful operations for pointer variables are assignment and equality.

Pointer variables can be used as parameters to non-BASIC procedures, the POKE command, and the PEEK function.

NONEXECUTABLE STATEMENTS

The BASIC statements DIM, DEF FN, DEFINT/SNG/DBL/STR, OPTION BASE, and OPTION COMMA instruct the compiler on how to compile other statements. This type of statement is called nonexecutable.

A nonexecutable statement takes effect when it is compiled and affects all statements that are compiled after it. Therefore, nonexecutable statements should be located at the beginning of a BASIC program.

The following gives an example of how nonexecutable statements are treated by the BASIC compiler.

```
10 GOTO 30
20 DEFINT I
30 I = 100000
```

When line 20 is compiled, the DEFINT statement declares that I is of type integer. Line 30 is compiled as assigning the value 100,000 to integer I. When this program is run, an overflow occurs at line 30 because 100,000 is too large to be an integer.

A BASIC module entered by CHAIN cannot depend on the nonexecutable statements in any other modules since each BASIC module is compiled separately. If nonexecutable statements are required in a module, they must be included in that module. (There is one exception. DEF FN statements do not need to be repeated.) In the following example, note that the DIM statement is repeated in both modules.

```
A.Bas
10 DIM X[5,5]
20 CHAIN MERGE "B.Bas",100,ALL

B.Bas
100 DIM X[5,5]
110 X[1,1] = 0
```

The BASIC compiler does not allow an array variable and a scalar (nonarray) variable to have the same name. If a program contains an array and nonarray variable, the compiler detects the error "Array referenced as scalar".

EXTERNAL USER-DEFINED FUNCTIONS

External user-defined functions are functions that are defined in one BASIC module and referenced in another.

The BASIC compiler does not automatically convert numeric actual parameters to external functions. Therefore, actual parameters must be of the expected type. Also, the compiler does not check that the number of parameters is correct.

When the BASIC program is linked, an error is detected if the number and/or type of actual parameters is different from the number and/or type of parameters required by the function. Error detection is based on the following scheme. When a public user-defined function is declared, an extra public symbol is created that is the function, followed by a string of characters, representing the number and type of formal parameters. Similarly, when a reference is made to an external function, an extra external reference is created that is the name of the function, followed by a string of characters representing the number and type of actual parameters.

In both cases, each character represents a parameter. The character value represents the type, where s is the string, i is integer, r is real (single-precision), and d is double-precision.

For example, the symbol created for the definition:

```
DEF FNADD2%(L%,M%) = L%+M%
```

would be FNADD2%ii.

If actual parameters are correctly passed to an external user-defined function, no Linker errors are detected. However, if actual parameters are incorrectly passed, the Linker will detect an "Unresolved external" error.

TYPE CONVERSION

When necessary, the BASIC compiler converts a numeric constant from one type to another. The following rules and examples should be kept in mind.

If a numeric constant of one type is set equal to a numeric variable of a different type, then the number is stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, then a "Type mismatch" error message occurs.) As shown in the following statements, the integer "23" would be printed as a value of A.

```
10 A% = 23.42
20 PRINT A%
```

During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, as that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

EXAMPLES

```
10 D# = 6#/7
20 PRINT D#
 .8571428571428571
10 D = 6#/7
20 PRINT D
 .857143
```

The arithmetic was performed in double-precision and the result was returned in D# as a double-precision value. The arithmetic was performed in double-precision and the result was returned to D (single-precision variable), rounded and printed as a single-precision value.

Logical operators convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error message occurs.

When a floating-point value is converted to an integer, the fractional portion is rounded. For example, as shown in the following statements, the integer "56" would be printed as the value of C.

```
10 C% = 55.88
20 PRINT C%
```

If a double-precision variable is assigned a single-precision value, then only the first seven digits, rounded, of the converted number are valid. This is because only seven digits of accuracy were supplied with the single-precision value. The absolute value of the difference between the printed double-precision number and the original single-precision value is less than 6.3E-8 times the original single-precision value. As shown in the following example, statement 30 would print "2.04" as the value of A and "2.039999961853027" for B.

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
```

EXPRESSIONS AND OPERANDS

An expression can be simply a string or numeric constant, a variable, or a value-returning non-BASIC procedure. It can combine constants, variables, and procedure calls with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC can be divided into four categories:

arithmetic
relational
logical
functional

Arithmetic Operators

The arithmetic operators, in order of precedence, are shown in Table 2-1.

Table 2-1: Arithmetic Operators

OPERATOR	OPERATION	SAMPLE EXPRESSION
^	Exponentiation	X^Y
-	Negation	-X
*	Multiplication	X*Y
/	Floating-point division	X/Y
\	Integer division	X\Y
MOD	Modulus arithmetic	X MOD Y
+	Addition	X+Y
-	Subtraction	X-Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

The following lists a few sample algebraic expressions and their corresponding BASIC expressions.

EXPRESSION	EXPRESSION
X+2Y	X+Y*2
X-Y/Z	X-Y/Z
XY/Z	X*Y/Z
$(X+Y)/Z$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X(-Y)	X*(-Y)

(Note that two consecutive operators must be separated by parentheses.)

Integer Division and Modulus Arithmetic

Integer division is denoted by the backslash (\). The operands are rounded to integers (in the range of -32768 to 32767) before the division is performed, and the quotient is truncated to an integer, as shown in the following examples.

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating-point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4 = 2 with a remainder of 2)
25.68 MOD 6.99 = 5 (26/7 = 3 with a remainder of 5)
```

The precedence of modulus arithmetic is just after integer division.

Overflow and Division by Zero

If, during the evaluation of an expression, a division by 0 is encountered, then the "Division by zero" error message is detected. Machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in 0 being raised to a negative power, then the "Division by zero" error message occurs. Positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is detected. Machine infinity with the correct algebraic sign is supplied as the result, and execution continues.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result can then be used to make a decision regarding program flow. Table 2-2 lists the allowable operators and a sample expression.

Table 2-2: Allowable Operators

OPERATOR	RELATION TESTED	EXPRESSION
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(Note that the equal sign is also used to assign a value to a variable with the LET command.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression:

```
X+Y<(T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by z.

Logical Operators

Logical operators perform tasks on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result that is either "true" (not zero) or "false" (0). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in Table 2-3 below. The operators are listed in order of precedence.

Table 2-3: Logical Operatores

Logical Operator	X Value	Y Value	Result of Operation between X and Y
NOT	1	--	0
	0	--	1
AND	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	1	1	0
	1	0	1
	0	1	1
	0	0	0
IMP	1	1	1
	1	0	0
	0	1	1
	0	0	1
EQV	1	1	1
	1	0	0
	0	1	0
	0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision. For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to 16-bit, signed, twos complement integers in the range of -32768 to +32767. (If the operands are not in this range, then an error results.) If both operands are supplied as 0 or -1, then logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, that is, each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator can be used to "mask" all but one of the bits of a status byte at a machine input/output port. The OR operator can be used to "merge" two bytes to create a particular binary value. The following examples help demonstrate how the logical operators work.

```
63 =              binary 111111
16 =              binary 010000
63 AND 16 =       binary 010000 = 16

15 =              binary 1111
14 =              binary 1110
15 AND 14 =       binary 1110 = 14

-1 =              binary 1111111111111111
8 =               binary 1000
-1 AND 8 =        binary 1000 = 8

4 =               binary 100
2 =               binary 010
4 OR 2 =          binary 110 = 6

10 =              binary 1010
10 OR 10 =        binary 1010 = 10

-1 =              binary 1111111111111111
-2 =              binary 1111111111111110
-1 OR -2 =        binary 1111111111111111 = -1.
```

(The bit complement of sixteen 0's is sixteen 1's which is the twos complement representation of -1.)

NOT X=-(X+1) The twos complement of any integer is the bit complement plus one.

Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC's intrinsic functions are described in Section 4.

String Operations

Strings can be concatenated with the plus (+) character. For example, as shown in the following statements, line 20 prints "FILENAME" and line 30 prints "NEWFILENAME".

```
10 A$="FILE":B$="NAME"
20 PRINT A$+B$
30 PRINT "NEW"+A$+B$
```

Strings can be compared using the same relational operators as listed in Table 2-2 above.

String comparisons are made by taking one character at a time from each string and comparing the ASCII code. If all the ASCII codes are the same, then the strings are equal. If the ASCII codes differ, then the lower code number precedes the higher. If, during string comparison, the end of one string is reached, then the shorter string is said to be smaller. Leading and trailing blanks are significant.

String comparisons are used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

STRUCTURE OF A BASIC PROGRAM

The Main Module

All compiled BASIC programs must contain one and only module that is designated as the main module. The main module is the first module executed when the program is run.

The compiler designates (by default) a module as main at compile time. The Suppress Main compiler option prevents a BASIC module from being designated as main.

The Suppress Main option must be used whenever a program contains two or more modules. Since only one module can be main, all modules except the main one must be compiled with the Suppress Main option.

Data Statements

If a BASIC program consists of more than one module, the data statements of each module are logically concatenated. The order of concatenation is the same as the order of the object modules as seen by the Linker when the BASIC program is linked.

The RESTORE statement restarts the data statement pointer to the beginning of the concatenated data statements. The RESTORE command is described in more detail in Section 3.

The Public Code Option

The Public Code option allows a BASIC module to transfer control to another BASIC module by using GOSUB, GOTO, or by calling a user-defined function.

The Public Code option is selected when a module is compiled.

Public Code means that all line numbers and user-defined functions in a module are accessible to all modules. Program control can be transferred to any line number of a user-defined function in the module.

In the example shown below, modules Main.Bas and Work.Bas comprise a BASIC program. Work.Bas is compiled with the Public Code option. Program control is transferred from Main.Bas to Work.Bas by using a user-defined function (line 10), GOSUB (line 20), and GOTO (line 30).

```
Main.Bas                    Work.Bas
10 PRINT FNADD2(1,1)        100 DEF FNADD2(X,Y)=X+Y
20 GOSUB 200                200 PRINT "HELLO"
30 GOTO 300                 210 RETURN
                            300 PRINT "GOODBYE"
                            310 END
```

Note that if two modules in a BASIC program are compiled with the Public Code option, they cannot contain the same line numbers.

The Common Data Option

BASIC modules that comprise a BASIC program can share variables using the Common Data option. This option is selected when a module is compiled.

Common Data means that reference to a variable from two or more modules refer to the same area in memory.

In the example shown below, modules A.Bas and B.Bas comprise a BASIC program and are compiled with the Common Data option. (B.Bas is also compiled with the Public Code option). The variable X is assigned the value 10 in A.Bas. This value is printed in B.Bas, since the reference to X in both modules refer to the same area of memory.

```
A.Bas                       B.Bas
10 X = 10                   100 PRINT X
20 GOSUB 100                110 RETURN
30 END
```

Note that when two or more modules share an array, the array must be the same size in each of the modules.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this Section, the arguments are abbreviated as follows:

X and Y	represent any numeric expressions,
P	represents a pointer (single-precision number),
I and J	represent integer expressions, and
X$ and Y$	represent string expressions.

If a floating point value is supplied where an integer is required, BASIC rounds the fractional portion and uses the resulting integer.

SECTION 3

BASIC COMPILER COMMANDS AND FUNCTIONS

This section defines all the BASIC Compiler commands and functions alphabetically for easy reference. Each command or function is presented with the following information:

- an indication of command or function
- a statement which shows the general form of the command/function
- a description of the command/function that explains how the command/function works
- an example of the command/function used in a Basic Compiler program
- an explanation of the example describing the action caused by the command/function

Underlined words represent the information that you supply when using the command/function. For example, the manual treats the PRINT command with the form:

PRINT variable1, variable2

This tells you that the PRINT command may be used with any number of variables, and that you fill in the variable name when you write a PRINT statement.

ABS

Form of Function

ABS (numeric expression)

DESCRIPTION

The ABS function returns the absolute value of the numeric expression.

```
' EXAMPLE: ABS
 LPRINT "Absolute value of 7 and -7 is";ABS(-7)
```

ASC

Form of Function

ASC (string expression)

DESCRIPTION

The ASC function returns the decimal representation of the ASCII code which is equivalent to the first character of the string expression. If the string expression is null an error message results.

```
' EXAMPLE: ASC

 LPRINT "The numeric difference between upper and lower case
ASCII  is";ASC("a")-ASC("A")
```

ATN

Form of Function

ATN (numeric expression in radians)

DESCRIPTION

The ATN function returns the arctangent of the numeric expression. The numeric expression is assumed to be in radians and the function returns values in the range of -pi/2 to pi/2.

The evaluation of the ATN function is always performed in single precision.

```
' EXAMPLE: ATN
 LPRINT "ArcTangent of 12 = "ATN(12)
```

CALL

Form of Statement

CALL procedure name

or

CALL procedure name (first parameter, second parameter,...)

DESCRIPTION

The CALL command allows the user to call a nonvalue returning, non-BASIC procedure. The command may call a procedure using only the procedure name as in the first format, or it may call a procedure and pass information to that procedure as is done in the second format.

The parameters must be integers or single precision numbers. You can not pass strings, arrays or double precision numbers directly.

Example: CALL

```
CALL Proc1
CALL Proc2(Var1,Var2)
```

CDBL

Form of Function

CDBL (numeric expression)

DESCRIPTION

The CDBL function converts a numeric expression to a double precision value.

```
' EXAMPLE: CDBL
 PI# = CDBL(3.14)
```

CHR$ Form of Function

CHR$ (ASCII code number)

DESCRIPTION

The CHR$ function returns the string equivalent of the specified ASCII number. The CHR$ function is often used to display special characters on the video.

```
' EXAMPLE: CHR$
 LPRINT "66 is the ASCII value of ";CHR$(66)
```

CHAIN

CHAIN

Form of Statement

CHAIN "modulename"

CHAIN "modulename", line number

DESCRIPTION

The CHAIN command transfers control from one BASIC module to another. The modulename in the statement format above is the name of the called program.

When the line number option is used, the execution of the called module begins at that line. If the line number option is not used, execution begins at the first line of the called module.

If the Common Data option is selected from the BASIC Compile menu, variables in the current program are shared with other modules. If this option is not selected, the data is not common.

If a module is compiled with the [No CHAIN/ERL info?] option, it cannot be entered using the line number option of the CHAIN command.

EXAMPLE	EXPLANATION
120 CHAIN "MYPROG"	Statement number 120 calls the program MYPROG and begins execution at the first line of MYPROG.
150 CHAIN "MYPROG", 200	Statement number 150 calls the program MYPROG. Execution of MYPROG now begins at line number 200 of MYPROG.

CINT

Form of Function

CINT (numeric expression)

DESCRIPTION

The CINT function converts a decimal numeric expression to an integer. The conversion is done by rounding the fraction portion of the number. The numeric expression must be in the range -32768 to 32767. If it is not, then an error message results.

```
' EXAMPLE: CINT

 LPRINT "CINT truncates -6.34 to ";CINT(-6.34)
```

CLOSE

CLOSE

Form of Statement

CLOSE

CLOSE #first filenumber,
#2nd filenumber,. . .

DESCRIPTION

The CLOSE command is used to end either input to or output from a disk.

The CLOSE command without a file number closes all open files.

The CLOSE command with a file number closes the indicated file. CLOSE file numbers must agree with the file numbers used in the OPEN command to begin output or input.

The END command also closes all open files automatically.

Stop does not close disk files.

EXAMPLE: CLOSE

```
CLOSE
CLOSE#1
```

COS

Form of Function

COS (numeric expression in radians)

DESCRIPTION

The COS function returns the cosine of the numeric expression. Notice that the input to the function is in radians. The evaluation of the COS function is always performed in single precision.

```
' EXAMPLE: COS
 LPRINT "The cosine of 2 is ";COS(2)
```

CSNG

CSNG

Form of Function

CSNG (numeric expression either double precision or integer)

DESCRIPTION .

The CSNG function converts a numeric expression to a single precision value.

```
' EXAMPLE: CSNG

 LPRINT "Large numbers may be converted to single precision with
CSNG"
 LPRINT "      1234567890 is ";CSNG(1234567890)
```

CVI, CVS, CVD

Form of Function

CVI (2 byte string)

CVS (4 byte string)

CVD (8 byte string)

DESCRIPTION

These functions convert string values to numeric expressions. When numeric values are read from a random disk file, they must first be reconverted to numeric values using these functions.

CVI converts a two byte string to an integer value.

CVS converts a four byte string to a single precision value.

CVD converts an eight byte string to a double precision value.

The functions MKI$, MKS$, and MKD$ are opposite in function.

```
' EXAMPLE: CVI/CVS/CVD

 G$=CHR$(10)
' LPRINT CVI(G$)
```

DATA

DATA

Form of Statement

DATA constant, constant, ...

DESCRIPTION

The DATA statement is a non-executable statement that stores either numeric or string constants. The constants in a DATA statement are accessed by means of READ statements in a program.

DATA statements are used in the order of their occurrence in the source file.

String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise quotation marks are not needed.

```
' EXAMPLE: DATA

' see the example for the READ command
```

DATETIME$ Form of Function

DATETIME$ (" ")

or

DATETIME$ (string parameter)

DESCRIPTION

The DATETIME$ function returns a string containing the date and/or time. If the statement is written with a null string, a default template is used that produces a date time of the format.

10/15/82 23:10:14

The date and time can be formatted by using any of the following parameters.

KEY	EXPANSION
h	hour in 12-hour notation (1-12)
t	hour in 24-hour notation (0-23)
m	minute (0-59)
s	second (0-59)
a	AM/PM
y	year (1952-2042)
o	month number (1-12)
n	month name
d	day of month (1-31)
w	weekday name
modifier	meaning
0	zero-fill
*	variable width

DATETIME$ (Continued)

Fields are defined by enclosing the parameters in exclamation points. By default, fields are a fixed-width. Values are truncated to fit into the specified width, on the left for numeric values and on the right for name values. To fill leading digit positions with zeros, use zero (0) as the first character in the field.

Variable width fields can be obtained by entering an asterisk (*) as the first character in the field. The resulting field will be exactly as wide as needed to hold the replacement value.

Capitalization of name values corresponds to the capitalization of the key character. Thus "!AA!" results in "AM" and "!aa!" results in "am". Similarly, "!N!" results in "MAY" and "!n!" results in "May." (Weekday and month names always begin with a capital letter.) Capitalization of key characters for numeric values is not significant.

Text that is not enclosed in exclamation marks is copied directly to the output string. For example, "Today is !*w!." expands to "Today is Wednesday."

```
' EXAMPLE: DATETIME$

 LPRINT "Default Date/Time format"DATETIME$("")
 T$ = "tt!:!mm! !dd!-!oo!-!yy!"
 LPRINT "Special Date/Time format"DATETIME$(T$)
```

DEF

Form of Statement

DEF variable type, initial letter(s) of variables included in the declaration or range(s) of letters

DESCRIPTION

The DEF statement declares variable types.

Type must be:

INT for integer variables
SNG for single precision variables
DBL for double precision variables
STR for string variables

DEF statements may declare a range of variables to be a particular type by including the initial letters of these variables in the DEF statement.

If no type declaration statements are encountered, all variables without declaration characters are assumed as single-precision variables.

EXAMPLE: DEF

```
DEFDBL A-G
DEFSTR X
DEFINT A-C,W-Z
```

DEF FN

Form of Statement

DEF FN function name=
function definition
or
DEF FN function name
(parameter list)=
function definition

DESCRIPTION

The DEF FN statement defines and names a user-created function.

The function name must follow the rules for variable naming. If the function defined is numeric, the function name must be a valid numeric variable name. If a string function is defined, the function name must be a valid string variable.

The function name is always directly preceded by FN.

The variables that appear in the function definition are only used to define the function. They do not affect program variables which may have the same name. When the function is called, the arguments in the calling statement are exchanged on a one for one basis with the parameters in the function defining statement.

A DEF FN statement must be executed before the function it defines can be called. If a function is called before it is defined, the compiler assumes that it will be found in another module. This other module must be linked or an error occurs.

```
' EXAMPLE: DEF FN

DEF FNAT$(X,Y)=CHR$(255)+"C"+CHR$(X)+CHR$(Y)
PRINT fnAT$(35,5);"M A I N   M E N U"
```

DEFLPRINT

Form of Function

erc% = DEFLPRINT (string expression)

DESCRIPTION

The DEFLPRINT function redirects the output of the LPRINT commands to valid devices other than [LPT]. The string expression must be a valid device specification as defined by the operating systems in SamGen.Asm.

DEFLPRINT will close a previously open DEFLPRINT device and open the new one specified. In the case of a spooler device, such as [SPL] or [SPLB], the file will be enqueued immediately.

DEFLPRINT returns a system error code.

EXAMPLE:

```
erc%=DEFLPRINT("[SPL]")
LPRINT("Printing to Spooler")
erc%=DEFLPRINT("[NUL]")
```

DIM

DIM

Form of Statement

DIM array name[array size],
array name[array size]. . .

DESCRIPTION

The DIM statement specifies the maximum number of elements for an array.

The DIM statement sets all elements of the specified numeric arrays to a value of 0, and all elements of string arrays as having no value.

```
' EXAMPLE: DIM

 DIM Array![20,12]
 DIM A[5]
```

END

Form of Statement

END

DESCRIPTION

The END statement terminates program execution, closes all files and returns to the Executive.

```
' EXAMPLE: END

 IF K > 1000 &
 THEN END &
 ELSE K = K + 1
```

Note for System Analysts

End finishes the application and returns to the current exit run file. The exit run file is normally the Executive. It may be reset by any application.

EOF

Form of Function

EOF (file number)

DESCRIPTION

The EOF function returns an evaluation of true when the end of a sequential file is encountered. This allows the user to read a sequential file of unknown length and branch when the end of the file is encountered.

EXAMPLE: EOF

```
OPEN "I",#1,"DATA1"
FOR I = 1 to 1000
IF EOF(1) THEN END
INPUT #1,VAR(I)
NEXT I
```

ERR/ERL

Form of Statement

ERR = errorcode

ERL = linenumber

DESCRIPTION

The ERR and ERL system variables are used during an error trapping routine. The variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF. . . THEN statements to direct program flow in the error trap routine. These variables may not be assigned values.

If they are assigned values, the compiler generates a fatal error and prints the following message:

FATAL ERROR ENCOUNTERED

Contact technical support and report this number: 18

The compiler stops compiling and truncates the .Lst file at the line where the error was encountered.

EXAMPLE: ERR/ERL

```
120 INPUT "CURRENT COST";C
130 IF C > BUDGET THEN ERROR 215
IF ERR=215 THEN PRINT &
  "COST IS OVER BUDGET"
IF ERL=130 THEN RESUME 120
```

ERROR

Form of Statement

ERROR integer expression

DESCRIPTION

The ERROR command allows the user to simulate the occurrence of a BASIC error or create a new error message.

The integer expression of the ERROR command must be in the range of 0 - 255.

To define a new error message, use a value that is greater than any used by BASIC's error messages. This user defined error can then be handled in an error trap routine. See example below.

```
' EXAMPLE ERROR

 ON ERROR GOTO 255
 '
 '
 '
 ERC=BEEP()
 IF ERC <> 0 THEN ERROR 39
 '
 '
 '
255 IF ERR = 39 &
 THEN LPRINT "SYSTEM ERROR #"SYSERC()" AT LINE"ERL &
 ELSE LPRINT "BASIC ERROR #"ERR" AT LINE"ERL
```

EXP

Form of Function

EXP (numeric expression)

DESCRIPTION

The EXP function returns the value of e (e=2.71828 18284 59045) to the power of the numeric expression. The numeric expression can be a single or double precision value.

```
' EXAMPLE: EXP

 R = 5.5
 ANS!=EXP(2*R)
```

FIELD

Form of Statement

FIELD #file number, field width AS string variable, field width AS string variable,....

DESCRIPTION

FIELD statements allocate space for variables in a random file buffer.

FIELD statements must precede GET commands or PUT commands when using random access files.

The file number is the number under which the file was OPENed. Field width is the number of characters to be allocated to string variable.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise an error message occurs.

Any number of FIELD statements can be executed for the same file, and all FIELD statements that were executed are in effect at the same time.

NOTE: Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, then the variable's pointer is moved to string space.

If a dummy file was opened to allow the use of FIELD to define records, do not close the file. When the file is closed, the buffer space defined by FIELD is reclaimed, and its contents are unpredictable.

```
' EXAMPLE: FIELD

' SEE APPENDIX F
```

FIX

Form of Function

FIX (numeric expression)

DESCRIPTION

The FIX function returns the integer part of the numeric expression. The fractional portion of the number is ignored. No rounding of values occurs. See SINT and INT; they also return integer values.

```
' EXAMPLE: FIX

 LPRINT "RESULTS OF FIX(57.34)"FIX(57.34)
```

FOR...NEXT

FOR ... NEXT

Form of Statement

FOR variable=initial value TO final value

FOR variable=initial value TO final value, STEP increment

DESCRIPTION

FOR ... NEXT statements allow a series of instructions to be performed a given number of times.

The initial values and final values may be numbers, expressions or variables.

The variable is used as a counter, and is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is greater than the final value. If it is not greater, then BASIC Branches back to the statement after the FOR statement and the process is repeated. If it is greater, then execution continues with the statement following the NEXT statement.

If step is not specified, then the increment is assumed to be one. If step is negative then the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop and the loop is executed until this counter is less than the final value.

For ... Next loops can be nested but each loop must have a unique variable name as its counter. Also the NEXT statement for the inside loop must appear before the NEXT statement for this outside loop.

The variables in the NEXT statement can be omitted, in which case the NEXT statement matches the most recent FOR statement. If a NEXT is encountered before its corresponding FOR statement, then an error message results.

```
' EXAMPLE: FOR ... NEXT

 FOR J = 10 TO 1 STEP -1
  '
  '
  '
 NEXT J
```

FRE

Form of Function

FRE (0)

DESCRIPTION

The FRE function returns the number of bytes in memory that are not being used by BASIC.

EXAMPLE: FRE

```
PRINT FRE(0)
```

GET

Form of Statement

GET #file number

GET #file number, record number

DESCRIPTION

A GET statement reads a record from a random disk file into the random buffer. The file number is the number under which the file was opened.

The record number specifies the record to be read. If it is an invalid record number an error message results.

If the record number is not specified, the next record after the last GET is read into the random buffer.

The largest possible record number is 32767.

```
' EXAMPLE: GET

' SEE APPENDIX F
```

GETRA

Form of Function

GETRA (pointer)

DESCRIPTION

The GETRA function returns the relative address of a pointer. A pointer can be found with the PTR function, or with certain external library functions.

```
' EXAMPLE: GETRA

 G$ = "GETRA EXAMPLE"
 RA! = GETRA(PTR(G$))
```

GETSA

Form of Function

GETSA (pointer)

DESCRIPTION

The GETSA function returns the segment address of the pointer. The pointer of the variable is returned by the PTR function.

```
' EXAMPLE: GETSA

 G$ = "GETSA EXAMPLE"
 SA! = GETSA(PTR(G$))
```

GOSUB ... RETURN

Form of Statement

```
GOSUB line number

Linenumber Statement
           Statement
           Statement                SUBROUTINE
           Statement
           Statement

RETURN
```

DESCRIPTION

The GOSUB command causes program execution to branch to the subroutine which starts at the line number indicated in the statement. When the statements in the subroutine are completed and the RETURN instruction is met, then execution of the program continues at the line number which follows the nearest GOSUB statement.

A subroutine can be called any number of times in a program and a subroutine can be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

```
' EXAMPLE: GOSUB ... RETURN

 GOSUB 335
 IF A=1 &
 GOTO 340
 '
 '
 '
335 A! = 1
 B! = 2
 '
 '
 RETURN
```

GOTO

GOTO

Form of Statement

GOTO line number

DESCRIPTION

The GOTO command branches to the indicated line number. The program execution then proceeds from that point in the program sequence.

```
' EXAMPLE: GOTO
 GOTO 350
```

HEX$ Form of Function

HEX$ (numeric expression)

DESCRIPTION

The HEX$ function returns the string which is the hexadecimal equivalent of the numeric expression. The numeric expression is rounded to an integer before the hexadecimal string is obtained.

See OCT$ for octal conversion.

```
' EXAMPLE: HEX$

 J = 10
 LPRINT J "DECIMAL IS " HEX$(J) "HEXADECIMAL"
```

IF . . . THEN

IF...THEN / IF...THEN...ELSE / IF...GOTO / IF...GOTO...ELSE

	Form of Statement
IF...THEN	IF expression THEN statement
	IF expression THEN line number
IF...THEN...ELSE	IF expression THEN statement ELSE statement
	IF expression THEN line number ELSE line number
IF...GOTO line number	IF expression GOTO line number
IF...GOTO...ELSE	IF expression GOTO line number ELSE statement
	IF expression GOTO line number ELSE line number

DESCRIPTION

IF statements conditionally direct the flow of a program. When the expression evaluated in the statement is true, the statement or line number following the THEN or GOTO portion of the statement is executed. If the expression evaluated is not true, control passes to either the next executable statement, or to the ELSE portion of the statement, if it exists. THEN and ELSE statements may use a combination of statements and line numbers.

IF...THEN / IF...THEN...ELSE / IF...GOTO / IF...GOTO...ELSE

Nesting of IF Statements:

IF...THEN...ELSE statements can be nested. Nesting is limited only by the length of the line. For example

```
IF     X > Y                    &
THEN   PRINT "GREATER"          &
ELSE   IF      Y > X            &
       THEN    PRINT "LESS"     &
       ELSE    PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, then each ELSE is matched with the closest unmatched THEN. For example

```
IF     A = B                    &
THEN   IF      B = C            &
       THEN    PRINT "A = C"    &
       ELSE    PRINT "A<>C"
```

does not print "A<>C" when A<>B. Prints "A<>C" when A=B and B<>C.

NOTE

When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value cannot be exact. Therefore, the test should be against the range over which the accuracy of the value can vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

```
' EXAMPLE: IF ... THEN [... ELSE]/IF ... GOTO

 IF (1=1) &
 THEN J=14

 IF (1=2) &
 THEN J=400 &
 ELSE G$="TEST IF ... THEN"
```

INP

Form of Function

INP ("B",port number)

INP ("W",port number)

DESCRIPTION

The INP function returns either the byte read from the specified port (form 1) or the word read from the specified port (form 2).

The port number must be an integer in the range of 0 to 65535 for form 1.

The port number must be an even integer in the range of 0 to 65534 for form 2. An Illegal function call error results if an odd number port is used in form 2. See OUT.

INPUT

Form of Statement

```
INPUT variable1, variable2, . . .

INPUT "Prompt message"  ;
      variable1, variable2,. . .

INPUT; "Prompt message"

INPUT "Prompt message",
```

DESCRIPTION

The INPUT statement causes the program to pause while the user enters the variables requested in the statement. The INPUT statement causes a question mark to appear on the video display.

When the form of the INPUT includes a prompt message, such as format 2 above, the message is displayed before the question mark. If a comma follows the prompt, the question mark is omitted.

If INPUT is immediately followed by a semi-colon, the carriage return typed by the user is not echoed to the screen.

The variables in the list can be numeric or string variable names including subscripted variables. The type of each data that is input must correspond with the type specified by the variable name.

Responding to INPUT with too many or too few items, or with the wrong type of value (for example, numeric instead of string) causes the message "?Redo from start" to be printed. No assignment of input value is made until an acceptable response is given.

Note:

INPUT does not accept octal or hex constants.

```
' EXAMPLE: INPUT

 INPUT "Press a key to indicate your choice:",IN$
```

INPUT #

Form of Statement

INPUT# file number, variable1, variable2, . . .

DESCRIPTION

The INPUT # statement reads data from a sequential disk file. The values read from the indicated file number are assigned to the variables which are listed as part of the statement. The file number must agree with the file number of the OPEN statement.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. Leading spaces, RETURN's, and BOUND's are ignored when using numeric values. The first character encountered that is not a space, RETURN, or BOUND is assumed to be the start of a number. The number terminates on a space, RETURN, BOUND, or comma.

If BASIC is scanning the sequential data file for a string item, then leading spaces, RETURN's, and BOUND's are also ignored. The first character encountered that is not a space, RETURN, or BOUND is assumed to be the start of a string item. If this first character is a quotation mark ("), then the string item consists of all characters read between the first quotation mark and the second. Thus, a quoted string cannot contain a quotation mark as a character. If the first character of the string is not a quotation mark, then the string is an unquoted string, and terminates on a comma, RETURN, or after 255 characters are read. If end of file is reached when a numeric or string item is being INPUT, then the item is terminated.

```
' EXAMPLE: INPUT#

' SEE APPENDIX F
```

INPUT$ Form of Function

INPUT$ (number of string characters)

INPUT$ (number of string characters),
file number)

DESCRIPTION

The INPUT$ function returns a string of characters of the specified number. When the function is inputting from the keyboard (format 1), the characters entered are not displayed on the video and all control characters except CANCEL may be used. CANCEL interrupts the execution of the function.

The INPUT$ function may also return strings of characters from sequential files by specifying the file number.

```
' EXAMPLE: INPUT$

 PRINT fnAT$(27,20)"Press any key to continue"
 IN$ = INPUT$(1)
```

INSTR

INSTR

Form of Function

INSTR (string1, string2)

INSTR (position for starting search, string1, string2)

DESCRIPTION

The INSTR function searches for the first occurrence of string 2 in string 1. The value it returns is the position at which the match occurs.

The second form of the function allows the user to specify the position at which the search begins. The position for starting search must be in the range 0 to 255. If the position is greater than the length of string1 or if string1 is null or if string2 cannot be found, then INSTR returns 0. String1 and string2 can be string variables, string expressions or string literals.

```
' EXAMPLE: INSTR

 USER$ = ":SignOnVolume:Sys"
 FirstColon% =INSTR(User$,":")
 NextColon% = INSTR(FirstColon%+1,User$,":")
 Length% = LEN(USER$)-NextColon%
 LPRINT "SignOn volume is"; MID$(USER$,FirstColon%+1)
```

INT

Form of Function

INT (numeric expression)

DESCRIPTION

The INT function returns the largest integer less than or equal to the numeric expression. See CINT and FIX; they also return integer values.

```
' EXAMPLE: INT

 LPRINT "INT EXAMPLE #1"INT(99.89)
 LPRINT "INT EXAMPLE #2"INT(-12.11)
```

KILL

Form of Statement

KILL "filename"

DESCRIPTION

The KILL statement deletes a file from the disk. The "filename" can be a program file or a sequential or random access data file. The KILL statement may only be used to delete a disk file that is not currently open.

If a KILL statement is given for a file that is currently open or for a file that does not exist, then an error message results.

```
' EXAMPLE: KILL

' SEE APPENDIX F
```

LEFT$ Form of Function

LEFT$ (string expression, number of characters of the string)

DESCRIPTION

The LEFT$ function returns the specified number of characters beginning at the left.

If the number of characters specified in the LEFT$ function is greater than the length of the string expression, the entire expression is returned. If it is zero then the null string is returned.

```
' EXAMPLE: LEFT$

 LPRINT "Three left characters"LEFT$("SysImage.Sys",3)
```

LEN

Form of Function

LEN (string expression)

DESCRIPTION

The LEN function returns the number of characters in a string. All characters including blanks are counted.

```
' EXAMPLE: LEN
 J% = LEN(G$)
```

LET

Form of Statement

LET variable = expression

or

variable = expression

DESCRIPTION

The LET statement assigns the value of an expression to a variable. The LET portion of the statement is optional. The variable, equal sign, and the expression are sufficient for assignment statements.

```
' EXAMPLE: LET

 LET X = 40
 LET Y = 60
 LET AREA = X * Y
 X = 40
 Y = 60
 AREA = X * Y
```

LINE INPUT

Form of Statement

LINE INPUT string variable

or

LINE INPUT "prompt string";
string variable

DESCRIPTION

The LINE INPUT statement allows users to input an entire line and assign it to a string variable. The second form of the LINE INPUT statement provides a message on the video display prior to the line to be entered.

The prompt string is a string literal that is printed at the video display before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the RETURN is assigned to <string variable>.

If LINE INPUT is immediately followed by a semicolon, then the RETURN typed by the user to end the input line does not go to the next line on the display.

A LINE INPUT can be escaped by typing CANCEL. BASIC returns to command level and types "OK". Typing CONT resumes execution at the LINE INPUT.

```
' EXAMPLE: LINE INPUT

 LINE INPUT "File name: ",In$
```

LINE INPUT #

Form of Statement

LINE INPUT # file number, string variable

DESCRIPTION

The LINE INPUT # statement reads an entire line from a sequential disk file and assigns it to a string variable.

The file number must agree with the file number of the OPEN statement.

File number is the number under which the file was OPENed.

String variable is the variable name to which the line is assigned.

LINE INPUT# reads all characters in the sequential file up to the end-of-line. Then it skips over the end-of-line sequence, and the next LINE INPUT# reads all characters up to the next end-of-line. (if an end-of-line sequence is encountered, then it is preserved.)

LINE INPUT# is especially useful if each line of a data file was broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

```
' EXAMPLE: LINE INPUT#

 OPEN "I",1,"[SYS]<SYS>.USER"
 LINE INPUT #1,In$
 CLOSE #1
```

LOC

Form of Function

LOC (file number)

DESCRIPTION

The LOC function returns the record number of the next record in a random access file. The next record number is the record number of the last record used with a GET or PUT statement plus 1.

```
' EXAMPLE: LOC
' SEE APPENDIX F
```

LOF

Form of Function

LOF (file number)

DESCRIPTION

The LOF function returns the record number of the last record of a random access file. This allows the user to determine the end of a random disk file.

```
' EXAMPLE: LOF
' SEE APPENDIX F
```

LOG

Form of Function

LOG (numeric expression)

DESCRIPTION

The LOG function returns the natural logarithm of the numeric expression as a single precision value.

```
' EXAMPLE: LOG

 IF LOG(X) > 1.860 &
 GOTO 9000
```

LPOS

Form of Function

LPOS (any numeric expression)

DESCRIPTION

The LPOS function returns the current position of the print head in the line printer buffer. This is not necessarily the actual physical position of the print head.

The numeric expression is a dummy argument and may be any value since it does not affect the function.

```
' EXAMPLE: LPOS

 IF LPOS(X) > 60 &
 THEN LPRINT CHR$(13) ' CARRIAGE RETURN
```

LPRINT

LPRINT

Form of Statement

LPRINT

LPRINT list of expressions

DESCRIPTION

The LPRINT statement prints data on the line printer. The values may be numeric or string. String expressions must be enclosed in quotation marks.

LPRINT assumes that the printer width is 132 characters by default but can be modified by the use of the WIDTH command. A carriage return is inserted at the end of every line. To suppress carriage returns set the printer width to 255.

LPRINT with no variables or expressions listed prints a blank line.

```
' EXAMPLE: LPRINT/LPRINT USING

 LPRINT USING "UNIT PRICE..... $$##.##";UP!

 LPRINT "Market is Down"
```

LPRINT USING

Form of Statement

LPRINT USING "formatting expression";
list of expressions

DESCRIPTION

The LPRINT USING statement prints data on the line printer using a format supplied by the user.

The formatting options are the same as those for the PRINT USING command. See the PRINT USING Section for a listing of formatting expressions and instructions for their use.

```
' EXAMPLE: LPRINT/LPRINT USING

 LPRINT USING "UNIT PRICE..... $$##.##";UP!
```

LSET and RSET

Form of Statement

LSET string variable = string expression

RSET string variable = string expression

DESCRIPTION

Both the LSET and the RSET commands are used to move data into a random file buffer before being stored in a random access file.

LSET left justifies the string variable in the field allocated in the buffer.

RSET right justifies the string variable in the field allocated in the buffer.

If the string is too long for the field, then characters are dropped from the right. Numeric values must be converted to strings before they are LSET and RSET by using the MKI$, MKS$ and MKD$ functions. LSET and RSET can also be used with a nonfielded string variable to left or right justify a string in a given field.

```
' EXAMPLE: LSET

 FIELD 1, 4 AS A$, 20 AS N$
 LSET N$ = CHARNAME$

' SEE APPENDIX F FOR FURTHER EXAMPLES
```

MAKEPOINTER

Form of Function

MAKEPOINTER (relative address integer, segment address integer)

DESCRIPTION

The MAKEPOINTER function returns the pointer which represents the location of the relative address and segment address specified.

```
' EXAMPLE: MAKEPOINTER

 'PRINT THE FIRST BYTE OF MEMORY
 PRINT PEEK("B",MAKEPOINTER(0,0))
```

MID$

Form of Function

MID$ (string expression, beginning position)

or

MID$ (string expression, beginning position, length of character string)

DESCRIPTION

The MID$ function returns a character string which is a portion of the string expression specified in the function.

If the beginning position specified is greater than the length of the furnished string expression, the function returns a null string.

See LEFT$, RIGHT$, and the MID$ command.

```
' EXAMPLE: MID$ Function

 W$ = "SunMonTueWedThuFriSat"
 D$ = MID$(W$,D,3)
```

MID$ Form of Statement

MID$ (string variable1, beginning position in string of replacement) = string variable2

MID$ (string variable1, beginning position in string of replacement, number of characters from string variable2 to be used) = string variable2

DESCRIPTION

The MID$ command replaces a portion of one string expression with another string expression. When the number of characters from the second string value is not entered the entire second string is used up to the length of the original string expression.

See LEFT$, RIGHT$, and the MID$ function.

```
' EXAMPLE: MID$ Command

 DATE$ = "Today is xxx."  ' xxx will be replaced by the
 MID$(DATE$,10,3) = D$    ' three character abbreviation
```

MKI$ MKS$ MKD$ Form of Function

MKI$ (integer expression)

MKS$ (single precision expression)

MKD$ (double precision expression)

DESCRIPTION

These functions convert numeric values to string values. When numeric values are to be placed in a random buffer with an LSET or an RSET statement, they must first be converted to string values using these functions.

MKI$ converts an integer to a two-byte string.

MKS$ converts a single precision number to a four-byte string.

MKD$ converts a double precision number to an eight-byte string.

See CVI, CVS, and CVD.

```
' EXAMPLE: MKI$, MKS$, MKD$
' SEE APPENDIX F
```

NAME

Form of Statement

```
NAME  "old file name"
  AS  "new file name"
```

DESCRIPTION

The NAME command renames a file which has been saved on disk. The old file name must exist and the new file name must not exist to change file names.

```
' EXAMPLE: NAME

 NAME "MasterFile" AS "OldMaster"
```

OCT$ Form of Function

OCT$ (numeric expression)

DESCRIPTION

The OCT$ function returns the string which is the octal equivalent of the numeric expression. The numeric expression is rounded to an integer before the octal string is obtained.

See HEX$ for hexadecimal conversion.

```
' EXAMPLE: OCT$
 PRINT OCT$(NUM!)
```

ON ERROR GOTO

Form of Statement

ON ERROR GOTO line number

DESCRIPTION

The ON ERROR GOTO statement directs program flow to the first line of an error handling routine. All subsequent errors cause program action to be shifted to the error handling routine.

To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors print an error message and halt program execution.

```
' EXAMPLE: ON ERROR GOTO

 ON ERROR GOTO 610
```

ON...GOSUB

ON ... GOSUB

Form of Statement

ON expression GOSUB line number list

DESCRIPTION

The ON ... GOSUB command causes a program to branch to one of the specified subroutines. Each line number in the statement must refer to the first line of a subroutine.

The expression controls which line number from the list of line numbers will be the object of the branch. If the expression is the integer 3, control passes to the third line number in the list. When the expression is a fraction, it is rounded to the nearest integer.

If the value of the expression is negative, zero, greater than the number of entries, or greater than 255 an error message results.

EXAMPLE: ON...GOSUB

```
ON A GOSUB 100,200,300,400
```

ON ... GOTO

Form of Statement

ON expression GOTO line number list

DESCRIPTION

The ON ... GOTO command causes the program to branch to one of the specified line numbers. Each line number in a statement must refer to the first line in a subroutine.

The expression controls to which of the line numbers the program will branch. If the expression is the integer 2, control passes to the second line number in the list of line numbers.

When the expression is a fraction, it is rounded to the nearest integer. If the expression is 0 or greater than the number of entries in the number list, control passes to the next executable statement.

If the value of the expression is negative, zero, greater than the number of entries, or greater than 255 an error message results.

EXAMPLE: ON...GOTO

```
ON A GOTO 100,200,300,400
```

OPEN

OPEN

Form of Statement

OPEN "mode", #file number, "file name"

OPEN "mode", #file number, "file name", record length

DESCRIPTION

The OPEN statement must precede any input or output using disk files.

The mode must be one of the following:

"A" Output is to be appended to a sequential file

"O" Output is to a sequential file

"I" Input is from a sequential file

"R" Output to or input from a random access file

When using "R" mode, each record is written to the disk before the next program statement is executed

"B" Output to or input from a random access file

When using "B" mode, each record is written to the disk file some time before the file is closed, but not necessarily before the next program statement is executed.

"S" Input is from a random access file.

When using "S" mode, more than one program may use the disk file at the same time.

The file number is an integer expression whose value is between one and 255. The number is then associated with the file for as long as it is open. The BASIC runtime allows six files open at one time. This can be changed by reconfiguring the runtime.

The "filename" is a string expression having a maximum of 50 characters.

An optional record length may be specified for random access files. The default record length is 128 characters.

EXAMPLE: OPEN

```
OPEN"I",#5,"DATAIN"
```

OPTION BASE

Form of Statement

OPTION BASE 1

or

OPTION BASE 0

DESCRIPTION

The OPTION BASE command sets the minimum value for array subscripts. The default value is option base 0.

Given the following dimension statement:

```
DIM  A[5]
```

OPTION BASE 1 would reserve space for the locations

```
A[1]    A[2]    A[3]    A[4]    A[5]
```

OPTION BASE 0 would reserve space for the locations

```
A[0]    A[1]    A[2]    A[3]    A[4]    A[5]
```

EXAMPLE: OPTION BASE

```
DIM A[5]
OPTION BASE 1
```

OPTION COMMA

OPTION COMMA

Form of Statement

OPTION COMMA
or
OPTION COMMA string exp

DESCRIPTION

The OPTION COMMA statement changes the physical representation of the following for the input and output of numeric data lists during program execution.

1. Logical input delimiter, which separates data items in the INPUT and WRITE statements;

2. Logical comma, which is used by the PRINT USING statement when formatting numbers;

3. Logical currency symbol, which is used by the PRINT USING statement when formatting numbers;

4. Logical decimal point, which separates the integer and fractional parts of floating point numbers.

If the optional string expression is omitted or is the null string, OPTION COMMA resets the above items to the following new default values.

	Old Value	New Value
logical input delimiter	comma (,)	slash (/)
logical comma	comma (,)	period (.)
logical currency symbol	dollar ($)	dollar ($)
logical decimal point	period (.)	comma (,)

If the optional string parameter is included and is non-null, OPTION COMMA resets the above items based on the first four characters in the string expression.

		New Value
logical input delimiter	-	first character in string exp
logical comma	-	second character in string exp
logical currency symbol	-	third character in string exp
logical decimal point	-	fourth character in string exp

If the string expression contains less than four characters, then the new default values are used for the unspecified items. OPTION COMMA is an executable statement unlike OPTION BASE.

NOTE

OPTION COMMA only affects the execution of a BASIC program with respect to the input and output of numeric data lists. It does not change the syntax of any part of a BASIC source program.

In particular, OPTION COMMA does not change the syntax of data statements or PRINT USING format strings.

```
' EXAMPLE: OPTION COMMA

  OPTION COMMA ""
```

OUT

Form of Statement

OUT"B", port number,integer expression

or

OUT"W", port number,integer expression

DESCRIPTION

The OUT command has two forms. The OUT"B" form sends a byte to a byte port. In this form the port number must be an integer in the range of 0 to 65535 and the integer expression must be in the range of 0 to 255.

The second form of this command, OUT"W", sends a word to a word port. In this form both the port number and the integer expression must be in the range of 0 to 65535 and the port number must be even.

See INP.

PEEK

Form of Function

PEEK ("B", single precision pointer)

or

PEEK ("W", single precision pointer)

DESCRIPTION

The PEEK "B" form returns the byte read from the memory location specified by the pointer. The byte value is a decimal in the range of 0 to 255.

The PEEK "W" form returns the word read from the memory location specified by the pointer. The word value is a decimal in the range of 0 to 65535.

PEEK is the complementary function of POKE.

```
' EXAMPLE: PEEK
A% = PEEK("B",PTR(A!))
```

POKE

Form of Statement

POKE"B", single precision pointer, integer expression

POKE"W", single precision pointer, integer expression

DESCRIPTION

The two forms of POKE statements differ in the value range of the integer expression.

The first form which writes a byte into the memory location specified by the pointer has an integer expression in the range of 0 to 255.

The second form which writes a word into the memory location specified by the pointer has an integer expression in the range of 0 to 65535.

Both integer expressions are the data that are to be poked into the specified memory location.

```
' EXAMPLE: POKE

 POKE "B",A!,0
```

POS

Form of Function

POS("X")

POS("Y")

DESCRIPTION

The POS function returns the position of the cursor in terms of line and column location.

The POS("X") returns the cursor column position. The leftmost position is 0.

The POS("Y") returns the cursor line position. The top position is 0.

```
' EXAMPLE: POS

 IF POS("X") > 60 &
 THEN PRINT CHR$(10)
 IF POS("Y") > 29 &
 THEN PRINT CHR$(12)
```

PRINT

PRINT

Form of Statement

PRINT list of expressions

DESCRIPTION

The PRINT statement causes values to be output to the video display. The values may be numeric or string. String expressions must be enclosed in quotation marks.

The list of variables or expressions may be separated with commas or semicolons.

Separating by semicolons causes each value to be displayed immediately after the last value. Separating by commas causes each value to be displayed in a pre-set column.

```
' EXAMPLE: PRINT

 PRINT "PRINT EXAMPLES"
 ?
 ? "? IS THE SYNONYM FOR 'PRINT'"
 ?
 ?   "TEXT   CAN   BE   INTERSPERSED   WITH   NUMBERS";34;"   AND
QUOTES";CHR$(34)
 ? "TITLES";A!
```

PRINT USING

Form of Statement

PRINT USING "formatting expression";
list of expressions

DESCRIPTION

The PRINT USING statement displays data using a format supplied by the user. The formatting options are applied to either numeric or string variables.

String fields may be formatted using one of the following characters:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, then two characters are printed; with one space, three characters are printed, and so on. If the string is longer than the field, then the extra characters are ignored. If the field is longer than the string, then the string is left-justified in the field and padded with spaces on the right.

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input.

Numeric fields may be formatted using the following special characters:

"#" A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, then the number is right-justified (preceded by spaces) in the field.

"." A decimal point can be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, then the digit always is printed (as 0 if necessary). Numbers are rounded as necessary.

PRINT USING (Cont'd)

"+"	A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.
"-"	A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.
"**"	A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.
"$$"	A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.
"**$"	The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces are filled with asterisks and a dollar sign is printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.
","	A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.
"^^^^"	Four carats (or up-arrows) can be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position can be specified. The significant digits are left-justified and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position is used to the left of the decimal point to print a space or a minus sign.

PRINT USING (Cont'd)

"_"	An underscore in the format string causes the next character to be output as a literal character. The literal character itself can be an underscore by placing "_" in the format string.
"%"	If the number to be printed is larger than the specified numeric field, then a percent sign is printed in front of the number. If rounding causes the number to exceed the field, then a percent sign is printed in front of the rounded number.

```
' EXAMPLE: PRINT USING
 PRINT USING "##.##";.78
 PRINT USING "###.##";987.657
 PRINT USING "##.##   ";10.2,5.3,66.789,.238
 PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
 PRINT USING "##.##-";-68.95,22.449,-7.01
 PRINT USING "**#.#";12.39,-0.9,765.1
 PRINT USING "$$###.##";456.78
 PRINT USING "**$##.##";456.78,2.34
 PRINT USING "####,.##";5430.436
 PRINT USING "####.##,";5430.436
 PRINT USING "##.####^^^^";234.56,987.123
 PRINT USING ".####^^^^  ";88888,-99999
 PRINT USING "+#.###^^^^   ";123,-456
 PRINT USING "TOTAL _$ = $$####.##";5545.89
 PRINT USING "##.##";876523.67
```

PRINT #

Form of Statement

PRINT # file number,list of expressions

DESCRIPTION

The PRINT # statement writes data to a sequential disk file.

The file number must agree with the file number used in the OPEN statement for the specified file.

The PRINT # writes data to a disk file just as it would appear on the video display when a PRINT statement is used.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the video display with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it is input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, then the extra blanks that are inserted between print fields are also written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

PRINT # (Cont'd)

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string variables. If the strings themselves contain commas, semicolons, significant leading blanks, RETURN's, or BOUND's, then write them to disk surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to disk

```
CAMERA, AUTOMATIC  93604-1
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the disk, write double quotes to the disk image using CHR$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk

```
"CAMERA, AUTOMATIC""  93604-1"
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement can also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

PRINT # USING

PRINT # USING

Form of Statement

PRINT# file number USING
"formatting expression";
list of expressions

DESCRIPTION

The PRINT # USING statement writes data to a sequential disk file using a specified format.

All of the formatting expressions used in the PRINT USING statement may be used with the PRINT # USING command.

See the PRINT USING section of this manual for a listing of formatting expressions.

EXAMPLE: PRINT # USING

```
PRINT #2 USING"$$###.##";A;B;C
```

PTR

Form of Function

PTR (variable name or array name)

DESCRIPTION

The PTR function returns the address of the first byte of data assigned to the variable specified. The variable specified in the PTR statement must have been assigned a value prior to the execution of the PTR function.

PTR is often used to obtain the address of a variable or array when a non BASIC procedure is called.

All variables should be assigned values before a PTR function is used to obtain the address of an array. This is because array addresses change when variables are assigned values.

```
' EXAMPLE: PTR

 G$ = "PTR TEST"
 PRINT CHR$(PEEK("B",PTR(G$)))
```

PUT

Form of Statement

PUT #file number

PUT #file number, record number

DESCRIPTION

The PUT statement takes a record from the random buffer and writes it to a random disk file.

The file number of the PUT statement must agree with the file number used in the OPEN statement for the specified file.

When the first form of the PUT statement is used, the record is assigned the next available record number.

When the second form of the PUT statement is used, the record is assigned the record number specified in the statement. The maximum possible record number is 32767.

```
' EXAMPLE: PUT

' SEE APPENDIX F
```

PWA

Form of Function

PWA (P)

DESCRIPTION

The PWA function returns a pointer to word-aligned data given P, where P is a pointer (single-precision number). If the relative address (ra) portion of the parameter is even (that is, it points to a word boundary), the parameter is simply returned. Otherwise, a pointer to the next word boundary is returned.

Whenever a pointer to word-aligned data is used, a data area that is at least one byte longer than normal is required. This ensures that the data area is large enough in the case where it is addressed by a word-aligned pointer.

```
' EXAMPLE: PWA

 DIM BUFFER%[512]

 ERC% = OPENDAFILE...,
   PWA(PTR(BUFFER%[0]))....)
```

RANDOMIZE

RANDOMIZE

Form of Statement

RANDOMIZE

DESCRIPTION

RANDOMIZE is used with the RND function to establish a beginning point (seed) for the random number generator. The RANDOMIZE command takes a random number seed from the run-time system clock.

If RANDOMIZE is not used with RND, then RND will return the same sequence of numbers each time a program is executed. RANDOMIZE changes the seed to produce a different sequence each time the program is executed.

See also RND.

```
' EXAMPLE: RANDOMIZE

 RANDOMIZE
```

READ

Form of Statement

READ variable1, variable2, variable3, ...

DESCRIPTION

The READ statement must be used along with DATA statements.

READ statements assign values to variables from DATA statements in the order in which they occur in the DATA statement. If a program has more than one DATA statement, the statements are read in order of their statement numbers.

READ statements assign numeric or string values to program variables. Variables named in the READ statement must be the same type as the value in the DATA statement or an error message results.

If the number of variables exceeds the number of elements in the DATA statements, then an error message results.

```
' EXAMPLE: READ

 READ IN$
```

REM

REM

Form of Statement

```
REM programmer-supplied remarks
   or
'  programmer-supplied remarks
```

DESCRIPTION

The REM statement allows the programmer to enter explanatory remarks in a program. REM statements have no effect on program execution. When a program containing REM statements is run, the remarks are not displayed. REM statements appear only in the listing of the program.

REM statements may be used anywhere in a BASIC PROGRAM. They can aid programmers by labeling programs and program sections for easy identification. They can also be used to comment on particular statements within a program, by using a single quotation mark instead of :REM at the end of a line.

```
' EXAMPLE: REM or '

REM IS USED TO SPECIFY A REMARK THAT THE COMPILER
'    SHOULD IGNORE.  THE SINGLE QUOTE OR TICK MARK
'    CAN ALSO BE USED AS A REM STATEMENT.
```

RESTORE

Form of Statement

RESTORE

DESCRIPTION

The RESTORE command moves the data stack pointer to the first value in the first DATA statement. Program READ statements begin reading at this point and continue to read values in the usual order. When a line number is specified in the RESTORE statement, the compiler issues a warning "RESTORE<line number> not supported" and resets the data stack pointer to the first value in the first DATA statement.

```
' EXAMPLE: RESTORE

 RESTORE
```

RESUME

RESUME

Form of Statement

RESUME

RESUME 0

RESUME NEXT

RESUME line number

DESCRIPTION

The RESUME command is used in error handling routines. It continues program execution after an error recovery procedure.

A RESUME statement that is not in an error handling routine causes the error message RESUME without error to be displayed.

The RESUME command may direct program flow in three ways.

1. When the form is RESUME or RESUME 0, program execution resumes at the statement which caused the error.

2. When the form is RESUME NEXT, program execution resumes at the statement which immediately follows the one which caused the error.

3. When the form is RESUME line number, program execution resumes at the specified line number.

EXAMPLE: RESUME

```
PRINT ERR;"at line";ERL
IF ERR=39&
THEN PRINT "System error =";SysErc()
STOP
RESUME NEXT
```

RGPARAM$ Form of Function

RGPARAM$(I,J)

DESCRIPTION

The RGPARAM$ function returns a string that is the Jth parameter in the Ith file from the form used to invoke the BASIC program. If no parameter exists, the null string is returned.

The first parameter is number 0. The first field is the form name and is number 0. The first field containing data is number 1.

For further information refer to the section on Parameter Management in the B 20 Operating System manual (form 1171675).

```
' EXAMPLE: RGPARAM$

 COMMAND$ = RGPARAM$(0,0)
 CASE$ = RGPARAM$(0,1)
 FILENAME$ = RGPARAM$(1,0)
 IF FILENAME$ = "" &
 THEN INPUT "FILENAME";FILENAME$
```

RIGHT$

RIGHT$ — Form of Function

RIGHT$ (string expression, number of characters of the string)

DESCRIPTION

The RIGHT$ function returns a string consisting of the number of characters specified beginning at the right of the string. The number of characters must be in the range of 0 to 255.

If the number of characters specified is greater than or equal to the length of the string expression, the entire expression is returned. If it is zero then the null string is returned.

See LEFT$ and MID$.

```
' EXAMPLE: RIGHT$

 G$ = RIGHT$(FILENAME$,2)
```

RND

Form of Function

RND

DESCRIPTION

The RND function returns a random value between 0 and 1.

The sequence of random numbers generated is the same each time the program is run if the random number generator is not reseeded using the RANDOMIZE command.

```
' EXAMPLE: RND
 PRINT INT(RND*6+1)
```

SGN

Form of Function

SGN (numeric expression)

DESCRIPTION

The SGN function returns the sign of the numeric expression. When the expression is greater than 0, the function returns a value of 1.

When the expression is equal to 0, the function returns 0. When the expression is less than 0, the function returns a value of -1.

```
' EXAMPLE: SGN

 IF SGN(X) = -1 &
 THEN X = X * -1
```

SIN

Form of Function

SIN (numeric expression in radians)

DESCRIPTION

The SIN function returns the sine of the numeric expression. The numeric expression must be in radians.

The evaluation of the SIN function is always performed in single precision.

```
' EXAMPLE: SIN
 PRINT SIN(xRads!)
```

SPACE$

SPACE$ Form of Function

SPACE$ (numeric expression)

DESCRIPTION

The SPACE$ function returns a string of spaces of the length specified by the numeric expression. The numeric expression must be in the range of 0 to 255. Any fractional values are rounded before the function returns the string of spaces.

```
' EXAMPLE: SPACE$
 BUFFER$ = SPACE$(80)
```

SPC

Form of Function

SPC(numeric expression)

DESCRIPTION

The SPC function returns a string of blanks. The SPC function differs from the SPACE$ function in that it may only be used along with PRINT and LPRINT commands.

The numeric expression must be in the range of 0 to 255. Fractional values are rounded.

```
' EXAMPLE: SPC

 PRINT SPC(Margin!);TITLE$
```

SQR

Form of Function

SQR (numeric expression)

DESCRIPTION

The SQR function returns the square root of the numeric expression. The numeric expression must be greater than or equal to 0.

```
' EXAMPLE: SQR
 PRINT SQR(2)
```

STOP

Form of Statement

STOP

DESCRIPTION

The STOP command is used to halt program execution. STOP may be used anywhere in a program. When STOP is encountered, the message "Stopped at line linenumber (Press GO to continue; press Cancel or Finish to exit program) is displayed. At this point, if GO is pressed any statements after the STOP statement will be executed.

The STOP command does not close data files as the END statement does.

```
' EXAMPLE: STOP

' SEE THE EXAMPLE FOR RESUME
```

STR$ Form of Function

STR$(numeric expression)

DESCRIPTION

The STR$ function returns the string representation of the numeric expression.

```
' EXAMPLE: STR$
 MID$(BUFFER$,5,1) = STR$(X)
```

STRING$

Form of Function

STRING$ (character string length, ASCII code)

or

STRING$ (character string length, string expression)

DESCRIPTION

The STRING$ function returns a string of the length specified. The elements of the string are the equivalent of the ASCII code specified (form 1) or the first character of the string expression given. (form 2)

```
' EXAMPLE: STRING$

 G$ = STRING$(10,"-")
 PRINT G$;TITLE$;G$
```

SWAP

SWAP

Form of Statement

SWAP variable, variable

DESCRIPTION

The SWAP statement exchanges the values of any two variables. The variables may be any type, (INTEGER, STRING, or SINGLE PRECISION) but both variables must be the same type or an error message results.

```
' EXAMPLE:

 SWAP CURRENT$,PREVIOUS$
```

SYSERC

Form of Function

SYSERC()

DESCRIPTION

The SYSERC function returns the system status code associated with the last BASIC disk access procedure. This allows the user to get more information about BASIC errors. The status codes returned by the SYSERC function are described in the B 20 Standard Software Operations Guide.

The following example illustrates the use of SYSERC() in an error trapping routine. The program deletes an old (possibly nonexistant) file and opens a new file of the same name. If the old file does not exist, a system error results. Since the program does not care if the old file exists or not, the error is trapped, the message printed out and execution continued in the normal manner. (Remember that the BASIC error number is not the same as the system error number.)

```
' EXAMPLE: SYSERC

' SEE THE EXAMPLE FOR RESUME
```

SYSTEM

SYSTEM

Form of Statement

SYSTEM

DESCRIPTION

The SYSTEM command causes an exit from the program. The same action may be accomplished by pressing the FINISH key two times.

```
' EXAMPLE: SYSTEM

 SYSTEM
```

TAB

Form of Function

TAB (print position)

DESCRIPTION

The TAB function moves to the display or print position specified by the numeric expression. If the position specified in the TAB function is prior to the current position, the TAB moves to the next line.

The range of values for the numeric expression is 1 to 255. Fractions are rounded.

TAB may be used only with PRINT and LPRINT commands.

The TAB function is useful in formatting tables and reports.

```
' EXAMPLE: TAB
 PRINT TAB(10);TEXT$
```

TAN

Form of Function

TAN (numeric expression in radians)

DESCRIPTION

The TAN function returns the tangent of the specified numeric expression.

The numeric expression must be in radians.

The evaluation of the TAN function is always performed in single precision.

```
' EXAMPLE: TAN
 Y = TAN(X) * PI
```

USING$

Form of Function

USING$ (string expression,
list of expressions)

DESCRIPTION

The function USING$ returns the formatted output of the PRINT USING statement. The value of the returned string is the same as the output that would be generated if the PRINT USING were invoked with identical numeric parameters.

USING$ does not format text.

```
' EXAMPLE: USING$

' SEE THE EXAMPLES FOR LPRINT USING
```

VAL

Form of Function

VAL (string expression)

DESCRIPTION

The VAL function returns the numerical value of a string representation of a number. If the first character of the string expression is not a +, -, or a digit, the function returns the value 0.

NOTE

VAL does not accept octal and hex constants.

```
' EXAMPLE: VAL

 INPUT "ENTER NUMBER";IN$
 IN = VAL(IN$)
 IF IN = 0 &
 THEN PRINT "ENTRY NOT A NUMBER. TRY AGAIN."
```

VERSION$ Form of Function

VERSION$

DESCRIPTION

The VERSION$ function returns a string that is the contents of the [Version] field of the Linker command form.

CAUTION

If VERSION$ is used and the program is linked without filling in the [Version] field, a linker error will occur.

```
' EXAMPLE: VERSION$
 PRINT "WONDERFUL TEST PROGRAM. version";VERSION$
```

WAIT

WAIT

Form of Statement

WAIT"B", port number, I[,J]

WAIT"W", port number, I[,J]

DESCRIPTION

The port number is the address of the port, specified by an integer expression in the range 0 to 65535. The port may be either a word port (16 bits) or a byte port (8 bits) as indicated by the letter "W" or "B" in the statement. When monitoring a word port, the port number must be even. If port number is odd, the error message "illegal function call" will occur.

I and J are integer expressions which define the input pattern the program is waiting for. The data read at the port is exclusive ORed with the integer expression J, and then ANDed with I. If the result is 0, then BASIC loops back and reads the data at the port again. If the result is nonzero, then execution continues with the next statement. If J is omitted, then it is assumed to be 0.

When monitoring a byte port (WAIT"B"), I and J are integer expressions in the range of 0-255.

When monitoring a word port (WAIT"W), I and J are integer expressions in the range of 0-65535.

CAUTION

It is possible to enter an infinite loop with the WAIT statement.

EXAMPLE: WAIT

```
WAIT "B",32,2
WAIT "W",3030,F008
```

WHILE ... WEND

Form of Statement

```
WHILE expression
    .
    .
    .
STATEMENTS
    .
    .
    .
WEND
```

DESCRIPTION

The WHILE ... WEND statement allows the conditional repetition of statements while the expression is true. When the evaluated expression is not true, program execution transfers to the statement following the WEND statement.

The WHILE statement must precede the matching WEND statement. If WHILE statements are nested, the inner WEND statement must appear before the outer WEND statement.

```
' EXAMPLE: WHILE ... WEND

 MAX! = 10
 DIM BSORT$(10)
 BSORT$(1) = DATETIME$("")
 BSORT$(2) = RGPARAM$(2,0)
 BSORT$(3) = "CONSTANT"
 BSORT$(4) = G$

'     BUBBLE SORT
 FLIPS = 1
 WHILE (FLIPS = 1)
     FLIPS = 0
     FOR J = 1 TO MAX!
  IF BSORT$(J) > BSORT$(J+1) &
  THEN SWAP BSORT$(J),BSORT$(J+1): &
   FLIPS = 1
     NEXT J
 WEND ' END BUBBLE SORT
```

WIDTH

WIDTH

Form of Statement

WIDTH number of characters per line

WIDTH LPRINT number of characters per line

DESCRIPTION

The WIDTH command sets the width of the printed line at the video display. The WIDTH LPRINT command sets the width of the printed line at the line printer.

The number of characters per line must be in the range of 15 to 132 for the video display. For WIDTH LPRINT, the number of characters per line is dependent on the characteristics of the printer. If a compiled BASIC program has a WIDTH statement containing a value greater than 80, the video display width expands to 132 columns on a B22. On a B21 a WIDTH statement containing a value greater than 80 will be ignored. If the WIDTH is less than 15, the video display is set at 80 columns. Any number greater than 132, except 255, is treated as 132. The number 255 suppresses carriage returns.

When you print a string which is longer than the width you have declared, the extra characters are continued on the next line.

EXAMPLE: WIDTH

```
WIDTH 15
```

WRITE

Form of Statement

WRITE list of expressions
separated by commas

DESCRIPTION

The WRITE statement outputs data on the video display. When the WRITE statement is not followed by a list of expressions, a blank line is displayed. WRITE statements may be used with either numeric or string variables.

The WRITE statement differs from the PRINT statement in that it prints numeric values followed by a comma and prints string variables enclosed in quotation marks.

EXAMPLE: WRITE

```
WRITE A,B,C$
```

WRITE#

WRITE# Form of Statement

WRITE# filenumber, list of expressions separated by commas

DESCRIPTION

The WRITE# command writes data to the sequential file specified by the filenumber. The file number must agree with the filenumber of the OPEN statement for the specified file.

The expressions and variables in the list may be either numeric or string.

The WRITE# statement differs from the PRINT# in two ways. First it uses a comma to separate the values that are output. Second, it writes string variables enclosed in quotation marks. The WRITE# offers the advantage of not requiring the user to insert specific delimiters when writing to sequential files.

EXAMPLE: WRITE#

```
OPEN,"0",#1,"DATA"
WRITE#1,A$;B$
```

SECTION 4
INSTALLING, COMPILING, AND DEBUGGING A BASIC PROGRAM

INTRODUCTION

This section discusses installing and using the BASIC compiler to create an object module, and using the Linker to create the run file. Also described is how to install a run file with the "New Command" command, and how to invoke the system debugger.

INSTALLING THE BASIC COMPILER

The following are specific instructions for installing Basic Compiler.

Disk Contents

Basic Compiler (style ID B20BC4-1) contains the following files on a single disk:

```
VOLUME: [B20BC4-1]

   DIR: <Sys>
        fileHeaders.sys
        mfd.sys
        log.sys
        sysImage.sys
        badBlk.sys
        crashDump.sys
        bootExt.Sys          (8" only)
        DiagTest.sys         (8" only)
        XEInstall.Sub

        Install.Sub
        FdSys.Version
        Sys.Cmds
        BrSwp.obj
        BasGen.obj
        BrCnfg.obj

   DIR: <Burroughs>
        BasicCompile.Run
        BasCompiler.lib

   DIR: <BCCnfg>
        BrCnfg.Asm
        BasGen.Asm
        BasicComp.fls
        ObjBasRes.fls
        ObjFormsSwp.fls
```

Loading Instructions

1. If the system is a cluster system, turn off the cluster workstations.

2. Insert the product distribution disk in drive [f0] of the master.

   (Do **NOT** press the RESET button.)

3. Install the product as follows:

```
Command      Software Installation  <RETURN>
               [Cmd File]
               [Files to]
               [Confirm?]
               [Install file]
```

   Name the [Cmd file] where 'Basic Compiler' will be created. The default is [Sys]<Sys>Sys.Cmds.

   Type the destination [vol]<dir> of the files in the field [Files to]. The BasCompiler.lib will always be placed in [Sys]<Sys>.

   Answer 'y' to [Confirm?] to check each file as it is installed.

   The only valid install file on the disk is [f0]<sys>Install.Sub. Since this file is the default, there is no need to type it.

4. The installation procedure:

   Creates a command in the command file specified called 'Basic Compiler'

   Copies the necessary files to run and customize the compiler to the directory specified in [Files to].

5. After the installation is complete, save the disk as a master.

Memory Requirements

The compiler requires 170K bytes of user memory to compile any program. A program created with the Basic Compiler requires at least 37K bytes, but may require more depending upon the size of the program and the options used. For further information refer to Appendix G, Programming Hints.

INVOKING THE BASIC COMPILER

To invoke the BASIC compiler from the B20 Executive,type "B C" (or as many letters as are required to make the command unique.

The following form is displayed:

```
Basic Compile
      Source files
      [Object files]
      [List files]
      [BasGen file]
      [Suppress main?]
      [Public code?]
      [Common data?]
      [Errors only?]
      [No bounds checking?]
      [No RESUME info?]
      [No CHAIN/ERL info?]
      [No code generation?]
```

A description of the source fields is listed below:

Source files

Fill in the "Source files" field with the names of one or more files containing BASIC source programs. Each BASIC source file is compiled separately.

[Object files]

Fill in the "[Object files]" field with the names of one or more object files to be created by the compiler. If no object file name is specified, the default name will be the source file name suffixed by ".Obj." Any existing suffixes will be replaced by .OBJ. If two or more entries are specified in both the "Source files" and "[Object files]" fields, then the source and object file names match according to position. That is, the first source file name corresponds to the first object file name, and so on.

[List files]

Fill in the "[List files]" field with the names of one or more list files to be created by the compiler. If no list file name is specified, the default name is the source file name suffixed by ".Lst." If two or more entries are specified in both the "Source files" and "[List files]" fields, the source and list file names match according to position. That is, the first source file name corresponds to the first list file name, and so on.

[BasGen file]

If the BASIC module being compiled calls non-BASIC procedures, fill in the "[BasGen file]" field with the name of the object module produced by assembling BasGen.Asm (see Appendix B for further details).

[Suppress main?]

If the BASIC module being compiled is not main, fill in the "[Suppress main?]" field with y or yes. The default is no; the module is compiled as a main module. One module in a program must be the main module.

[Public code?]

If the BASIC module being compiled contains line numbers or user-defined functions that are entered from other modules (via GOTO, GOSUB, or function invocation), fill in the "[Public code?]" field with y or yes. When this option is selected, all line numbers and user-defined functions are made public. The default is no; the code is not public.

[Common data?]

If the BASIC module being compiled contains variables that are shared with other modules, fill in the "[Common data?]" field with y or yes. The default is no; the data is not common.

[Errors only?]

If you fill in the "[Errors only?]" field with y or yes, only error messages are sent as output to the list file. The default is no; a full listing of the entire module is sent as output.

[No bounds checking?]

Fill in the "[No bounds checking?]" field with y or yes if you do not want the compiler to generate code that checks the value of array subscripts to ensure that they are in bounds. The default is no; code for bounds checking is generated.

[No RESUME info?]

Fill in the "[No RESUME info?]" field with y or yes if the program does not use RESUME, RESUME 0, or RESUME NEXT. This option suppresses the generation of information needed to support these statements and reduces the program's memory requirements. The default is no; the RESUME information is generated.

[No CHAIN/ERL info?]

Fill in the "[No CHAIN/ERL info?]" field with y or yes if the program does not use ERL, RESUME, RESUME 0, RESUME NEXT, or CHAIN <line number>. This option suppresses the generation of information needed to support these statements and reduces the program's memory requirements. The default is no; the CHAIN/ERL information is generated.

(Note that RESUME <line number> can be used even when this option is selected.)

[No code generation?]

Fill in the "[No code generation?]" field with y or yes if you do not want the compiler to produce an object module. The default is no; the object module is produced. Use this option when you are using the compiler only as a syntax checker.

COMPILATION

The compiler generates an object file (or module) and a list file for each BASIC module. Object and list file name specifications are optional.

Each module that comprises the BASIC program is compiled separately. A set of modules that uses identical compiler options can be compiled in a single invocation of the compiler by entering each source file name in the "Source files" field, separated by spaces.

One module must be compiled as the main module (see the "Main Module" subsection in the "General Information" section above). All other modules must be compiled with the Suppress Main option.

Any module can be compiled with the Common Data option.

Any module that is not going to be placed in an overlay can be compiled with the Public Code option.

FATAL ERROR CONDITIONS

There are certain conditions that can arise while the program is compiling that force compilation to be terminated. These fatal errors are described in Section 6.

LISTING FORMAT

All list files created by the BASIC compiler are formatted as shown in the examples in Figures 4-1 and 4-2.

```
Basic Compiler 4.0x

Source file [sys]<sys>tutorial.bas compiled on Oct 19, 1982 at
11:19 AM

List file is [sys]<sys>tutorial.lst
Object file is [sys]<sys>tutorial.obj
Options:  public code, common data

10 'Basic program using Tutorial.form
11 'Illustrating fixed and repeating fields, user input and
program output
20 option base 1
```

Figure 4-1. Example of the Beginning of a List File.

As shown in Figure 4-1, the first line in a BASIC list file always shows the version number of the compiler. This is followed by the source file name and the date and time of compilation, and the list file and object file names. In the above example, the source, list, and object file names are preceded by their [volume] and <directory>.

The names of the options that were specified are shown (in this case, Public Code and Common Data), followed by the first lines of code.

As shown in Figure 4-2 below, the memory allocation (in bytes) for the program appears after the last line of code. This is broken down by the amount of memory allocated to code, data, constants, and stack.

Next, symbol table usage (in bytes) is shown by total bytes, number of bytes used, and number of bytes remaining.

The number of errors and warnings that occurred are shown next. In this example, no errors or warnings occurred.

The last line in the listing shows the number of statements compiled. In this example, 117 statements were compiled.

```
9010 print "error ", erc%:stop
9020 goto 1500

Memory allocation in bytes
  code: 2455  data: 146  constants: 1162  stack: 500

Symbol table usage in bytes
  total: 65520  used: 2990  remaining: 62530

No errors, 117 statements compiled
```

Figure 4-2. Example of the End of a List File.

INVOKING THE LINKER

After the BASIC modules are compiled, the resulting object modules are linked with the Linker to produce an executable run file.

To invoke the Linker, type "Link" in the Executive command form. Then press RETURN. The following form is displayed:

```
Link

  Object modules
  Run file
  [List file]
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max memory array size]
  [Min memory array size]
  [System build?]
  [Version]
  [Libraries]
  [DS allocation]
  [Symbol file]
```

Fill in the fields as required. Normally, you will be concerned with two fields: The "Object modules" and "Run file" fields. All fields after the "Run file" field are optional. You can default optional fields by leaving them blank. (Press RETURN to skip a field.) For definitions of all the fields in the Link form, see the "Invoking the Linker" section of the B20 Linker/Librarian Reference Manual .

OVERLAYS

When you invoke the Linker, you can optionally specify that one or more modules (object files) be placed in one or more overlays. If a module is in an overlay, the code portion of the module resides on disk until it is actually needed by the executing program. At that time, the code portion is automatically read from the disk into memory and executed (see the BTOS Operating System Manual.)

Overlays reduce the physical memory requirements of a program. A program typically performs a sequence of independent steps or contains a number of distinct subsystems. If each step or subsystem is contained in a separate module, then each module can be placed in an overlay. In this way, a program can be run that is much larger than the amount of available physical memory.

To use overlays, a BASIC program need not contain any special commands or statements. The compiled BASIC run-time system automatically allocates memory to contain the overlays loaded from disk.

To use the overlays, the following restrictions apply:

- A module compiled with the public code option cannot be in an overlay.
- The main program cannot be in an overlay.
- The object module, BrSwp.Obj, must be included in the "Object module" field of the Linker command form.
- The following run-time modules in BasCompiler.Lib cannot be placed in an overlay:

BrInit	BrMem
BrErc	BrErcA
BrFlow	BrChn
BrH	BrSwp

A module is placed in an overlay by inserting /O (which represents the overlay switch) at the end of the object module name in the "Object modules" field of the Linker command form.

The example below shows the "Object modules" field of the Linker command form with an object module name containing the /O switch.

Object modules BrSwp.Obj X.Obj Y.Obj/O

The CHAIN statement (see the "CHAIN" subsection in the "Commands and Statements" section below) transfers control to a BASIC module in an overlay.

Note that GOTO, GOSUB, or a user-defined function call cannot be used to transfer control to an overlay because line numbers and user-defined function names cannot be made public.

OVERLAYING THE RUN-TIME SYSTEM

The memory requirements of a BASIC program can be substantially reduced if parts of the BASIC run-time system are placed in overlays. In fact, it may be necessary to overlay the run-time system because once overlays are used, the size of the resident is restricted to 64k bytes.

The files ObjBasRes.Fls, BasicComp.Fls, and ObjFormsSwp.Fls are partial object module lists that indicate which run-time modules must be resident and which can be in overlays. Use the Editor or Word Processor to construct an object module list based on these files and your own list of BASIC modules.

BASICRES.fls contains a list of object modules that must be resident. The modules in this list cannot be placed in overlays.

BASIC.fls contains a list of object modules that can be placed in overlays. These modules can also be resident.

ObjFormsSwp.Fls contains a list of object modules in the Forms library that can be placed in overlays. These modules can also be resident.

LINKER ERRORS

Incorrect BASIC programs can cause a number of Linker errors. Some sample Linker errors and their causes are given below.

Multiply defined symbol Begin in file A.Obj

> Two or more main modules are linked together. Only one main module can be in a BASIC program.

Multiply defined symbol @100 in file A.Obj

> Two or more public modules with overlapping line numbers are linked together. Public modules in a BASIC program must have line numbers that do not overlap.

Multiply defined symbol fnadd2% in file A.Obj
Multiply defined symbol fnadd2%ii in file A.Obj

> Two or more public modules define user-defined functions that have the same name. Public modules in a BASIC program must have user-defined names that are unique.

Unresolved Externals:
@1000 in file(s): B.Obj

> A BASIC module referenced a line number that is not defined by any other module. A line number that is referenced in a GOTO/GOSUB statement must be declared in the module containing the GOTO/GOSUB, or in another public module in the same BASIC program.

```
Unresolved Externals:
fnadd2% in file(s): B.Obj
fnadd2%ii in file(s): B.Obj
```

In this example, both the user-defined function name and the function name followed by a parameter string are unresolved externals. A BASIC module referenced a user-defined function that is not defined by any other module. A user-defined function that is referenced must be declared in the module containing the reference, or in another public module in the same BASIC program.

```
Unresolved Externals:
fnadd2%ir in file(s): B.Obj
```

In this example, only the function name followed by the parameter string is an unresolved external. A BASIC module referenced an external user-defined function incorrectly. The number and/or type of actual parameters passed to the function are different from the number and/or type of parameters required by the function.

The string of characters following the type character ($, %, !, or #) in the function name identifies the number and type of actual parameters. Each character represents a parameter. The character value represents the type, where s is string, i is integer, r is real (single-precision), and d is double-precision.

Check the definition of the user-defined function to determine the number and type of actual parameters to be passed.

```
Error: Resident code exceeds 64k
```

The size of the resident code exceeds 64k bytes. More modules must be placed in overlays. If no more BASIC modules can be placed in overlays, place portions of the BASIC run-time system in overlays. Use the files ObjBasRes.Swp, BASIC.fls, and ObjFormsSwp.Fls to construct an object module list that places parts of the run-time system in overlays.

```
Warning: Non "CODE" class loaded into overlay
```

This warning can occur if "BrSwp.Obj" is placed in an overlay. Review the object module list and make sure that "BrSwp.Obj" appears before any module that uses the /O switch.

This warning can also occur when an assembly language module is placed in an overlay. The Linker expects all code segments to have the class-name "CODE".

Warning: Module B.Obj compiled with Publics is not resident

This warning occurs if a module compiled with the Public code option is placed in an overlay. Public Code modules cannot be placed in overlays.

Warning: No "OverlayFault" procedure loaded

BrSwp.Obj has been left out of the object modules list. When a module is placed in an overlay, BrSwp.Obj must be included in the object module list before any of the modules that use the /O switch.

FATAL RUN-TIME ERRORS

Conditions may arise while executing a BASIC program that make it impossible for the program to continue running. These conditions are due to fatal run-time errors.

When a fatal run-time error occurs, the BASIC program may either:

1. return to the Executive, where a status code is displayed, indicating what error condition occurred.

   or

2. stop program execution with a fatal error message and the prompt:

   (Press FINISH or CANCEL to return to the Executive).

If a status code in the range 1380 through 1390 is displayed, a BASIC run-time internal error has occurred. In this case, report the error to your system administrator or technical support.

PUBLIC SYMBOLS

BASIC programs that are to be debugged should be compiled with the Public Code and Common Data options. If this is done, the symbol file created by the Linker, when the program is linked, will contain public symbols for all line numbers, user-defined functions, and program variables.

The public symbol for a line number consists of the @ ("at" sign) followed by the line number. For example, "@10" is the public symbol for line number 10.

The public symbol for a user-defined function consists of the user-defined function name followed by a type character. The type character corresponds to the type of value that the function returns. These characters are "$" for strings, "%" for integers, "!" for single precision, and "#" for double precision. For example, fnadd2% is the public symbol for a user-defined function named fnadd2%. To examine or modify the code that performs this function, type fnadd2% followed by MARK.

To set a breakpoint, enter the debugger, type the appropriate public symbol, and then type CODE-B.

To display memory, use one of the debugger's memory display commands (CODE-D, MARK, right arrow, and left arrow). Use the appropriate public symbol as an operand.

ADDITIONAL PUBLIC SYMBOLS

The compiler generates additional public symbols that are described below in Table 4-1.

Table 4-1. Additional Public Symbols

Public Symbol	Description
Begin	Located at the beginning of the main program.
ChainTable	Located at the beginning of the chain table data structure. This structure is used at run time by the CHAIN command.
<module name>	This symbol is the filename of the BASIC module without any file extension. It is located at the beginning of the code portion of non-main BASIC modules.
<module name>Entry	Located at the first instruction that is executed in a module when it is entered using the CHAIN command.

NAMES OF BASIC RUN-TIME PROCEDURES

The compiler makes many calls to procedures in the compiled BASIC run-time environment. Each procedure has a corresponding public symbol that is the same as its name. These procedures are described below in Table 4-2.

Table 4-2. Names of BASIC Run-Time Procedures

Procedure	Description
BrAsc	Convert first character of string to integer.
BrChain	CHAIN.
BrChr	Convert integer to string of length 1 (CHR$).
BrCleanup	Exit program (END).
BrClose	Close file (CLOSE).
BrCloseAllFiles	Close files (CLOSE).
BrCvi	Convert 2-byte string to integer (CVI).
BrCvs	Convert 4-byte string to single-precision (CVS).
BrCvd	Convert 8-byte string to double-precision (CVD).
BrDateTime	Get formatted date and/or time.
BrDefLP	Define line printer filename (DEFLPrint).
BrEof	Query end of file (EOF).
BrError	Simulate error (ERROR).
BrException	Handle error exception.
BrField	Associate string with file buffer (FIELD).
BrGet	Read random record (GET).
BrHexOctI	Convert integer to hexadecimal/octal ASCII string (HEX$, OCT$).

Table 4-2. BASIC Run-Time Procedures (Contd.).

Procedure	Description
BrHexOctR	Convert single-precision to hexadecimal/ octal ASCII string (HEX$, OCT$).
BrHexOctD	Convert double-precision to hexadecimal/ octal ASCII string (HEX$, OCT$).
BrInit	Initialize run-time system.
BrLPos	Query line printer position (LPOS).
BrMki	Convert integer to 2-byte string (MKI$).
BrMks	Convert single-precision to 4-byte string (MKS$).
BrMkd	Convert double-precision to 8-byte string (MKD$).
BrInch	Input characters (INPUT$).
BrInLine	Input line (LINE INPUT, LINE INPUT #).
BrInpt	Input data (INPUT, INPUT #).
BrKill	Delete file (KILL).
BrLoc	Query current record (LOC).
BrLof	Query last record (LOF).
BrOnError	Set up error handler (On Error GOTO).
BrOpen	Open file (OPEN).
BrPos	Query cursor position (POS).
BrPrnt	Print data (PRINT, PRINT #, LPRINT).
BrPU	Print data using format information (PRINT USING, PRINT USING #, LPRINT USING).
BrPut	Write random record (PUT).
BrRand	Send random number generator (RANDOMIZE).

Table 4-2. BASIC Run-Time Procedures (Contd.)

Procedure	Description
BrRandom	Generate a random number (RND).
BrRgParam	Read command form.
BrRename	Rename file (NAME).
BrResume	Resume program (RESUME).
BrStop	Suspend program (STOP).
BrVException	Handle error exception.
BrWidth	Set video width (WIDTH).
BrWrite	Write data (WRITE, WRITE#).
DAbs	Compute double-precision absolute value (ABS).
DAdd	Add double-precision values.
DCeilD	Computer largest integer from double-precision value.
DDiv	Divide double-precision values.
DecodI	Convert string to signed integer.
DecodR	Convert string to single-precision.
DecodD	Convert string to double-precision.
DFLtI	Convert signed integer to double-precision.
DFLtR	Convert single-precision to double-precision.
DivI	Divide integer values.
DMul	Multiply double-precision values.
DNeg	Negate double-precision value.
DSign	Compute sign of double-precision value (SGN).

Table 4-2. BASIC Run-Time Procedures (Contd.)

Procedure	Description
DSub	Subtract double-precision values.
EncodI	Convert signed integer to string (STR$).
EncodR	Convert single-precision to string (STR$).
EncodD	Convert double-precision to string (STR$).
FEqD	Compare double-precision for equality.
FLeD	Compare double-precision for less than or equal to.
FLtD	Compare double-precision for less than.
FEqR	Compare single-precision for equality.
FLeR	Compare single-precision for less than or equal to.
FLtR	Compare single-precision for less than.
FStrEq	Compare string for equality.
FStrLe	Compare string for less than or equal to.
FStrLt	Compare string for less than.
IAbs	Compute absolute value of integer (ABS).
IAdd	Add integer values.
IMod	Mod integer values.
IRounD	Convert double-precision to integer (rounded).
ISign	Compute algebraic sign of an integer (SGN).
ISub	Subtract integer values.
ITrunR	Truncate single-precision to integer (FIX).

Table 4-2. BASIC Run-Time Procedures (Contd.)

Procedure	Description
ITrunD	Truncate double-precision to integer (FIX).
MulI	Multiply integer values.
RAbs	Compute single-precision absolute value (ABS).
RAdd	Add single-precision values.
RAtan	Compute single-precision arctangent (ATAN).
RCeilR	Compute largest integer from single-precision value (INT).
RCos	Compute single-precision cosine (COS).
RDiv	Divide single-precision values.
ReadDA	Read DATA (READ).
RestDA	Initialize or restore DATA (RESTORE).
RExp	Compute exponential function (EXP).
RExpI	Raise single-precision values to a power.
RExpR	Raise single-precision values to a power.
RFLtI	Convert signed integer to single-precision.
RLn	Compute log arithmetic function (LOG).
RMul	Multiply single-precision values.
RNeg	Negate single-precision values.
RRounD	Convert double- to single-precision (rounded).
RSign	Compute algebraic sign of a single-precision value (SGN).
RSin	Compute sine function (SIN).

Table 4-2. BASIC Run-Time Procedures (Contd.)

Procedure	Description
RSqrt	Compute square root function (SQR).
RSub	Subtract single-precision values.
RTan	Compute tangent function(TAN).
StrAss	Concatenate strings.
StrAssl	Assign string.
StrFind	Search string (INSTR).
StrFe	Query free memory (FRE).
StrGet	Compute substring (MID$).
StrLOp	Compute left substring (LEFT$).
StrLSet	Copy string (LSET).
StrMake	Construct string (STRING$, SPACE#).
StrPut	Assign string to substring (MID$).
StrROp	Compute right substring (RIGHT$).
StrRSet	Copy right string (RSET).
StrVal	Convert string to double-precision value (VAL).

DATA TYPES AND REGISTER USAGE CONVENTIONS

The register usage conventions in compiled BASIC generated code follow many of the conventions for other languages. These conventions are summarized below.

- Integers require one word. Integer values are returned by functions in the AX register. Integer parameters are passed by value.

- Single-precision floating point numbers (normal variables in compiled BASIC) require two words. Single-precision values are returned by functions in the ES and BX registers. ES contains the high order word and BX contains the low order word. Single- precision parameters are passed by value.

- Double-precision floating point numbers require four words. Double-precision values are not returned by functions in registers. Rather, the address of the value is returned in the AX register. Double-precision parameters are passed by reference. That is, the address (16-bit) of the variable is passed.

- String descriptors require three words. The first two words comprise a pointer that addresses the data portion of the string. The third word contains the length of the string. String descriptors are not returned by functions in registers. Rather, the address of the string descriptor is returned in the AX register. String parameters are passed by reference. That is, the address (16-bit) of the string descriptor is passed.

SECTION 5
BASIC ERROR MESSAGES

INTRODUCTION

This section lists the error messages which you may encounter when using the BASIC compiler. It describes each error and suggests some possible actions that you may take to solve the problem. When an error is detected, a message is output to the list file and displayed on the video screen. The message is displayed in the following format:

```
*** Error after label nnnn:
```

where nnnn is the name of the label at which the error occurred.

Similarly, when the compiler issues a warning, a message is output to the list file and displayed on the screen.

There are three types of error messages: run-time errors, compile-time errors, and fatal compiler errors.

The first section lists the run-time errors, which include an error number as part of the message.

The messages are listed in order of their error number. A cross reference is also provided which lists the error messages alphabetically so that you will be able to locate the message quickly.

The compile-time errors and the fatal compiler errors are listed alphabetically in separate sections.

Run-Time Errors

Number 1

Message

NEXT without FOR

A variable in a NEXT statement does not correspond to any variable in a FOR statement.

Check to make sure that each NEXT statement references a FOR statement.

Check that the index variable of the FOR statement matches the variable in the NEXT statement.

Check that the loop was not entered after the FOR statement.

Number 2

Message

Syntax Error

The indicated program line contains an error. This error does not allow this line to be executed.

Check the spelling of the BASIC command used in the line.

Make sure the punctuation used with the BASIC statement is correct.

Check the use of spaces in the BASIC statement.

Make sure that the number of open parentheses "(" is equal to the number of closed parentheses ")".

Check that reserved words have not been used as variable names.

Number
3

Message

RETURN without GOSUB

The program has encountered a RETURN statement for which there was no previous GOSUB statement.

Check that the RETURN is used to end a valid subroutine.

Check that the line number specified in the GOSUB statement is correct.

Trace the program to ensure that the subroutine has not been entered accidently.

Number
4

Message

Out of data

A READ statement in the program has been executed but there are no remaining DATA statements.

Trace the program to check the sequence of READ statements.

If the READ statements occur in program loops, check that the loop executes the desired number of times.

If a trailing data value is used to end the reading of values, test that the exit condition is met.

Number 5

Message

Illegal function call

The value passed to a function is not in an acceptable range.

Check that the values used within the parenthesis portion of the function are acceptable entries.

If the program calls a user defined function, make sure the function has been properly defined before it is called.

Number 6

Message

Overflow

The result of a calculation is too large to be represented in BASIC number format.

Check the order of the arithmetic operators used in the calculations.

Make sure that any variable used in the calculation has the proper intermediate value. Print these variables before the calculation.

Number 7

Message

Out of memory

A program is too large, has too many FOR. . . NEXT loops or GOSUB's, too many variables, or expressions that are too complicated.

Check to see that FOR. . . NEXT loops are not nested unnecessarily. Check the entry to subroutines to ensure that subroutines are not nested unnecessarily.

If the program is too large, consider using subprograms with the CHAIN command.

Number 8

Message

Undefined line

A line has been referenced in a BASIC statement, but the line number does not exist.

Check the line number specified in GOTO, GOSUB, or IF. . .THEN. . .ELSE statements.

Number 9

Message

Subscript out of Range

An array element is referenced with a subscript that does not agree with the dimensions of the array.

Check that the dimensions of the array are correct.

If the array element is a program calculated variable, check the intermediate value of the subscript before referencing the array.

Number 10

Message

Duplicate Definition

Two DIM statements are given for the same array, or a DIM statement is given for an array that has already been used with the default dimension.

Check the names of arrays used in different program segments.

If a CHAIN command has been used, check the DIM statements in each subprogram.

Number 11

Message

Division by zero

A division by zero has been encountered in an expression.

Check intermediate values in calculations.

Check the order of the arithmetic operators used.

Number 12

Message

Illegal direct

A statement that is illegal in direct mode has been entered as a direct mode command.

Check that INPUT, DEF FN, GET, or DATA statements have not been used in a direct mode statement.

Number 13

Message

Type mismatch

A string variable name is assigned a numeric value or a numeric variable is assigned a string value.

A function that expects a numeric value is given a string value or vice versa.

If a form of the DEF command has been used, check variable names.

Number 14

Message

Out of string space

String variables caused BASIC to exceed the amount of free memory remaining.

Check to see if some of the string variables created could be formed using BASIC string functions.

Number 15

Message

String too long

An attempt is made to create a string more than 32K characters long.

Check any string manipulations performed, especially any concatenations.

Number 16

Message

String formula too complex

A string expression is too long or too complex.

Break the string expression into smaller expressions.

Number 17

Message

Can't continue

An attempt is made to continue (CONT) a program that halted due to an error, was modified during a break in execution, or does not exist.

If the program was edited during the break in execution, rerun the entire program.

Make sure the program is still in memory.

Number 19

Message

NO RESUME

An error trapping routine has been entered but it contains no RESUME statement.

Check that the flow in the error trapping routine does not branch illegally.

Number 20

Message

RESUME without error

A RESUME statement has been encountered before an error trapping routine was entered.

Trace program to ensure that the error handling routine has not been entered accidently.

Number 21

Message

Unprintable error

An error message is not available for the error condition that exists.

Check that any ERROR has a properly defined error code.

Number	Message
22	Missing operand

An expression contains an operator with no operand following it.

Check the placement of parentheses in statements using arithmetic and/or relational operators.

Make sure that no operators have been used as variable names.

Number	Message
23	Line buffer overflow

An attempt has been made to input a line that has too many characters.

If the INPUT function has been used, check the number of characters in the entered line.

Number 26

Message

FOR without NEXT

A FOR statement has been encountered without a matching NEXT statement.

Check to make sure that each FOR statement is paired with a NEXT statement.

Check that the index of the FOR statement matches the variable in the NEXT statement.

Number 29

Message

WHILE without WEND

A WHILE statement is encountered without a matching WEND.

Check that each WHILE statement is paired with a WEND statment.

Number 30

Message

WEND without WHILE

A WEND statement is encountered without a matching WHILE.

Check that each WEND statement is paired with a WHILE statement.

Check to see that the WHILE statement was not accidently bypassed in the program.

Trace the execution of the program.

Number 31

Message

Invalid parameter

The value (parameter) does not evaluate to an integer or single precision number.

Check to make sure that the numbers you are using were input as numerics.

Make sure numeric values have not been converted to their string equivalents.

Number 32

Message

Unknown procedure

The procedure name following a CALL command is unknown to BASIC.

Check the spelling of the called procedure.

Check punctuation of CALL statement.

Number 33

Message

Too much parameter data

BASIC passed more parameter data than the called procedure requires.

Check the elements of the CALL statement.

Check the punctuation in the CALL statement.

Number
34

Message

Too little parameter data

BASIC passed less parameter data than the called procedure requires.

Check the elements of the CALL statement.

Check the punctuation in the CALL statement.

Number
35

Message

Procedure called as function

A non-value returning procedure was invoked in an expression.

Use the CALL command to invoke the procedure.

Number 36

Message

Function called as procedure

A value returning procedure was invoked with a CALL statement.

Invoke the procedure in an expression.

Number 37

Message

Nesting limit exceeded

In using procedures as parameters of other procedures, you nested too deeply.

Assign the results of one of the procedures to a variable and use the variable as a parameter in the other procedure.

Number 38 — Message

Pointer required

The BASIC command or function requires a pointer variable as a parameter.

Check that a pointer has been created with a BASIC statement.

Trace program to be sure that you did not branch around the pointer creation.

Number 39 — Message

System error

Check SysErc( ) for system error number. System errors can be found in the BTOS manual.

Number 50

Message

Field overflow

When using a random access file, the FIELD statement is trying to allocate more space (bytes) than is allowed.

Check that the FIELD statement allocates the proper space for each variable.

In the OPEN statement, change the record length option.

Number 51

Message

Internal error

An internal malfunction has occurred on the specified disk.

If this error occurs frequently, change the disk.

Number 52

Message

Bad file number

A statement references a file with a file number that is not open.

Check that an OPEN statement has been executed.

Number 53

Message

File not found

A KILL or OPEN statement references a file that does not exist on the current disk.

Check the disk file contents.

Check to ensure that the file in question is not a restricted access file.

Number 54

Message

Bad file mode

An attempt has been made to use PUT, GET or LOC with a sequential file. These statements are restricted to random files.

A file was OPENed with a file mode other than I, O, A, R, S, or B.

Check the OPEN statement in question for file mode specification.

Number 55

Message

File already open

A sequential output mode OPEN was used for a file that is already open.

Make sure that sequential files in I or A Mode have been closed prior to being reopened in O mode.

Check that a KILL command was not used with an open file.

Number 57

Message

Disk I/O error

An input/output error occurred on a disk.

Check to see if the floppy disk you are using is damaged.

Make sure that the disk is write enabled if you are writing to a floppy disk.

Number 58

Message

File already exists

The file name specified in a NAME statement is identical to a file name already in use on the disk.

Choose a new file name.

Number 61

Message

Disk full

All disk storage space is in use.

If using a floppy disk, change to a new disk.

If using the Winchester disk, output to a floppy or delete files no longer desired from the Winchester.

Number 62

Message

Input past end

An INPUT statement has been executed after all the data in a sequential file was input.

Check to make sure that the file from which data are input is not an empty (null) file.

Use the EOF function when you are not sure of the size of a sequential file.

Number 63

Message

Bad record number

In a PUT or a GET statement, the record number is greater than the maximum allowed (32767) or equal to 0.

If the record number is a variable assigned a value during program execution, check the values of this variable prior to the PUT or GET statement.

Number 64

Message

Bad file name

An illegal form has been used for the file name.

Check that the file name is not a BASIC reserved word.

Number 66

Message

Direct statement in file

A statement without a line number has been encountered while loading an ASCII format file. The LOAD operation has been terminated.

Check that each BASIC statement is preceded by a line number.

Check to see that the file being loaded is not a data file.

Number 67

Message

Too many files

An attempt is made to create a new file when all 255 directory entries are full.

Re-execute program with a new disk mounted.

Number 68

Message

Not readable as a random file

An attempt to open a random file has failed because the file contains variable length records.

Check to make sure that the disk file is a random access file.

Number 69

Message

Record size mismatch

An attempt to open a random file has failed because the record size specified in the OPEN statement is different from the record size specified when the file was created.

Check the record size specified for the original random file and correct the record length specification in the OPEN statement.

Number 70

Message

Record does not exist

An attempt to read a record from a random file has failed because the record does not exist.

Check the GET statement to make sure that the record specified is a record number that exists in the random file.

Check the number of records in the random file using the LOF function.

Number 71

Message

Not a random file

An attempt to open a random file has failed because the file is not in the proper format.

Check to make sure that the disk file is a random access file.

Number 72

Message

Malformed record

An attempt to read or write a record in a random file has failed because the file structure cannot be verified.

Use the MAINTAIN FILE utility to check the data file.

Number 73

Message

Read/write beyond EOF

An attempt to open a random file has failed. This is either because the file is not in Standard Access Method format or because an internal error has occurred.

Check to make sure that the file being opened was created as a random access file.

Cross Reference to Run-Time Errors

Message	Number
Bad file mode	54
Bad file name	64
Bad file number	52
Bad record number	63
Can't continue	17
Direct statement in file	66
Disk full	61
Disk I/O error	57
Division by zero	11
Duplicate Definition	10
Field overflow	50
File already exists	58
File already open	55
File not found	53
FOR without NEXT	26
Function called as procedure	36
Illegal direct	12
Illegal function call	5
Input past end	62
Internal error	51
Invalid parameter	31
Line buffer overflow	23
Malformed Record	72
Missing Operand	22
Pointer required	38

Message	Number
Nesting limit exceeded	37
NEXT without FOR	1
NO RESUME	19
Not a random file	71
Not readable as a random file	68
Out of data	4
Out of memory	7
Out of string space	14
Overflow	6
Procedure called as function	35
Read/write beyond EOF	73
Record does not exist	70
Record size mismatch	69
RESUME without error	20
RETURN without GOSUB	3
String formula too complex	16
String too long	15
Subscript out of Range	9
Syntax Error	2
Too little parameter data	34
Too many files	67
Type mismatch	13
Unknown procedure	32
WEND without WHILE	30
WHILE without WEND	29
Undefined line	8
Unprintable Error	21

COMPILE TIME ERRORS

Actual parameter list inconsistent
: The number of type of actual parameters in an external user-defined function invocation differs from that of a previous invocation.

Array referenced as scalar
: A variable that was previously referenced or dimensioned as an array has been referenced without subscripts. (Scalar means "nonarray".)

Array too big
: The size of an array exceeds 64k bytes. Arrays can contain 32,767 integer elements, 16,383 single-precision elements, or 8,191 double-precision elements. This error can occur when a non-BASIC procedure invocation is compiled as an array reference because a BasGen file was not used or did not contain the procedure in question.

Compiled CHAIN MERGE differs from interpreter
: The compiler does not support the MERGE option of the CHAIN statement (see the "CHAIN" subsection in the "Command and Statements" section above).

COMMON not supported
: The compiler does not support the COMMON statement. However, the effect of COMMON can be achieved by using the [Common data?] compiler option (see the "Common Data" subsection in the "General Information" section above).

Data segment overflow
: The cumulative sizes of all program variables exceeds 64k bytes.

Double precision parameter required
: This function call requires double-precision parameters.

Duplicate array definition
: You have entered two DIM statements for the same array, or a DIM statement for an array that has already been used with the default dimension.

Duplicate fn name
: A user-defined function cannot be redefined. It can appear in only one DEF statement.

Duplicate formal parameter name
: Each formal parameter in a single user-defined function definition must have a unique name.

Duplicate non-BASIC procedure
: The BasGen file contains two or more non-BASIC procedure interfaces that have the same name.

Duplicate OPTION BASE
: Only one OPTION BASE statement can occur in a BASIC program.

ERASE not supported
: The compiler does not support erasing of arrays.

ERASE parameter not an array
: The parameters of an ERASE statement must be array variables.

Expected ; not found
: A semicolon (;) must follow the prompt string in an INPUT or LINE INPUT statement, or the string expression in a PRINT USING statement.

Expected = not found
: This is a syntax error in an assignment statement or a user-defined function definition. The equal sign (=) must appear in this statement.

Expected AS not found
: The keyword AS was not found in the FIELD or NAME statement.

Expected B or W not found
: PEEK, POKE, INP, and OUT require a "B" for byte operations or a "W" for word operations.

Expected BASE not found
: This is a syntax error in an OPTION BASE statement.

Expected close parenthesis not found
This is a syntax error in an expression. A close parenthesis is missing.

Expected comma not found
This is a syntax error in an expression. A comma is missing.

Expected fn name not found
This is a syntax error in a user-defined function definition. An identifier beginning with "fn" must follow DEF.

Expected formal parameter not found
This is a syntax error in a user-defined function definition. If a user-defined function requires no parameters, do not use parentheses.

Expected GOTO not found
This is a syntax error in an IF, ON...GOTO, or ON ERROR GOTO statement. The keyword GOTO is missing.

Expected INPUT not found
This is a syntax error in a LINE INPUT statement. The keyword INPUT is missing.

Expected open parenthesis not found
This is a syntax error in an expression. An open parenthesis is missing.

Expected print item not found
This is a syntax error in a PRINT USING or LPRINT USING statement. An expression to print is missing.

Expected scalar variable not found
The loop variable of the FOR statement must be a scalar (nonarray). It cannot be a constant or an array variable.

Expected target label not found
: This is a syntax error after GOSUB, GOTO, or THEN. A statement label must appear to indicate to which statement control must be transferred.

Expected THEN not found
: This is a syntax error in an IF statement. The keyword THEN must follow the logical expression.

Expected TO not found
: This is a syntax error in a FOR statement. The keyword "TO" must appear.

Expected variable not found
: This is a semantic error in a DIM or FOR statement. A variable is required.

Expected variable reference not found
: The statement requires an address expression. That is, an expression that can evaluate the address of a variable. A scalar or an array variable is an address expression; a constant or an arithmetic expression is not.

Expression is missing an operand
: This is a syntax error in an expression. An operand is missing.

Expression too big
: An expression is too long or complex to be compiled. Rewrite the expression as several simpler expressions.

Fn defined recursively
: A user-defined function cannot reference itself within its own definition.

Function called as procedure
: A non-BASIC procedure that returns a value cannot be invoked with the CALL statement. The procedure must be invoked in an expression.

Integer parameter required
: This function call requires an integer parameter.

Invalid BasGen file
: The file specified in the "[BasGen file]" field during compilation is not of the expected format. The BasGen file is the object module produced by assembling BasGen.Asm.

Invalid CALL target
: The procedure name that follows the CALL statement here is not a non-BASIC procedure. Non-BASIC procedures must be declared in the BasGen file.

Invalid character
: A character appears in an identifier or numeric constant that is not allowed.

Invalid data item
: Data items in the DATA statement must be either a quoted string, an unquoted string, or a numeric constant.

Invalid data type for non-BASIC procedure
: A non-BASIC procedure cannot return a string or double-precision value.

Invalid dimension
: Upper bounds in DIM statements must be integer constants. Variables or noninteger values cannot be used as array bounds.

Invalid fn name
: This is a syntax error in a user-defined function definition. User-defined function names must begin with "fn".

Invalid label
: This is a syntax error in a statement or target label. A label must be in the range 0 to 65535.

Invalid numeric constant
: This is a syntax error in a numeric constant. An invalid character appears in the middle of a number.

Invalid parameter to non-BASIC procedure
A non-BASIC procedure invocation cannot contain a double-precision value as an actual parameter.

Invalid use of file number character
The character number sign (#) is valid only when it appears before a file number in a statement or immediately after a double-precision constant, variable, or user-defined function.

Invalid range of letters
The range of letters in a DEFINT/SNG/DBL/STR statement must be within the range A through Z. The second letter must also be greater than or equal to the first.

Label out of order
Statement labels must appear in ascending order.

Matching NEXT not found
A FOR statement exists that does not have a matching NEXT statement to mark the end of the FOR block.

Matching WEND not found
A WHILE statement exists that does not have a matching WEND statement to mark the end of the WHILE block.

Misplaced assignment operator
The assignment operator is out of place.

Misplaced close parenthesis
A close parenthesis is out of place.

Misplaced comma
A comma is out of place.

Misplaced open parenthesis
An open parenthesis is out of place.

NEXT/FOR mismatch
The identifier following the NEXT statement must match the loop variable identifier in the corresponding FOR statement.

NEXT without FOR
A NEXT statement must be preceded by a corresponding FOR statement.

Not enough parameters
An intrinsic or user-defined function has been called with fewer actual parameters than required.

Not enough subscripts
An array has been referenced using fewer subscripts than required.

Numeric overflow
A constant in this statement is outside the range limits for numbers. The integer range is -32768 to 32767. The single-precision and double-precision range is approximately -1.7E+38 to 1.7E+38.

OPTION BASE after array reference
The OPTION BASE statement cannot appear after an array is referenced or dimensioned.

OPTION BASE not 0 or 1
This is a syntax error in an OPTION BASE statement. The number following the words OPTION BASE can only be 0 or 1.

Out of memory
The compiler has run out of memory for the symbol table. The program cannot be compiled unless its size is reduced or more memory is made available to the compiler. The compiler can use up to 64k bytes for its symbol table.

Procedure called as function
A non-BASIC procedure that does not return a value has been invoked in an expression. The procedure can only be invoked with the CALL statement.

RESTORE <line number> not supported
: The compiler does not support the <line number> option of the RESTORE command. A simple RESTORE occurs.

RESUME not allowed with current options
: RESUME, RESUME 0, or RESUME NEXT statements are not allowed if the [No RESUME info?] or [No CHAIN/ERL info?] compiler options are used.

Scalar referenced as array
: An identifier that was previously referenced as a scalar (or nonarray) has been referenced as an array. Scalars and arrays cannot share the same identifier.

Single-precision parameter required
: This function call requires a single-precision parameter.

Statement not recognized
: The compiler does not recognize this statement as a valid BASIC statement.

String constant not terminated
: The end of file has been reached within a string constant. Check that all string constants are terminated with the double-quote character.

String parameter required
: This function call requires a string parameter.

String type mismatch
: A string has been used when a numeric is required, or a numeric has been used when a string is required.

Subscript above upper bound
: An array subscript is greater than the upper bound.

Subscript below lower bound
: An array subscript is less than the lower bound.

Syntax error after DELETE
: This is a syntax error in the CHAIN statement. The line number range after the DELETE option is invalid.

Too little parameter data
: A non-BASIC procedure invocation is invalid. The actual parameter list contains less data than is required.

Too many parameters
: An intrinsic or user-defined function has been called with more actual parameters than required.

Too many subscripts
: An array has been referenced using more subscripts than needed.

Too much parameter data
: A non-BASIC procedure invocation is invalid. The actual parameter list contains more data than is required.

Type mismatch
: The types of operands in this statement are not the same. In the SWAP statement, the two operands must be of the same type.

WEND without WHILE
: This WEND statement has no matching WHILE statement preceding it.

FATAL COMPILER ERRORS

The following errors that occur while compiling will force compilation to be terminated.

Cannot access temporary file
: A file error occurred while reading or writing the temporary file, Proto.Tmp, created by the compiler.

Cannot read BasGen file
: A file error occurred while reading the BasGen file.

Cannot read source file
: A file error occurred while reading the BASIC source file.

Cannot read file <filename>
: A file error occurred while attempting to open an input file.

Cannot write file <filename>
: A file error occurred while attempting to open an output file.

Cannot write list file
: A file error occurred while writing the list file.

Cannot write object file
: A file error occurred while writing the object file.

Invalid Yes/No parameter
: Your input to a field in the BASIC Compile form was invalid. Enter 'y', 'yes', 'n', or 'no'.

No source files specified
: You did not enter a source file name in the "Source files" field of the BASIC Compile command form. Enter a valid BASIC source file name.

Out of label table
: Your BASIC source program requires more compiler generated labels than can be accommodated. The compiler generates labels while compiling FOR, WHILE, IF and ON GOTO statements. To compile the program, reduce its size.

Symbol table full
: Your BASIC source programs contain more variables and constants than can be accommodated. Check the symbol table usage statistic in the list file of a previous successful compilation. If 64k bytes of memory is available for the symbol table, you must reduce the size of your program. Otherwise, increase the amount of memory available to the BASIC compiler.

Unable to allocate memory
: BTOS error while allocating short-lived memory.

Unable to deallocate memory
: BTOS error while deallocating short-lived memory.

APPENDIX A
CONTROL OF THE VIDEO DISPLAY

CONTROL OF THE VIDEO DISPLAY

This appendix describes how to control the video display by using a series of special commands. B 20 BASIC can control the following video display features:

- character attributes (reverse video, underlining, half brightness, and blinking)
- screen attributes (half brightness and reverse video)
- cursor positioning and visibility
- line scrolling
- pausing between frames of information
- the keyboard LED indicators.

In addition, B 20 BASIC allows you to:

- fill-in various size rectangles with a specified character
- erase information in lines or frames.

Each of these features is described in this section. First the statement and an explanation of the statement are presented, then an example of the statement is provided.

CONTROLLING CHARACTER ATTRIBUTES

B 20 BASIC allows you to display characters on the screen using:

- reverse video
- underlining
- half-brightness
- blinking

Each of these features may be used alone or in combination with one or more other features. For example, B 20 BASIC allows you to display characters that are in half-brightness and underlined.

To control the display of characters, a special form of the CHR$ function is used. All character attributes are controlled by entering CHR$(255) plus specified letters to control the video display. Table A-1 gives a summary of the character attribute controls. An X in a column of the table indicates the character attribute that is activated by each entry.

TABLE A-1. CHARACTER ATTRIBUTE CONTROL ON THE B 20

	BLINKING	REVERSE VIDEO	UNDERLINING	HALF-BRIGHTNESS
CHR$(255) + "AA"				
CHR$(255) + "AB"				X
CHR$(255) + "AC"			X	
CHR$(255) + "AD"			X	X
CHR$(255) + "AE"		X		
CHR$(255) + "AF"		X		X
CHR$(255) + "AG"		X	X	
CHR$(255) + "AH"		X	X	X
CHR$(255) + "AI"	X			
CHR$(255) + "AJ"	X			X
CHR$(255) + "AK"	X		X	
CHR$(255) + "AL"	X		X	X
CHR$(255) + "AM"	X	X		
CHR$(255) + "AN"	X	X		X
CHR$(255) + "AO"	X	X	X	
CHR$(255) + "AP"	X	X	X	X

For example, the following statement is used to activate the underlining of a character or characters:

```
120 PRINT CHR$(255)+"AC"
```

All PRINT statements after this statement underline the information displayed. To turn off the underlining option, enter:

```
PRINT CHR$(255)+"AZ"
```

or

```
PRINT CHR$(255)+"AA"
```

Notice that each time a character attribute is activated, the previous character attribute control is ignored. To use the character attributes in combination, refer to Table A-1 for the proper control characters.

CONTROLLING CHARACTER ATTRIBUTES ON A B 26

B 26 machines support two additional character attributes, **bold** and struckthrough, in addition to the four supported by the B 22 and B 21.

In video byte streams, all six character attributes may be set in any combination for subsequent characters by use of the following multi-byte escape sequence:

```
OFFh, 'B', mask
```

where "mask" is an 8-bit binary value designating some combination of the six attributes, as follows:

Bit	Hex Value	Attribute
0	01h	Half Bright
1	02h	Underline
2	04h	Reverse Video
3	08h	Blinking
4	10h	Bold
5	20h	Struck-through

Note that this mask byte has the same format used by PutFrameAttrs and also corresponds to the actual attribute byte stored in the video map itself, and interpreted by the video controller hardware.

The former style of escape sequence for controlling character attributes was:

OOFh, 'A', char

where "char is and ASCII character 'A' through 'P'. This scheme has been retained for B26 compatibility, but has not been extended to include bold and struck-through.

CONTROLLING SCREEN ATTRIBUTES

B 20 BASIC allows you to control the entire screen of your computer. Using a special form of the CHR$ function, you can reverse the entire screen or control the brightness of the information displayed. Unlike the character control features, the screen control attributes may be used together. You may use the appropriate command to reverse the video of the entire screen and then change the reversed display to half-brightness using a second command. The screen attribute controls also allow you to turn off the reverse video and half-brightness features.

To reverse the video on the screen, the statement used is:

```
CHR$(255)+"RN"
```

When this statement is executed, the entire screen is reversed. To return the screen to its original format, use the statement:

```
CHR$(255)+"RF"
```

Both of these screen attribute control statements must be executed in a PRINT statement before they are activated.

To set the screen at half-brightness, the statement used is:

```
CHR$(255)+"HN"
```

When this statement is executed, the entire screen displays information at half its original brightness. To return the screen to its original brightness, use the statement:

```
CHR$(255)+"HF"
```

Again, both of these statements must be executed in a PRINT statement before they are activated.

CONTROLLING CURSOR POSITION AND VISIBILITY

B 20 BASIC allows for the control of both cursor position and its visibility. A special form of the CHR$ function is used for this purpose.

To control the cursor position the following statement is used:

CHR$(255)+"C" + CHR$(column position of cursor)+

CHR$ (row position of cursor)

Both the column and the row positions are integers and must be within the range of your B 20 system screen size. If the row column number you enter is greater than the screen size of your system, the control of the cursor is ignored.

To deactivate the display of the cursor, the following statement is used:

CHR$(255)+"VF"

To make the cursor visible, use the statement:

CHR$(255)+"VN"

CONTROLLING LINE SCROLLING

B 20 BASIC allows you to scroll down a portion of a screen and fill the upper portion of the screen with blanks. It also allows for scrolling up a portion of the screen and filling the lower portion with blanks.

To scroll down the screen, the following command is used:

CHR$(255)+"S"+CHR$(first line)+CHR$(last line)+CHR$(count)+"D"

When this command is executed, the screen scrolls down beginning at the line number entered as the first line and continues up to the line number entered as the last number. The number of lines scrolled is the number entered for the variable count in this command. The top "count" lines of the screen are left blank.

To scroll up the screen the following command is used:

CHR$(255)+"S"+CHR$(first line)+CHR$(last line)+CHR$(count) +"U"

When this command is executed the screen scrolls up beginning at the last line and continuing up to the first line.

EXAMPLE

```
PRINT CHR$(255)+"S"+CHR$(1)+CHR$(10)+CHR$(9)+"D"
```

This statement scrolls down the screen and leaves the top 9 lines of the screen blank.

CONTROLLING PAUSES BETWEEN FRAMES

With B 20 BASIC you can control the pausing between full screens of information. When the pausing feature is activated, the computer pauses and displays the following message when information is about to be scrolled off the screen:

Press NEXT PAGE to continue

The display of information is continued by pressing the NEXT PAGE key.

To activate pausing between frames, use the following statement:

```
CHR$(255)+"PN"
```

Once this statement has been executed the pause message is displayed as each screen fills with information. The NEXT PAGE key must be pressed to continue the display of information.

To discontinue the display of the pause message, the following statement is used:

```
CHR$(255)+"PF"
```

CONTROLLING THE KEYBOARD LED INDICATORS

B 20 BASIC allows you to light the LED indicators on the keyboard. To light one or more of the LED indicators, a special form of the CHR$ function is used. The form of the statement to turn-on an LED indicator is:

CHR$(255)+"I number or letter N"

The number or letter controls which of the LED indicators is lighted. Table A-2 shows the code for controlling each of the LED indicators.

TABLE A-2. Control Codes for LED Indicators

CODE	KEY
1	F1
2	F2
3	F3
8	F8
9	F9
0	F10
T	OVERTYPE

To turn off an LED indicator the following statement is used:

CHR$(255)+ "I number or letter F"

The number or letter is the code for the LED indicator as listed in Table A-2.

EXAMPLE

```
PRINT CHR$(255)+"I1N"
```

This statement turns on the LED on the F1 key.

FILLING A RECTANGLE

To fill-in a rectangle using B 20 BASIC, a special form of the CHR$ function is used. The following statement is used to fill in a rectangle of a specified size:

```
CHR$(255)+"F"+"any character"+ CHR$(column number)+

CHR$(row number)+CHR$(width)+ CHR$(height)
```

Any standard character may be used and the rectangle of the specified width and height may be placed in any column and row position on the screen.

EXAMPLE

```
PRINT CHR$(255)+"F"+"*"+ CHR$(10)+ CHR$(10)+CHR$(40)+ CHR$(5)
```

This statement displays a rectangle, the top left edge of which begins in row 10, column 10. The rectangle displayed is 40 characters wide and 5 characters high. In this example, a rectangle of asterisks (*) is displayed.

ERASING TO THE END OF THE LINE OR FRAME

B 20 BASIC allows you to erase to the end of the line or to the end of a frame. To erase to the end of a line, use the following statement:

```
CHR$(255)+"EL"
```

To erase to the end of a frame, the following statement is used:

```
CHR$(255)+"EF"
```

Erasing turns off all previously specified character attributes.

DISPLAYING SPECIAL CHARACTERS LITERALLY

B 20 BASIC enables you to display literal characters, and to also disable the literal display of characters. To enable the literal display of characters use the following statement:

```
CHR$(255) + "LN"
```

To disable the literal display of special characters, the following statement is used:

```
CHR$ + "LF"
```

DIRECTING VIDEO DISPLAY OUTPUT: THE "X" COMMAND

Format: CHR$(255) + "X" + CHR$(<frame>)

where <frame> is 0 to 7.

Purpose: To direct video output to frame <frame> of the video display.

Remarks: When BASIC is invoked, frame 0 is defined as the entire video display and no other frame exists. This command is meaningful only after you call the Video Display Manager's InitVidFrame operation to define new frames.

APPENDIX B
CALLING NON-BASIC PROCEDURES

OVERVIEW

BASIC can directly call procedures that are compiled or assembled into standard object module format. Hence, you can access procedures written in Pascal, FORTRAN, or assembly language. Using this facility, you can directly call the BTOS Operating System, Burroughs software products such as Forms, ISAM, and Sort/Merge, and your own non-BASIC procedures.

You invoke a non-BASIC procedure by its name. If the procedure does not return a value, use the CALL statement. If the procedure returns a value, the procedure name can appear in any numeric expression. Non-BASIC procedures can return byte, integer, or single-precision values. In addition, calls on value-returning procedures can be nested as parameters to other non-BASIC procedures.

Parameters are passed by value and must evaluate to integers or single-precision numbers. To pass a string or array, pass a pointer to the first byte or element of the structure. Pointers are stored as single-precision numbers; hence, they are valid parameters. You construct a pointer to a BASIC variable using the PTR function (see Section 3).

To call non-BASIC procedures, create a run file (Basic.Run) which contains the BASIC interpreter, a lookup table (rgProcedures), and the non-BASIC procedures. The process of creating a new Basic.run is described in detail in "Configuring BASIC" below.

BASIC provides several checks to detect incorrect procedural calls. For example, the number of bytes of parameter data passed by BASIC must match the number required by the procedure. See the "Error Detection" section later in this appendix for more details.

The following BASIC statements demonstrate calls to non-BASIC procedures. In the first example, the program calls Initialize, passing the contents of integer i%, and NoOp, passing no arguments.

```
100 CALL Initialize(i%)
110 CALL NoOp
```

In the second example, a call is made to the BTOS CloseFile operation, passing the integer constant 7. The returned status code is stored in the integer variable erc%.

```
100 erc% = CloseFile(7)
```

In the third example, a call is made to the BTOS OpenFile operation, and the returned status code is stored in erc%. The first parameter is a pointer to the BASIC variable fh%. OpenFile uses this pointer to store back the file handle of the opened file. The second parameter is the filename. The next parameter is the password. The final parameter is an integer whose two bytes represent the ASCII characters "mm", for mode modify.

```
100 fh% = 0
110 Filespec$="TestFile"
120 Password$-""
130 mode%=&h6d6d
140 erc% = OpenFile(PTR(fh%), PTR(Filespec$),
    LEN(Filespec$),
    PTR(Password$),
    LEN(Password),mode%)
```

In the fourth example, PrimeNumber returns 255 if the integer parameter is prime, and 0 otherwise. The expression j%+12 is evaluated before it is passed. If the parameter is prime, the THEN clause executes.

```
100 IF PrimeNumber(j%+12) THEN ...
```

The final example constructs a pointer to the beginning of the BASIC data segment. GETSA returns an integer representing the sa (segment address) portion of a pointer; MAKEPOINTER returns the pointer whose ra (relative address) and sa are given as parameters. This algorithm works because the BASIC variable i% is in the BASIC data segment.

```
100 i% = 0
110 pBasicDS!=MakePointer( 0, GetSa(PTR(i%)) )
```

INVOKING VALUE-RETURNING PROCEDURES

Format: <procedure name>([parameter list])

NOTE

Parentheses are always required.

Purpose: To call a value-returning non-BASIC procedure.

Remarks: Procedure invocation transfers program control to a value-returning procedure. The name of the procedure must be in rgProcedures, a lookup table defined in BasGen.Obj.

The procedure can return either a byte, an integer, or a single-precision number. A byte is stored as an integer and ranges in value from 0 to 255. The type of value that the procedure returns is specified by rgProcedures.

Procedure invocations can appear in any numeric expressions including parameter list of other non-BASIC procedures. In this case BASIC restricts the level of nesting to three. For example, flip(flip(flip(1))) is permitted, but flip(flip(flip(flip(1)))) is not.

If the procedure requires no parameters, the parameter must be null.

CAUTION

If you omit the parentheses, BASIC allocates a new variable and returns a value of 0 rather than invoking the non-BASIC procedure.

The "Parameter Passing" section below describes the parameter list in detail.

Example: `100 erc% = CloseFile(fh%)`

PARAMETER PASSING

Parameter List - A parameter list is a sequence of parameters separated by commas. Valid parameters are numeric expressions that evaluate integers or single-precision numbers. You cannot pass strings, arrays, and double-precision numbers directly; however, you can pass pointers to these variables.

BASIC passes parameters by value. If the parameter evaluates an integer, a word is pushed onto the stack. If the parameter evaluates a single-precision number, two words are pushed. The first word pushed represents the most significant bits of the value. If the parameter is a pointer, this corresponds to a segment address.

Modifying BASIC Variables from Non-BASIC Procedures

Because BASIC passes parameters by value, non-BASIC procedures must take pointers as parameters to all data to be modified by the procedure. The example below shows how a BASIC variable to be modified is passed to the OpenFile operation.

```
100 fh% = 0
110 erc% = OpenFile ( PTR(fh%),... )
```

String Parameters

You pass a string by passing a pointer to its first byte. In addition, you normally pass an integer describing the length of the string. The LEN function returns the current length of the string. The example below shows how a string representing a file specification is passed to the OpenFile operation.

```
100 Filespec$ = "(sys)<MyDir>MyFile"
110 erc% = OpenFile(...,PTR(Filespec$),LEN(Filespec$),...)
```

BASIC data segment. GETSA returns an integer representing

You can also pass strings that are to be modified by the non-BASIC procedure. Normally, this requires three parameters: a pointer to the first byte of the string, an integer describing the length of the string, and a pointer to a BASIC variable describing the number of bytes written. Consider the example below.

```
100 s$ = STRING$(32,"")      'Allocates a string
110                           'of 32 blanks
120 cbRet% = 0                'Integer to contain
130                           'the number of bytes
140                           'written
150 CALL AssignString(PTR(s$),LEN(s$),PTR(cbRet%)
160 s$ = LEFT$(s$, cbRet%)
```

s$ is initially a string of 32 blanks. AssignString writes bytes into s$, using the length parameter to prevent writing beyond the string. s$ is then adjusted to its new length, as defined by cbRet%, using the LEFT$ function.

You must execute the final length-adjustment step when using strings in this way to notify BASIC that the length of the string was changed.

Basic stores a string as a four byte pointer and a two byte length, The actual string is located in sData. A reference to the string name is the same as a reference to the pointer and length. This can be used to advantage when calling non-Basic procedures. In the preceding example, line 150 could be written as:

```
150 CALL AssignString(s$,PTR(cbRet%))
```

The reference to s$ is the same as PTR(s$),LEN(s$), but clearer and easier to write.

Array Parameters

You pass an array by passing a pointer to its first element. Arrays typically serve as working areas or buffers. The example below shows how you pass an array as a working area to a Direct Access Method operation.

```
100 DIM Dawa%(31)      'allocates 64bytes
110                    'actually, 32 words
120 erc% = OpenDaFile(PTR(Dawa%[0]),...)
```

Word-Aligned Data

Some non-BASIC procedures, such as OpenDaFile, require word alignment for buffers. Since BASIC does not guarantee word alignment of data, pointers may or may not point to word boundaries. However, you can always create a pointer to word-align data using the BASIC PWA function.

The PWA function returns a pointer to word-aligned data, given a pointer parameter. If the parameter is even (that is, points to a word boundary), the parameter is simply returned; otherwise, a pointer to the next word boundary is returned.

Whenever word alignment is needed, the data area should be at least one byte longer than normally required. This ensures that the data area is large enough in the case where it is addressed by a word-aligned pointer. The example below shows how a word-aligned buffer is passed.

```
100 DIM Buffer%(256)      'allocates 514 bytes
110                       '(only need 512
120                       'bytes)
130 erc% = OpenDaFile(...,Pwa(PTR(Buffer[0])), 512,...)
```

CAUTION

Pointers generated by PWA do not always point to the beginning of a BASIC variable, since BASIC variables are not always word-aligned. If a non-BASIC procedure modifies a BASIC variable using such a pointer, BASIC may not be able to reference the modified data correctly.

PROGRAMMING RECOMMENDATIONS

Follow these recommendations when calling non-BASIC procedures.

Recommendation 1

Allocate all variables before using pointers. When new variables are allocated, BASIC can reorganize memory, invalidating any pointers that were saved.

Allocate variables by assigning initial values. You can also allocate an array with the DIM statement.

Recommendation 2

Do not use the INPUT statement to store a value directly into an array. If you do this, BASIC reorganizes memory, invalidating any pointers that were saved.

INPUT the value into a simple variable, then assign the variable to the array. For example, use:

```
100 INPUT X
110 A[1] = X
```

rather than:

```
100 INPUT A[1]
```

Recommendation 3

Use brackets rather than parentheses to subscript arrays. By using brackets, you improve BASIC performance. You also improve program readability by distinquishing between arrays and value-returning procedures.

ERROR DETECTION

BASIC detects several errors while invoking non-BASIC procedures. These include:

- invalid parameters,
- unknown nonvalue-returning procedures,
- too much or too little parameter data,
- invoking a value-returning procedure with a CALL statement or a nonvalue-returning procedure in an expression, and
- exceeding the nesting limit when using procedures as parameters to other procedures.

Section 5 fully describes these errors.

CONFIGURING BASIC FOR NON-BASIC PROCEDURES

To configure a BASIC program in which non-BASIC procedures are called, you must create a special object file called BasGen.Obj before compiling the BASIC source file. BasGen.Obj is created by assembling BasGen.Asm, a file that contains interface definitions of the non-BASIC procedures. Your BASIC source file is then compiled together with BasGen.Obj to create the program's object file. This object file is linked with any required library files to create the final compiled BASIC run file.

Assembling BasGen.Asm

To create BasGen.Obj, invoke the Assembler by typing "Assemble (or as many letters as are required to make the command unique) in the Executive command form and then press RETURN. (For instructions on using the Assembler, see the B20 System Programmer's Guide/Assembler.)

Fill in the form as follows and then press GO.

```
Assemble
   File                          BasGen.Asm
   [Errors only?]                _____
   [GenOnly, NoGen, or Gen?]     _____
   [Object File]                 _____
   [Error File]                  _____
   [List on pass 1?]             _____
```

You must specify a file name in the "File" field. All other fields are optional.

During assembly, the Assembler asks questions of this type:

Are you calling Forms (y or n)?

Are you calling Sort/Merge (y or n)?

Are you calling ISAM (y or n)?

Are you calling CTOS OpenFile and CloseFile (y or n)?

In this example, you are creating a BASIC file in which Forms procedures are included, so you answer yes to the corresponding question by typing y and then pressing RETURN. You must also answer yes to the question "Are you calling CTOS OpenFile and CloseFile (y or n)?" by typing y, then pressing RETURN. Answering yes to these questions causes the interface definitions for the Forms procedures to be included in BasGen.Obj.

To answer no to the other questions, type n and press RETURN, or just press RETURN.

The file that is created by assembling BasGen.Asm is called BasGen.Obj, which will be compiled together with your source file.

NOTE

To access non-BASIC procedures other than those defined by BasGen.Asm, you can add entries to BasGen.Asm with the Editor or the Executive WRITEone Word Processor. Comments within BasGen.Asm explain how to add an entry. After you have edited BasGen.Asm, assemble it as described above.

The BasGen.Obj file is now created and is ready to be compiled with the source file.

Compiling the Source File and BasGen.Obj

To invoke the BASIC compiler, type "B C" (or as many letters as are required to make the command unique) in the Executive command form (see the "Invoking the Compiler" subsection in Section 5 for instructions on filling in the Basic Compile form).

In the example below, you are compiling a BASIC file that calls the Forms program. Fill in the form as follows and then press GO.

```
Basic Compile
   Source file                  your program name
   [Object file]
   [List file]
   [BasGen file]                BasGen.Obj
   [Suppress main]
   [Public code?]
   [Common data?]
   [Errors only?]
   [No bounds checking?]
   [No RESUME info?]
   [No CHAIN/ERL info?]
   [No code generation?]
```

The object file is created and ready to be linked with the necessary library file to create the executable run file.

Using the Linker to Create the Run File

To invoke the Linker, type "Link" in the Executive command form. (For instructions on using the Linker, see the B20 Linker/Librarian Reference Manual.) In the following example, the object module Tutorial.Obj is linked with the necessary library file. Fill in the form as follows and then press GO.

```
LINK
   Object modules               Tutorial.Obj
   Run file                     Tutorial.Run
   [List file]
   [Publics?]
   [Line Numbers?]
   [Stack Size]
   [Max memory array size]
   [Min memory array size]
   [System build?]
   [Version]
   [Libraries]                  Forms.Lib
   [DS allocation?]
   [Symbol file]
```

NOTE

The Linker automatically searches CTOS.Lib and BasCompiler.Lib, so you do not have to include these files in the "[Libraries]" field during linking.

Figure B-1 illustrates the compiling and linking of a program that calls non-BASIC procedures.

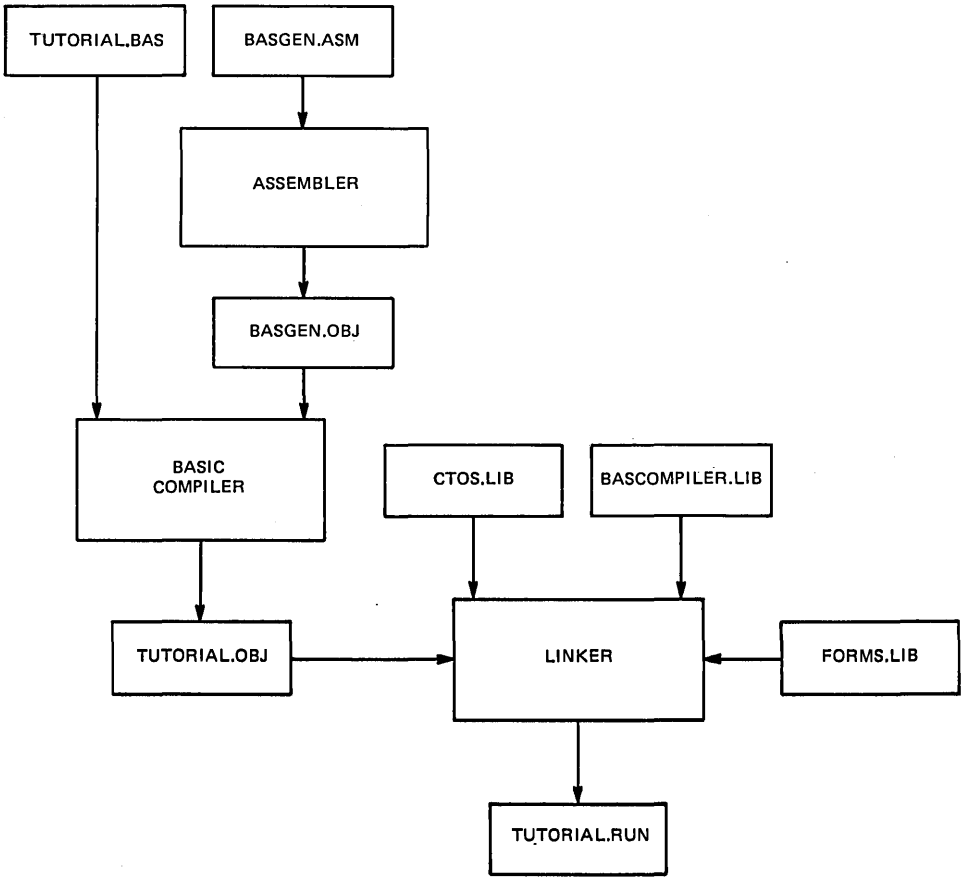

Figure B-1. Compiling and Linking a Program That calls Non-BASIC Procedures

APPENDIX C
CONVERTING PROGRAMS TO B 20 BASIC

If you have programs written in a BASIC other than Burroughs B 20 BASIC, some minor adjustments may be necessary before running them. Here are some specific things to look for when converting BASIC programs.

STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the BASIC statement DIM A$(J).

Some forms of BASIC use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for string concatenation.

In B 20 BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	B 20 BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, then convert as follows:

Other BASIC	B 20 BASIC
A$ (I)=X$	MID$(A$,1,1)=X$
A$(I,J)	MID$(A$,I,J-I+1)=X$

APPENDIX D
DERIVED MATHEMATICAL FUNCTIONS

Mathematical functions that are not intrinsic to BASIC can be calculated as follows.

Function	BASIC Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR (-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1))+SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=EXP(-X)/EXP(X)+EXP (-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP (-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR (-X*X+1))+1/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X +1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

APPENDIX E
DISK INPUT/OUTPUT

Disk input/output procedures for the beginning BASIC user are examined in this Appendix. If you are new to BASIC or if you are getting disk-related errors, then read through these procedures and program examples to make sure you are using all the disk statements correctly.

Wherever a file name is required in a disk command or statement, use a name that conforms to the BTOS requirements for file names (see the subsection on OPEN in Section 3).

PROGRAM FILE COMMANDS

The following commands and statements are used in program file manipulation.

KILL "filename" deletes the file from the disk. "filename" can be a program file, or a sequential or random access data file.

NAME "oldfile" AS "newfile" to change the name of a disk file, execute the NAME statement, NAME "oldfile" AS "newfile". NAME can be used with program files, random files, or sequential files.

Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

```
OPEN
CLOSE
PRINT#
PRINT# USING
INPUT#
LINE INPUT#
WRITE#
EOF
```

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.

   ```
   OPEN "O",#1,"DATA"
   ```

2. Write data to the file using the PRINT# statement. (WRITE# can be used instead.)

   ```
   PRINT#1,A$;B$;C$
   ```

3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.

   ```
   CLOSE #1
   OPEN "I",#1,"DATA"
   ```

4. Use the INPUT# statement to read data from the sequential file into the program.

   ```
   INPUT#1,X$,Y$,Z$
   ```

Figure E-1 below is a short program that creates a sequential file, "DATA", from information you input at the video display.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF  N$="DONE" & THEN  END
30 INPUT "DEPARTMENT ";D$
40 INPUT "DATE HIRED ";H$
50 PRINT #1,N$;",";D$;",";H$
60 PRINT: GOTO 20
```

Figure E-1. Creating a Sequential Data File.

Run the program and enter the data as shown:

```
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

Now look at Figure E-2 below. It access the file "DATA" that was created in the program of Figure E-1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT #1,N$,D$,H$
30 IF RIGHT$(H$,2)="78"  THEN PRINT N$
40 GOTO 20
EBENEZER SCROOGE
SUPER MANN
Fatal Error:  Input past end in line 20
```

Figure E-2. Accessing a Sequential File

The program in Figure E-2 reads, sequentially, every item in the file. When all the data is read, line 20 causes an "Input past end" error message. To avoid getting this error message, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that were written to or read from the file since it was OPENed. A sector is a 128-byte block of data.

Adding Data to a Sequential File

If you have a sequential file residing on disk and later want to add more data to the end of it, you simply open the file in "A" mode and start writing data.

Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, that is, anywhere on the disk; it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

```
OPEN
FIELD
LSET/RSET
GET
PUT
CLOSE
LOC
MKI$
MKS$
MKD$
CVI
CVS
CVD
```

Creating a Random File

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, then the default is 128

   ```
   OPEN "R",#1,"FILE",32
   ```

2. Use the FIELD statement to allocate space in the random buffer for the variables that are to be written to the random file.

   ```
   FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
   ```

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single-precision value, and MKD$ for a double-precision value.

   ```
   LSET N$=X$
   LSET A$=MKS$ (AMT)
   LSET P$=TEL$
   ```

4. Write the data from the buffer to the disk using the PUT statement.

   ```
   PUT #1,CODE%
   ```

Look at the program in Figure E-3. It takes information that is input at the video display and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The 2-digit code that is input in line 30 becomes the record number.

Note: Do not use a FILEDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10  OPEN "R",#1,"FILE"
20  FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30  INPUT "2-Digit code";CODE%
35  IF CODE%=99 & THEN END
40  INPUT "NAME";X$
50  INPUT "AMOUNT";AMT
60  INPUT "PHONE";TEL$:PRINT
70  LSET N$=X$
80  LSET A$=MKS$(AMT)
90  LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Figure E-3. Creating a Random File.

Accessing a Random File

The following program steps are required to access a random file.

1. OPEN the file in "R" mode.

   ```
   OPEN "R",#1,"FILE",32
   ```

2. Use the FIELD statement to allocate space in the random buffer for the variables that are to be read from the file.

   ```
   FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
   ```

NOTE

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

---

3. Use the GET statement to move the desired record into the random buffer.

   ```
   GET #1,CODE%
   ```

4. The data in the buffer can now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single-precision values, and CVD for double-precision values.

```
PRINT N$
PRINT CVS(A$)
```

The program shown in Figure E-4 below accesses the random file "FILE" that was created in Figure E-3. By entering the 3-digit code at the video display, the information associated with that code is read from the file and displayed.

```
10  OPEN "R",#1,"FILE"
20  FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30  INPUT "2-Digit code";CODE%
40  GET #1, CODE%
50  PRINT N$
60  PRINT USING "$$###.##";CVS(A$)
70  PRINT P$: PRINT
80  GOTO 30
```

Figure E-4. Accessing a Random File.

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

The program shown in Figure E-5 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory contains no more than 100 different part numbers. Lines 900 to 960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (lines 270 and 500) to determine whether an entry already exists for that part number.

Lines 130 to 220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
100  ' INVENTORY PROGRAM EXAMPLE
     ON ERROR GOTO 9000
     DEF FNAT$(X,Y)=CHR$(255)+"C"+CHR$(X)+CHR$(Y):
     EL$=CHR$(255)+"EL"
     PRINT FNAT$(5,5),"     INVENTORY FUNCTIONS"
     PRINT fnAT$(5,8)+EL$+"Which file to use";

115  INPUT FN$:IF FN$="" GOTO 115
     GOSUB 1000
     OPEN "R",#1,FN$+".RANDOM",39
     FIELD #1, 1 AS F$, 30 AS D$, 2 AS Q$, 2 AS R$, 4 AS P$

130  PRINT FNAT$(1,5),"     INVENTORY FUNCTIONS":PRINT
     fnAT$(1,8);
     PRINT ,1,"Initialize '"FN$"'."
     PRINT ,2,"Create new inventory."
     PRINT ,3,"Display part inventory."
     PRINT ,4,"Add to stock."
     PRINT ,5,"Subtract from stock."
     PRINT ,6,"Display all items needing reorder."
     PRINT ,7,"Change output device from '";OPD$;"'."
     PRINT ,8,"Return to Executive."

220  PRINT fnAT$(10,20);EL$: INPUT "Function (1-8) ";Function%
     IF  (Function% < 1) OR (Function% > 8) &
     THEN PRINT Function%" IS A BAD FUNCTION NUMBER.": GOTO 220
     PRINT CHR$(255)+"F "+CHR$(0)+CHR$(21)+CHR$(255)+CHR$(255)

230  ON Function% GOSUB 900,250,400,500,600,700,1000,800
     GOTO 130

REM  BUILD NEW ENTRY
250  GOSUB 840
     IF  ASC(F$)<>255 &
     THEN PRINT "Overwrite?";:In$=INPUT$(1): &
         IF In$<>"Y" OR In$<>"y" &
         THEN RETURN
     LSET F$=CHR$(0)
     INPUT "Description      :";Desc$
     LSET D$=Desc$
     INPUT "Quantity in stock:";Q%
     LSET Q$=MKI$(Q%)
     INPUT "Reorder level    :";R%
     LSET R$=MKI$(R%)
     INPUT "Unit price       :";P
     LSET P$=MKS$(P)
     PUT #1,Part%
     RETURN
```

Figure E-5. Inventory. (Page 1 of 3)

```
REM  DISPLAY ENTRY
400  GOSUB 840
     IF  ASC(F$)=255 &
     THEN PRINT "NULL ENTRY.":RETURN
     LPRINT USING "Part Number.....  ###";Part%
     LPRINT       "Description..... ";D$
     LPRINT USING "Quantity on hand.. #####";CVI(Q$)
     LPRINT USING "Reorder level..... #####";CVI(R$)
     LPRINT USING "Unit Price...... $$##.##";CVS(P$)
     LPRINT USING "Total value... $$$$##.##";CVS(P$)*CVI(Q$)
     RETURN

REM  ADD TO STOCK
500  GOSUB 840
     IF  ASC(F$)=255 &
     THEN PRINT "NULL ENTRY.": RETURN
     PRINT D$,: INPUT "Quantity to add";In%
     Q%=CVI(Q$)+In%
     LSET Q$=MKI$(Q%)
     PUT #1,Part%
     RETURN

REM  REMOVE FROM STOCK
600  GOSUB 840
     IF  ASC(F$)=255 &
     THEN PRINT "NULL ENTRY.": RETURN
620  PRINT D$: INPUT "Quantity to subtract";In%
     IF  In%=0 &
     THEN PRINT "NOTHING SUBTRACTED.": RETURN
     Q%=CVI(Q$)
     IF  (Q%-In%)<0 &
     THEN PRINT "Only";Q%;" in stock.":GOTO 620
     Q% = Q% - In%: LSET Q$ = MKI$(Q%)
     IF  Q%<CVI(R$) &
     THEN PRINT "Quantity is now below reorder level of";CVI(R$):&
        LPRINT "Quantity is now below reorder level of";CVI(R$)
     PUT #1,Part%
     RETURN

REM  DISPLAY ITEMS BELOW REORDER LEVEL
700  FOR J = 1 TO 100
       GET #1,J
       IFCVI(Q$)<CVI(R$) &
       THEN LPRINT D$;" Quantity";CVI(Q$);TAB(50);"ReOrder
       level";CVI(R$)
     NEXT J
     RETURN

REM  END PROGRAM
800  END
```

Figure E-5. Inventory. (Page 2 of 3)

```
REM  GET PART NUMBER
840  INPUT "Part number or 999";Part%
     IF  Part%=999 &
     GOTO850
     IF  (Part%<1) OR (Part%>100) &
     THENPRINT "BAD PART NUMBER.": GOTO 840 &
     ELSEGET #1, Part%
850  RETURN

REM  INITIALIZE FILE
900  PRINT "Are you sure?":In$=INPUT$(1)
     IF  In$="N" OR In$="n" &
     THENRETURN
     IF  In$<>"Y" AND In$<>"y" &
     GOTO900
     LSET F$=CHR$(255)
     PRINT "Please wait a moment ";
     FOR J=1 TO 100
       PUT#1,J
       PRINT ".";
     NEXT J
960  RETURN

REM  SET OUTPUT DEVICE
1000 PRINT fnAT$(5,8)+EL$;: INPUT "Output Device Name: ";In$
     IF  In$<>"" &
     THENOPD$=IN$
     erc = DEFLPRINT(OPD$)
     IF  erc<>0 &
     THENERROR 39
     RETURN

9000 IF  ERR = 39 &
     THENPRINT "System error #";ERC;" at line";ERL: &
         PRINT "SysErc() = ";SysErc() &
     ELSEPRINT "Basic error #";ERR;" at line";ERL
     RESUME NEXT
```

Figure E-5. Inventory. (Page 3 of 3)

APPENDIX F
CONFIGURING MEMORY USAGE AND RUN-TIME PARAMETERS

The file BrCnfg.Asm configures the following parameters used by the BASIC comipler run-time system:

1. The number of files that can be opened. The default is 6 files.

2. The size of the buffers used in opening sequential files. The default is 1024 bytes.

3. The size of the memory area used to contain overlays that are read from disk. The default is 16k bytes.

4. The size of the memory area that contains all the file buffers. The default is 12.8k bytes.

5. The size of the memory area that contains string data. The default is 8k bytes.

6. The amount of short-lived memory that is available to the BASIC program. The default is 48k bytes.

7. The flag that instructs the run-time system to clear the video display before the BASIC program is run. The default is to clear the video display.

To change any of the above parameters, edit the BrCnfg.Asm file. Comments within the file explain how to change a parameter.

Use the Assembler to assemble the modified version of BrCnfg.Asm and create the object module BrCnfg.Obj (see the B 20 System Programmer's Guide/Assembler).

Use the Librarian to enter BrCnfg.Obj in the BasCompiler.Lib library and replace the previous version. (For more information, see the B 20 Linker/Librarian Reference Manual.)

BASIC programs that are linked with the updated BasCompiler.Lib use memory as configured by the new version of BrCnfg.Asm.

APPENDIX G
PROGRAMMING HINTS

HINT 1: DIFFERENCES BETWEEN THE BASIC INTERPRETER AND THE BASIC COMPILER

This hint explains the changes required to compile source programs written for the BASIC Interpreter. It also explains the differences between the BASIC Interpreter and the BASIC Compiler. These changes allow the source code to be run with either the interpreter or the compiler.

CHANGES TO BE MADE TO EXISTING PROGRAMS

1. Move DEFINT/SNG/DBL/STR, DEF FN, and DIM statements to the beginning of the program.

2. If two or more modules make reference to a DIM statement, that DIM statement must be included in every source module. This code was conditional for the interpreter.

3. ERASE and CLEAR commands are treated as comments by the compiler.

4 Backward quotes or in-line comment delimiters are illegal in the compiler.

5. Quoted strings cannot be open-ended (i.e., ending quote omitted).

6. LPRINT and PRINT commands cannot be coded as follows:

```
LPRINT "hello"; using "###"; value%; "bye"
```

   The interpreter allows this syntax but the compiler does not.

7. You cannot have more than one module that has shared code and the same line numbers.

8. The maximum total line count of all BASIC programs that are linked together should not exceed 16-20K.

LANGUAGE FEATURES NOT SUPPORTED BY THE COMPILER

Key Directed Features

GO	exit insert mode in BASIC editor
ACTION-CANCEL	stop execution and return to command mode
ACTION-O	suspend video display output
ACTION-S	suspend execution
ACTION-Q	resume execution
CANCEL	abort input and return to command mode
CODE-A	enter the BASIC editor
CODE-R	retype the current line
DELETE	delete the current line
FINISH	exit BASIC

Statements

AUTO	enter a program
CLEAR	zero all variables
COMMON	preserve variables across a CHAIN
CONT	continue execution of the current program
DELETE	remove part of the current program
EDIT	modify the current program
ERASE	remove an array
LIST	display program on the video
LLIST	print program
LOAD	load a program from disk
MERGE	merge a program on disk with the current program
NEW	remove current program/delete all variables
RENUM	renumber current program

RESTORE <line number>
: restore the data stream at the specified <line number> - NOTE: RESTORE is supported by the compiler.

RUN
: execute the current program

SAVE
: store the current program

TRON
: turn statement tracing on

TROFF
: turn statement tracing off

LANGUAGE FEATURES WITH SYNTACTIC AND SEMANTIC DIFFERENCES

The language features described below are implemented in both the interpreter and the compiler. However, there are differences.

CHAIN

In compiled BASIC, the CHAIN statement transfers control from one BASIC module to another. The argument to the CHAIN statement is a string expression specifying a module name. The name of a BASIC module is the same as its filename, without the volume and directory name parts.

For example, suppose a BASIC program consists of the two source files Main.bas and Work.bas. The following statement transfers control from Main.bas to Work.bas.

```
CHAIN "Work.bas"
```

The MERGE, DELETE, and ALL options present in the interpreter are not meaningful in compiled BASIC. The effect of the ALL option can be achieved with the "common data" feature described in a later section.

Continuation Character

The compiler accepts either the MARK or ampersand character as a continuation character; the interpreter accepts only the MARK.

Line Numbers

Line numbers are optional in the compiler; they are required in the interpreter.

Logical Line Length

The length of a logical line is unbounded in the compiler; the interpreter restricts logical line length to 255.

String Constants

The compiler interprets two consecutive double-quote characters within a string constant as a single double-character contained within the string. The interpreter does not.

String Length

The compiler supports a maximum logical string length of 32K; the interpreter logical maximum is 255.

Common Data

BASIC modules that comprise a BASIC program can share variables using the "common data" feature. Common data means that references to a variable from two or more modules refers to the same storage. This option is selected when a module is compiled.

For example, suppose modules A.bas and B.bas are compiled with the "common data" option and both contain the statement "X = 0". The variable X is shared; that is, references to X in A.bas and B.bas refer to the same storage.

Public Code

Selecting the "public code" option when compiling modules allows a BASIC module to transfer control to another BASIC module using GOSUB or GOTO, or by calling a user-defined function. "Public code" means that all line numbers and user-defined functions in a module are public. Program control can be transferred to any line number or user-defined function in the module.

For example, suppose modules Main.bas and Work.bas comprise a BASIC program. Work.bas contains line number 100 and user-defined function FNA. Work.bas is compiled with the "public code" option.

The following statements in Main.bas transfer control from Main.bas to Work.bas.

```
GOSUB 100

GOTO 100
```

Line 100 in Work.bas is as follows:

```
100 X = FNA(1,2)
```

Two modules in a BASIC program compiled with the "public code" option cannot contain the same line number.

Main Modules

A BASIC program must contain exactly one main module. The main module is the first executed when the program is run.

By default, the compiler designates a module as main at compile time. The "suppress main" compiler option prevents a BASIC module from being designated as main.

Non-BASIC Procedures

As in the interpreter, compiler BASIC can call non-BASIC procedures. The "basgen file" compiler option supports the compilation of a module that references non-BASIC procedures. When the "basgen file" option is used, the compiler reads the "basgen file" (an object module) and records the names and interfaces of the non-BASIC procedures. Hence, the compiler can distinguish between non-BASIC procedures and arrays.

Data Statements

If a BASIC program consists of more than one module, the data statements of each module are concatenated. The order of concatentation is the same as the order of object modules as seen by the linker when the BASIC program is linked.

The RESTORE statement restarts the "data statement pointer" to the beginning of the concatenation data statements.

Using Virtual Code Management (Overlays)

When linking a BASIC program consisting of more than one BASIC module, some of the modules can be overlayed. This means that the code portion of the module resides on disk until the module is entered.

A module is placed in an overlay by placing the /O switch at the end of its object module name in the object files list of the linker command form.

There are two restrictions in using overlays.

1. The main module cannot be in an overlay.

2. Modules compiled with the "public code" option cannot be in overlays.

The CHAIN statement transfers control to a BASIC module in an overlay. Note that GOTO, GOSUB, or a user-defined function call cannot be used to transfer control to an overlay because line numbers and user-defined function names cannot be made public.

The BASIC program is not required to perform any initialization to support the use of overlays. Overlay initialization occurs automatically when the main module is entered.

When linking a BASIC program that uses overlays, a special object file must be included in the object file list of the linker command form. This object file is provided as part of the compiled BASIC runtime system.

Using Compiled BASIC

A BASIC program is compiled, linked, and executed in the following manner:

1. Each module that comprises the BASIC program is compiled using the BASIC COMPILE command form. Exactly one module must be compiled as main, and all other modules must be compiled with the "suppress main" option.

   Any module may be compiled with the "common data" compiler option.

   Any module that is not going to be placed in an overlay may be compiled with the "public code" compiler option.

   Any module that references non-BASIC procedures must be compiled in a single invocation of the compiler.

   Any set of modules that uses identical compiler options may be compiled in a single invocation of the compiler.

   The compiler generates an object and list file for each BASIC module. Object and list filename specifications are optional.

2. Once the BASIC modules are compiled, the resulting files are linked together using the Linker to produce an executable run file.

   The Linker searches [sys]<sys>BasCompiler.lib automatically to resolve references to the runtime system for compiled BASIC.

If overlays are used, a special object file must also be included in the object modules list.

3. The BASIC program is executed by using the Executive Run command form or by using a custom command created with the New command.

HINT 2: SINGLE- AND DOUBLE-PRECISION NUMBERS IN BASIC COMPILER

Some real numbers in BASIC cannot be represented as floating-point numbers; when they are converted to decimal the "wrong" answer is returned. This problem is caused by rounding errors in the conversion between a numeric constant and its floating-point representation and the subsequent conversion from floating-point back to a numeric constant. To prevent the incorrect printout of these values, use the PRINT USING or LPRINT USING commands. These commands round the real numbers to the desired number of decimal places, thus giving a correct printout of the values.

EXAMPLE 1:

```
10  A# = .0865
20  B# = .0962
30  PRINT TAB(10); A#; TAB(40);
40  PRINT USING "#.####"; A#
50  PRINT
60  PRINT TAB(10); B#;TAB(40);
70  PRINT USING "#.####"; B#
80  END
```

When the above program is run with the BASIC Interpreter, the values below are printed. Note that similar results occur with the BASIC Compiler. The first column printed uses the PRINT command, and the second column uses the PRINT USING command:

```
8.649998903274536D-02          0.0865

9.619998931884766D-02          0.0962
```

Note that the "wrong" value is also displayed when a LIST of the program is done with the BASIC editor as shown below.

```
10  A#=8.649999E-02
20  B#=9.619999E-02
30  PRINT TAB(10); A#; TAB(40);
40  PRINT USING "#.####"; A#
50  PRINT
60  PRINT TAB(10); B#; TAB(40);
70  PRINT USING "#.####"; B#
80  END
```

EXAMPLE 2:

```
10  ERC% = DEFLPRINT("[SPLB]")
20  FOR I = 1 TO 5 STEP .1
30     LPRINT TAB(10); I; TAB(25)
40     LPRINT USING "#.#"; I
50  NEXT I
60  ERC% = DEFLPRINT("[NUL]")
70  END
```

The following page is the printout of the above program using the BASIC Interpreter. Again, similar results occur with the BASIC Compiler. The values printed range from 1 to 5 and are incremented by a value of one-tenth (0.1). Note, also, that not all the values are printed incorrectly. The first column uses the LPRINT command, and the second column uses the LPRINT USING command.

LPRINT	LPRINT USING
1	1.0
1.1	1.1
1.2	1.2
1.3	1.3
1.4	1.4
1.5	1.5
1.6	1.6
1.7	1.7
1.8	1.8
1.9	1.9
2	2.0
2.1	2.1
2.2	2.2
2.3	2.3
2.4	2.4
2.5	2.5
2.6	2.6
2.7	2.7
2.8	2.8
2.9	2.9
2.999999	3.0
3.099999	3.1
3.199999	3.2
3.299999	3.3
3.399999	3.4
3.499999	3.5
3.599999	3.6
3.699999	3.7
3.799999	3.8
3.899999	3.9
3.999998	4.0
4.099999	4.1
4.199999	4.2
4.299998	4.3
4.399998	4.4
4.499998	4.5
4.599998	4.6
4.699998	4.7
4.799998	4.8
4.899998	4.9
4.999998	5.0

HINT 3: STRING SPACE USE IN THE BASIC COMPILER

This hint explains how the BASIC Compiler allocates and deallocates string space. This information is provided as a guideline to optimize a program that uses strings.

1. Literal strings included in a BASIC source do not use any string space. These strings are stored in the "string constant" section of the code file. In all of the following examples, the REMarks indicate the amount of string space available after the execution of each statement.

```
10  PRINT FRE(0)                    :REM 65510
20  PRINT "STRING VARIABLE"         :REM 65510
30  PRINT FRE(0)                    :REM 65510
```

2. Variable strings are allocated space in chunks of 8 bytes. Two bytes in the first string area are reserved for system use. Therefore, a string of 1 to 6 characters requires 8 bytes; 7 to 14 characters, 16 bytes; 15 to 22 characters, 24 bytes; etc. If an assigned string is reassigned with sufficiently smaller length to free one or more 8-byte chunks, the freed chunks are returned to available user memory.

```
10  PRINT FRE(0)                    :REM 65510
20  A$ = "A"                        :REM 65502
30  B$ = "B"                        :REM 65494
40  A$ = "1234"                     :REM 65494
50  A$ = "1234567"                  :REM 65486
60  A$ = "12345678901234"           :REM 65486
70  B$ = ""                         :REM 65494
80  A$ = ""                         :REM 65510
```

In both statements 20 and 30, 8 bytes of string space are used. Statement 40 does not use any more string space because A$ is still less than 6 characters. In statement 50, an additional 8 bytes of string space are used because A$ is now 7 characters long, which requires 16 bytes of string space. (Note that 8 bytes were already allocated in statement 20.) Again, statement 60 does not use any more string space because A$ did not exceed 14 characters. Statement 70 recovers 8 bytes of string space, and statement 80 recovers 16 more bytes of string space because both A$ and B$ were reassigned to null strings. No difference occurs in the use of string space if the variables are subscripted.

3. String functions such as STRING$, CHR$, SPACES$, etc., generate temporary variables. If a string function occurs in an assignment alone (no concatenation), no temporary variable is created. The amount of string space is calculated as any other string variable.

```
10  PRINT FRE(0)                          :REM 65510
20  A$ = STRING$(15,"A")                  :REM 65486
```

In this example, a 15-character string is created and requires 24 bytes of string space.

4. If string functions are used in concatenated expressions, temporary variables are created and retained. Temporary variables are assigned space like any other string variable. In the following example, three temporary string variables are allocated, one for each HEX$ function and one for the resultant concatenated string. After the statement is executed, the three temporary string variables remain. However, if this statement had been assigned to a variable, then only two of the temporary strings would remain because the resultant concatenated string is assigned to the variable. Note that arithmetic variables do not require any string space.

```
10  PRINT FRE (0)                         :REM 65510
20  A = 123                               :REM 65510
30  B = 456                               :REM 65510
40  PRINT HEX$(A) + HEX$(B)               :REM 65486
```

In this example, 24 bytes of string space are required. Eight bytes are required for HEX$(A), 8 bytes for HEX$(B), and 8 bytes for the temporary string variable created by concatenating HEX$(A) and HEX(B).

5. The number of temporary string variables assigned can be determined by the following formula:

   1 + the maximum number of string functions in a concatenated expression

Once allocated, subsequent concatenation expressions reuse the same temporary string variables. The temporary space can be recovered if necessary with a concatenation of zero-length strings, as shown in the following example.

```
10  PRINT FRE(0)                          :REM 65510
20  A$ = STRING$(200,"A") +
         STRING$(200,"B")                 :REM 64686
30  A$ = STRING$ (0,A$) +
         STRING$ (0,A$)                   :REM 65510
```

In statement 20, three temporary strings are created. One temporary string is created for each STRING$ function and one for the resultant string, A$. Therefore, only two temporary strings remain because the third is assigned to A$. The storage requirement would be the same if the concatenated string was printed instead of assigned, as shown in the example below. In this case, however, all three temporary strings remain.

```
10  PRINT FRE(0)                          :REM 65510
20  PRINT STRING$(200,"A") +
          STRING$(200,"B")                 :REM 64686
30  PRINT STRING$(0,"A") +
          STRING$(0,"A")                   :REM 65510
```

A 200-byte string requires (200 + 2)/8 eight-byte chunks, which equals 26 (25.25) chunks or 208 bytes. A 400 byte string requires (400 + 2)/8 chunks, which equals 51 (50.25) chunks or 408 bytes. Total string space allocation for statement 20 is 208 + 208 + 408 =824 bytes.

Statement 30 works the same way except it reuses the temporary variables from statement 20. In this case, the three temporary strings are assigned to length 0. After execution of this statement, all temporary and user string variables have been set to length 0, so all string storage space is again available.

APPENDIX H
FUNCTIONAL INDEX

VARIABLE/FUNCTION DEFINITION

DEF
DEF FN
DEFINT
DEFDBL
DEFSNG
DIM
LET .. =
=
OPTION BASE

CONSTANT DEFINITION

%, !, #, $, %O, %H, E, D

TERMINAL I/O

INPUT
INPUT$
LINE INPUT
OPTION COMMA
PRINT
PRINT USING
POS()
*SPC()
*TAB()
*USING()
*WIDTH
WRITE

PROGRAM BRANCHING

CHAIN
*IF .. GOTO
GOTO
GOSUB .. RETURN
*ON ERROR GOTO .. RESUME
ON .. GOTO
ON .. GOSUB

INTERNAL DATA MANAGEMENT

DATA
READ
RESTORE

* This command is listed in more than one category.

DIRECT MEMORY ACCESS

FRE()
PEEK()
POKE

POINTER MANIPULATION

GETRA()
GETSA()
MAKEPOINTER()
PTR()
PWA()

SEQUENTIAL FILE HANDLING

EOF()
*CLOSE
INPUT#
*INPUT$
LINE INPUT#
*OPEN
PRINT#
PRINT# USING
*USING$()
WRITE#

RANDOM ACCESS FILE HANDLING

*CLOSE
*CVD()
*CVI()
*CVS()
FIELD
GET
*KILL
LOC()
LSET
*MKI$()
*MKD$()
*MKS$()
*NAME
*OPEN
RSET

* This command is listed in more than one category.

ERROR HANDLING

ERL
ERR
ERROR
*ON ERROR GOTO .. RESUME
SYSERC()

DATE/TIME FORMATTING

DATETIME$

PRINTER OUTPUT

DEFLPRINT()
LPOS()
LPRINT
LPRINT USING
*SPC()
*TAB()
*USING$()
*WIDTH

PARAMETER ACCESS

RGPARAM$()
VERSION$

ARITHMETIC OPERATORS

+ ADDITION
- SUBTRACTION
- NEGATION
* MULTIPLICATION
/ FLOATING POINT DIVISION
\ INTEGER DIVISION
^ EXPONENTIATION
= ASSIGNMENT
MOD MODULUS

RELATIONAL OPERATORS

= EQUALITY
<> INEQUALITY

* This command is listed in more than one category.

< LESS THAN
<= LESS THAN OR EQUAL TO
>= GREATER THAN OR EQUAL TO

LOGICAL OR BOOLEAN OPERATORS

NOT
AND
OR
XOR
IMP
EQV

MATHEMATICAL FUNCTIONS

ABS()
ATN()
COS()
EXP()
FIX()
INT()
LOG()
RND()
SGN()
SIN()
SQR()
TAN()
RANDOMIZE
Derived Functions See Appendix D

TYPE CONVERSION (Types converted)

ASC()	String to ASCII
=	Similar Types
CINT()	Numeric to Numeric
CSNG()	Numeric to Numeric
CDBL()	Numeric to Numeric
CHR$()	ASCII to String
*CVI()	Numeric to Numeric
*CVD()	Numeric to Numeric
*CVS()	Numeric to Numeric
HEX$()	Numeric to String
INSTR()	String to Index
LEFT$()	String to String
LEN()	String to current Length
MID$()	String to String
OCT$()	Numeric to String
RIGHT$()	String to String

* This command is listed in more than one category.

SPACE$() String to String
STRING$() String to String
STR$() Numeric to String
SWAP Similar types
VAL() Numeric to String

PROGRAM CONTROL

CALL
CHAIN
END
FOR .. NEXT
*IF .. GOTO
IF .. THEN .. {ELSE}
STOP
SYSTEM
WHILE .. WEND

COMMENTS

'
REM

DEFAULT PROCEDURES IN BASGEN

GETSTAMFILEHEADER
GETMODULEID
GETPASCB
GETUCB
GETPARTITIONSTATUS
GETPSTRUCTURE
GETUSERNUMBER
QUERYREQUESTINFO
QUERYVIDHDW
STRINGSEQUAL
UPDATEVAM

PORT I/O

INP()
OUT
WAIT

* This command is listed in more than one category.

APPENDIX I

INSTALLATION PROCEDURES

The following are installation procedures for invoking the Install.sub and XEInstall.sub for the BASIC Compiler.

HARD DISK INSTALLATION

Boot the system from the master or cluster where the installation of BASIC Compiler is desired. The software is installed in the system files of the system directory.

- Insert the BASIC Compiler diskette in floppy drive [f0].

  Do not press the **RESET** button.

- Enter the **SOFTWARE INSTALLATION** command on the command line and press **GO**.

- Follow the instructions displayed on the screen.

- When installation is complete, remove the distribution diskette and store it in a safe place.

XE 520 INSTALLATION

Boot the cluster workstation being used for the installation from the XE520.

- Power off all other cluster workstations.

- Log onto user ADMIN.

- Insert the BASIC Compiler diskette in floppy drive [f0].

  Do not press the **RESET** button.

- Enter the **SUBMIT** command on the command line and press **RETURN.** The following parameter appears on the screen:

```
SUBMIT
   File List        [f0]<sys>XEInstall.sub
```

  Press **GO** to invoke the **SUBMIT** command.

- Follow the instructions displayed on the screen.

- When installation is complete, remove the distribution diskette and store it in a safe place.

APPENDIX J
PRODUCT USAGE ON B 26 DUAL FLOPPY STANDALONE SYSTEMS

INTRODUCTION

BASIC 4.0 can be used on B 26 Dual Floppy Standalone systems. Before creating, compiling, linking, and running BASIC programs on the B 26 Dual Floppy, duplicate the BASIC disk, and the Language Development disks by using the following procedures:

1. Place the system disk of the 4.0 B 26 Dual Floppy into disk drive [f0].

2. Execute the **FLOPPY COPY** command as follows:

```
FLOPPY COPY
   [Number of copies]
   [Overwrite OK?]
   [Dual floppy?]          Yes
   [Suppress verify?]
   [Device names(s)]
   [Device password(s)]
```

A 4.0 Linker disk for BASIC should also be set up to save time transferring disks in and out of the floppy drives, and to prevent stating CTOS.Lib and BasCompiler.Lib in the **LINK** command. To set up a 4.0 Linker, use the following procedure:

1. Place the system disk of the 4.0 B 26 Dual Floppy in disk drive [f0].

2. Place a blank disk in disk drive [f1].

3. Execute the **IVOLUME** command as follows:

```
IVOLUME
   Device name                          f1
   [Device password]
   Volume name                          BasLinker
   [Volume password]
   [System Image (default = 384)]       0
   [Log file (default = 2)]             0
   [Crash file (default = 0)]
   [Max. directories]                   2
   [Max. files on volume]               15
   [Primary file headers only?]         Yes
   [Max. files in Sys Directory]        12
   [Sys Directory password]
   [Write protect Sys Directory?]
   [Suppress format of medium?]
   [Surface tests]                      0
   [Debug?]
   [Log file]
   [Extend floppy tracks?]
   [Single-sided mini-floppy]
   [Bad spots (See Documentation)]
```

4. Execute the **COPY** command as follows:

```
COPY
   File from            [f0]<sys>Exec.Run
   File to              [f1]<sys>Exec.Run
   [Overwrite ok?]
   [Confirm each?]
```

5. Execute the **CREATE DIRECTORY** command as follows. Use the directory name of the directory that contains Linker.run on the Language Development disk (e.g., <Burroughs>).

```
CREATE DIRECTORY
   New directory name                   [f1]<Burroughs>
   [Default protection level
      (e.g., 15)]
   [Maximum number of files
      (default 45)]                     3
   [Password for new directory]
   [Volume password]
```

6. To execute the **COPY** command, type **Copy** and press **RETURN.** Place the Language Development 4.0 (B20 LD4-1) disk which contains Linker.run into disk drive [f0].

7. Execute the **COPY** command as follows:

```
COPY
   File from          [f0]<Burroughs>Linker.Run
   File to            [f1]<Burroughs>Linker.Run
   [Overwrite ok?]
   [Confirm each?]
```

8. To execute the **COPY** command, place the system disk of the 4.0 B 26 Dual Floppy disks in disk drive [f0]. Type **Copy** and press **RETURN.** Place the Language Development 4.0 (B20 LB4-1) disk which contains Linker.Run into disk drive [f0].

9. Execute the **COPY** command as follows:

```
COPY
   File from          [f0]<sys>Sys.Cmds
   File to            [f1]<sys>Sys.Cmds
   [Overwrite ok?]
   [Confirm each?]
```

10. BasCompiler.Lib must be transferred to the 4.0 Linker disk for BASIC. The name of the directory where BasCompiler.Lib resides should be used (e.g., <sys>).

    To execute the **COPY** command, place the system disk of the 4.0 B 26 Dual Floppy disks in disk drive [f0]. Type **Copy** and press **RETURN.** Place the BASIC 4.0 (B20 BC4) disk into disk drive [f0].

11. Execute the **COPY** command as follows:

```
COPY
   File from          [f0]<Burroughs>BasCompiler.Lib
   File to            [f1]<sys>BasCompiler.Lib
   [Overwrite ok?]
   [Confirm each?]
```

12. CTOS.Lib must be transferred to the 4.0 Linker for BASIC. The name of the directory where CTOS.Lib resides should be used (e.g. <Burroughs>).

    To execute the **COPY** command, place the system disk of the 4.0 B 26 Dual Floppy disks in disk drive [f0]. Type **Copy** and press **RETURN.** Place the Language Development 4.0 (B20 LD4-2) disk into disk drive [f0].

13. Execute the **COPY** command as follows:

```
COPY
   File from          [f0]<Burroughs>CTOS.Lib
   File to            [f1]<sys>CTOS.Lib
   [Overwrite ok?]
   [Confirm each?]
```

CREATING A PROGRAM

Use the following procedure to create a program:

1. Place the Editor disk of the 4.0 B 26 Dual Floppy in disk drive [f0].

2. Place a relatively empty source disk in disk drive [f1].

3. Execute the **EDIT** command as follows:

```
EDIT
   File           [f1]<YourDir>YourFile.Bas
   [Your name]
```

where

YourDir
: is the name of the directory on the source disk where the BASIC source program, YourFile.Bas, is to reside. The Editor is used and exited in a normal manner.

COMPILING A PROGRAM

Use the following procedure to compile the program:

1. Place the 4.0 BASIC disk in disk drive [f0].

2. Place the source disk with YourFile.Bas in disk drive [f1].

3. Execute the **BASIC** command as follows:

```
BASIC COMPILER
   Source file              [f1]<YourDir>YourFile.Bas
   [Object file]            _____________________
   [List file]              _____________________
   .
   .
   .
   [No code generation?]    _____________________
```

This compilation creates a list file called YourFile.Lst and an object file called YourFile.Obj on the source disk. To change the destination or name of these files, explicitly state the path and name of them. When the compilation is complete, the message

Please mount a system volume in [sys] and press GO to continue

is displayed. Put the system disk of the 4.0 B 26 Dual Floppy disks into drive [f0] and press **GO**.

LINKING A PROGRAM

Use the following procedure to link the program:

1. Place the 4.0 Linker disk for BASIC in disk drive [f0].

2. Place the source disk with YourFile.Obj in disk drive [f1].

3. Execute the **LINK** command as follows:

```
LINK
   Object modules     [f1]<YourDir>YourFile.Obj
   Run file           [f1]<YourDir>YourFile.Run
   [List file]        _________________________
   .
   .
   .
   [Debug?]           _________________________
```

This link creates a run file called YourFile.Run, a map file called YourFile.Map and a symbol file called YourFile.Sym. To change the destination or name of these files, explicitly state the path and name of them.

RUNNING A PROGRAM

Use the following procedure to run a program:

1. Place the system disk of the 4.0 B 26 Dual Floppy disks in disk drive [f0].

2. Place the source disk with YourFile.Run in disk drive [f1].

3. Execute the **RUN** command as follows:

```
RUN FILE
   Run file          [f1]<YourDir>YourFile.Run
   [Parameters]      ________________________
```

APPENDIX K
GLOSSARY

Common Data. Common data means that references to a variable from two or more modules point to the same area in memory.

Escape Sequence. An escape sequence is a sequence of characters that invokes special functions.

Library. A library is a collection of related object module files.

Main Module. The main module is the first module executed when the program is run. All compiled BASIC programs must contain exactly one main module.

Module. A module is a BASIC source file.

Nonexecutable Statement. In compiled BASIC, a nonexecutable statement (for example, DIM, DEF, FN and DEFINT/SNG/DBL/STR) takes effect when it is compiled. It affects all statements that are compiled after it.

Object Module. An object module is the result of a single compilation or assembly. A single object module is contained in an object module file (.Obj), while many object modules can be contained in a library file (.Lib).

Overlay. An overlay is a code segment, made up of the code from one or more object modules. An overlay is loaded into memory from disk only when it is needed and is not permanently memory-resident.

Printer Spooler. The printer spooler is a dynamically installed system service that transfers text from disk files to the printer interfaces of the workstation in which it is installed.

Program. A program consists of one or more modules that are linked together to form an executable run file.

Public Code. Public code means that all line numbers and user-defined functions in a module are accessible to all other modules in the program.

Public Symbol. A public symbol appears in the symbol file (created by the Linker) for all line numbers, user-defined functions, and program variables when the program is compiled with the common data and public code options.

Run File. A run file is a memory image of a task (in ready-to-run form) linked into the standard format required by the Operating System loader.

Run Time. The time during which a program is executing.

Run-Time System. The BASIC run-time system is the software that supports the features of the BASIC language. For example, the BASIC file system, the mathematical routines, and the string management routines are part of the run-time system. Physically, the run-time system is linked with a BASIC program and is part of the run file. The object modules that comprise the run-time system are contained in BasCompiler.Lib and CTOS.Lib.

Source File. The source code that constitutes the input to an assembler, interpreter, or a compiler for translation.

INDEX

Documentation Evaluation Form

Title: B 20 Systems BASIC Compiler Reference Manual (4.0)

Form No: 1180148

Date: April, 1985

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 Comment/Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error ☐ Other

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Burroughs Corporation
Corporate Product
Information East
209 W. Lancaster Ave.
Paoli, PA 19301 U.S.A.