Burroughs

Reference Manual

B 20 Systems

BASIC Interpreter

(Relative to Release Level 4.0)

Distribution Code SA

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

1180155

Burroughs

Reference Manual

B 20 Systems BASIC Interpreter

(Relative to Release Level 4.0)

Distribution Code SA

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

1180155

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 xiv	Original
1-1 thru 1-23	Original
1-24	Blank
2-1 thru 2-82	Original
3-1 thru 3-48	Original
4-1 thru 4-30	Original
A-1 thru A-12	Original
B-1, B-2	Original
C-1 thru C-9	Original
C-10	Blank
D-1	Original
D-2	Blank
E-1 thru E-11	Original
E-12	Blank
F-1 thru F-8	Original
G-1, G-2	Original
H-1, H-2	Original
1 thru 9	Original
10	Blank

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

OVERVIEW

The BASIC Language Reference Manual is divided into four main sections. Section 1 deals with a variety of topics, including invocation of the BASIC Interpreter from the Executive and information representation when using BASIC. Section 2 discusses BASIC commands, the third section deals with the system-supported functions of BASIC, and the fourth section explains the error messages used by BASIC. There are also seven appendices, which deal with special topics. Appendix A outlines the procedures used to produce special effects on the screen using B20 BASIC. This appendix shows how to produce effects such as reverse video, underlining, and flashing messages. Appendix B gives the necessary procedures for printing in BASIC when your B20 system has the print spooler option. This appendix outlines the necessary steps required to use the LLIST, LPRINT, and LPRINT USING commands when operating a B20 computer that uses the Spooler utility to print. Appendix C gives the necessary procedures and examples for creating sequential and random access files. Appendix D outlines mathematical functions not intrinsic to BASIC but that can be calculated using BASIC functions. Appendix E explains the steps for calling non-BASIC procedures and how to configure a new Basic.run file. Appendix F provides programming hints for the programmer. Appendix G deals specifically with dual floppy use.

In this manual, underlined words represent the information that you supply when using the BASIC command or function. For example, the manual treats the PRINT command with the form:

10 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.

This manual presents BASIC system commands without line numbers. For example, the AUTO command is presented as:

AUTO beginning line number, line number interval.

All other BASIC commands and functions used in this manual are presented with BASIC statement numbers.

ORGANIZATION OF THE MANUAL

This manual consists of four main sections and seven appendices. Their contents are described below.

SECTION 1: GENERAL INFORMATION

The first section contains general information about BASIC. It also contains instructions on how to load the BASIC Interpreter distribution disk B20BI4 (Level 4.0) on your B20 system and how to invoke BASIC from the executive.

SECTION 2: BASIC COMMANDS

The second section lists all BASIC commands alphabetically for easy reference. Each command is presented with the following information:

- a statement that shows the general form of the command
- a description of the command that explains how it works
- an example of the command used in a BASIC program
- an explanation of the example describing the action caused by the command

SECTION 3: BASIC FUNCTIONS

The third section defines the BASIC functions available on your B20 system. Again, each function is listed alphabetically for easy reference.

The functions are presented with:

- an example of the general form in which the function is used
- a description of how each function works
- an example of the function in a program
- an explanation of the example describing the action caused by the function

SECTION 4: BASIC ERROR MESSAGES

The fourth section lists the error messages you may encounter when using BASIC. These messages are listed sequentially by error number. A cross-reference is also provided listing error messages alphabetically so that you can locate the message quickly. The error message section describes each error message and even suggests some possible actions to solve the problem.

APPENDIX A: CONTROL OF THE VIDEO DISPLAY

Appendix A describes how to control the display of information on the B20 screen. B20 BASIC can control the following:

- 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
- keyboard LED indicators
- directing video display output

In addition, B20 BASIC allows you to:

- fill in various size rectangles with a specified character
- erase information in lines or frames
- display special characters literally

This appendix gives the procedures for creating these effects. Each procedure is described and presented with an example of its use.

APPENDIX B: PRINTING IN BASIC WITH THE PRINT SPOOLER

Appendix B presents the steps necessary to use the BASIC commands LPRINT, LPRINT USING, and LLIST on B20 systems that use the Spooler utility to print. Once again, the necessary steps and examples are presented.

APPENDIX C: DISK INPUT/OUTPUT

Appendix C details the steps to create sequential and random access files. Examples are given.

APPENDIX D: MATHEMATICAL FUNCTIONS

Appendix D gives the BASIC equivalent of mathematical functions not intrinsic to BASIC.

APPENDIX E: CALLING NON-BASIC PROCEDURES

Appendix E explains how to call non-BASIC procedures, gives some programming recommendations, and explains how to configure a new Basic.run file.

APPENDIX F: PROGRAMMING HINTS

APPENDIX G: INSTALLATION PROCEDURES

APPENDIX H: DUAL FLOPPY STANDALONE

HOW TO USE THIS MANUAL

This 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. The manual is complete in its treatment of BASIC commands, functions, and error messages so that a user can easily create and run a BASIC program. If the user desires to learn how to write in BASIC, the B20 BASIC Training Course is available.

SECTION 1
GENERAL INFORMATION

The BASIC Interpreter distribution diskette level 4.0 contains the following files:

Disk B20BI4

<SYS>FileHeaders.Sys
<SYS>Mfd.Sys
<SYS>Log.Sys
<SYS>SysImage.Sys
<SYS>BadBlk.Sys
** <SYS>DiagTest.Sys
** <SYS>BootExt.Sys
<SYS>CrashDump.Sys
<SYS>Install.Sub
<SYS>Sys.cmds
<SYS>FdSys.version
<SYS>BasicRes.run
<B20BI4>Basic.run
<B20BLD>StartBasicLink.sub
<B20BLD>Basic.lib
<B20BLD>MaxMemArrayRes
<B20BLD>ObjBasIntRes.fls
<B20BLD>Basgen.asm
<B20BLD>MaxMemArraySwp
<B20BLD>ObjBasIntSwp.fls

** These system files do not exist on the 5-1/4 inch diskette.

INVOKING BASIC

To invoke the BASIC Interpreter from the Executive, type "BASIC" (or as many letters as are required to make the command unique; see the Executive Manual) in the command field of the command form. The form illustrated below then appears.

```
BASIC
 [Initial Program]
 [Maximum number of open files (0-15)] _____
 [Maximum random record length]        _____
```

Fill in the fields. All the fields are optional. You can default optional fields by leaving them blank.

Press GO when the form is correctly filled in.

Field Descriptions

[Initial program]

Fill in the "[Initial program]" field with the name of an initial BASIC program for BASIC to load and run.

The default is none; that is, entering no response causes no initial program to be executed, and BASIC enters direct mode (see "Modes of Operation" below).

[Maximum number of open files (0-15)]

Fill in the "[Maximum number of open files (0-15)]" field with the number of files than can be open at the same time. The maximum is 15.

The default is 6; that is, entering no response limits to six the number of files that can be open at the same time.

The parameter given can affect the amount of free memory available to a BASIC program. See the discussion under the "[Maximum random file record length]" field.

[Maximum random record length]
Fill in the "[Maximum random record length]" field to limit the size of random file records.

The default is 128; that is, entering no response limits to 128 bytes the size of random file records.

The parameters given to "[Maximum number of open files (0-15)]" and "[Maximum random file record length]" can affect the amount of free memory available to a BASIC program. Before allocating free memory, BASIC allocates a buffer for each file that can be opened. Each buffer is as large as the maximum random file record length. If less than 61K bytes of free memory is available, you can recover free memory by decreasing either the maximum number of open files or the maximum record length.

MEMORY REQUIREMENTS: SWAPPING AND RESIDENT VERSIONS OF BASIC.RUN

The distribution diskette contains two versions of the BASIC Interpreter: swapping and resident. The installation procedure described earlier installs the swapping version.

The swapping BASIC Interpreter, Basic.run, uses the Virtual Code Management facility. Selected runtime procedures remain on the disk as overlays until they are required.

The swapping version reduces the memory requirements of the runtime system with the cost of a small performance degradation. In applications where it is preferable to trade memory for performance, BasicRes.run should be used.

The following amounts of memory are required to execute BASIC with a maximum allocation of free memory to the BASIC program.

Resident Version	128,896 bytes
Swapping Version	122,680 bytes

If these amounts of memory are not available, the amount of free memory available to the BASIC program is reduced.

Additional memory is required by BASIC for use as buffers when files are opened.

Each open sequential file requires an additional 512 bytes.

Each open random file requires a DAM buffer. Its size is generally a multiple of 512 greater than or equal to the record size plus 519. There are two exceptions to this formula. First, if 512 is a multiple of the record size plus 8, then the buffer size is 512. Second, if the record size plus 8 is a multiple of 512, then the buffer size is the record size plus 8.

MEMORY ORGANIZATION

BASIC is distributed among several code segments. An additional segment, the BASIC data segment, contains BASIC's stack, internal variables, and all free memory available to the BASIC program.

BASIC allocates up to 64K bytes to the BASIC data segment. If the largest data segment is allocated, 61K bytes of free memory are available. (If you use the defaults, see the previous section, "Invoking BASIC".)

MODES OF OPERATION

When BASIC is initialized, it types the prompt "Ok". "Ok" means BASIC is at command level and is ready to accept commands. At this point, BASIC can be used in either of two modes: direct or indirect.

In direct mode, BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations can be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC for quick computations that do not require a complete program.

Indirect mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

If you are using indirect mode and wish to save your program during the work session, before invoking BASIC check the amount of storage space you have available on the media that you will use to save your work. If there is a chance that you do not have enough available storage, either create additional storage space (possibly by deleting unnecessary files) or use another media. If your program is larger than the available storage space, by the time the DISK FULL error message appears your program may have been corrupted.

LINE FORMAT

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

```
nnnnn BASIC statement[:BASIC statement...]
<RETURN>
```

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

A BASIC program line always begins with a line number, ends with a <RETURN>, and can contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line. If you press <RETURN> at the end of a line, the logical line is terminated. If you press <MARK>, the physical line is terminated but you can continue typing the same logical line on the next physical line. Pressing <MARK> echoes a triangle and moves the cursor to the beginning of the next line.

Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range zero to 65529. A period can be used in the EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the BACKSPACE key. Once a character is deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, press the DELETE key.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC automatically replaces the old line with the new line.

On a Burroughs Information Processing System, there are three more sophisticated tools available for the preparation and editing of BASIC programs: the EDIT command (described in the "EDIT" subsection below), the Editor, and the word processor.

The Edit command lets you view and edit your program one line at a time, without leaving BASIC. If, however, you wish to make extensive changes in your program, you will probably find it easier to SAVE the program (using the ASCII option) and exit from BASIC to the Executive. Then invoke the Editor (see the Editor Manual for details), edit as desired, and finally reinvoke BASIC and reload your program.

You can create BASIC source programs and data files using WRITEone. These programs and data files can be formatted. However, BASIC processes formatted files as if they were not formatted.

To delete the entire program that is currently residing in memory, enter the NEW command. (See the NEW subsection in the "Commands and Statements" section below.) NEW is usually used to clear memory before entering a new program.

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 BASIC:

Character or Key	Name
	Space
=	Equal sign or assignment symbol
+	Plus sign or contatenation symbol
-	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 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
;	Semicolon
:	Colon or multiple statement separator
&	Ampersand or continuation symbol
?	Question mark

Character or Key	Name
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
BACKSPACE	Deletes last character typed
GO	Escapes Edit Mode subcommands
TAB	Moves print position to next tab stop. Tab stops are every eight columns.
MARK	Moves to next physical line
RETURN	Terminates input of a logical line

Control Characters

BASIC has the following control functions:

ACTION-CANCEL	If a program is not waiting for keyboard input, interrupts program execution and returns to BASIC command level.
ACTION-O	Halts program output while execution continues. A second ACTION-O restarts output.
ACTION-S	Suspends program execution.
ACTION-Q	Resumes program execution after an ACTION-S.
CANCEL	If a program is waiting for keyboard input, interrupts program execution and returns to BASIC command level.
CODE-A	Enters edit mode on the line being typed.
CODE-R	Retypes the line that is currently being typed.

DELETE	Deletes the line that is currently being typed.
FINISH	Asks if BASIC is to be exited. If FINISH is pressed again, BASIC is exited.

Video Display Special Characters

BASIC interprets the following characters in a special way when they are output on the video display:

Decimal Value	Key	Interpretation
1	up arrow	Move the cursor up one line. If the cursor is in the top line of the frame, reposition it to the bottom.
7	CANCEL	Activate audio alarm for one half-second.
11	down arrow	Move the cursor down one line. If the cursor is in the bottom line of the frame, reposition it to the top line.
12	NEXT PAGE	Blank the frame and position the cursor in its upper left hand corner.
13	BOUND	Ignored.
14	left arrow	Move the cursor left one character position. If the cursor is in the first column of the frame, reposition it to the last column.
18	right arrow	Move the cursor right one character position. If the cursor is in the last column of the frame, reposition it to the first column.

CONSTANTS

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

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

A string constant cannot contain a double quotation mark. Use the CHR$ function (see Section 3) to construct a string containing this character.

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.

1. Integer Constants
   Whole numbers between -32768 and +32767. Integer constants do not have decimal points.

2. Fixed-point Constants
   Positive or negative real numbers, that is, numbers that contain decimal points.

3. Floating-point Constants
   Positive or negative exponential form (similar to scientific notation). A floating-point constant consists 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:

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

   (Double-precision floating-point constants use the letter D instead of E. See below.)

4. Hexadecimal Constants
   Hexadecimal numbers with the prefix &H.
   Examples:

```
&H76
&H32F
```

5. Octal Constants
   Octal numbers with the prefix &O or &.
   Examples:

```
&0347
&1234
```

Single- and Double-Precision Form for Numeric 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 printed 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 (!)

Examples:

```
   46.8
  -1.09E-06
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:

```
345692811
      -1.09432D-06
    3489.0#
 7654321.1234
```

VARIABLES

Variables represent values used in a BASIC program. The value of a variable can be assigned explicitly by the programmer, or it can be assigned as the result of calculations in the program. Before a numeric variable is assigned a value, its value is assumed to be zero. Before a string variable is assigned a value, its value is assumed to be null.

Variable 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 (see below).

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 ($) as the last character. For example,
A$ = "SALES REPORT". 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. The type declaration characters for these variable names are as follows:

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

The default type for a numeric variable name is single-precision.

Examples of variable names are:

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

There is a second method by which variable types can be declared. The BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL can be included in a program to declare the types for certain variable names. These statements are described in detail under DEFINT/SNG/DBL/STR in Section 2.

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 variable 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, use brackets. This permits BASIC to easily distinguish between arrays and value-returning non-BASIC procedures. Brackets improve 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.

TYPE CONVERSION

When necessary, BASIC converts a numeric constant from one type to another. Keep the following rules and examples in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, 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, a "Type mismatch" error message occurs.) Example:

```
10 A% = 23.42
20 PRINT A%
RUN
 23
```

2. During expression evaluation, all operands in an arithmetic or relational operation are converted to the same degree of precision, that is, 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#
RUN
 .8571428571428571
```

The arithmetic was performed in double precision and the result was returned in D# as a double-precision value.

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

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.

3. Logical operators (see the subsection of that name below) 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.

4. When a floating-point value is converted to an integer, the fractional portion is rounded. Example:

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double-precision variable is assigned a single-precision value, 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. Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04  2.039999961853027
```

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, and functional.

Arithmetic Operators

The arithmetic operators, in order of precedence, are:

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.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC 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^Z}	X^(Y^Z)	
X(-Y)	X*(-Y)	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 (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer. For example:

```
      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 2)
25.68 MOD 6.99 = 5 (26/7 = 3 with a remainder 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 zero is encountered, the "Division by zero" error message is displayed, 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 zero being raised to a negative power, then the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, then the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

This error condition can not be trapped by conventional error-trapping routines. This allows for program execution to continue in spite of the error condition. The message is displayed, but variables ERR and ERL will not be set.

Relational Operators

Relational operators 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. (See IF in Section 2.)

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

(The equal sign is also used to assign a value to a variable. See LET in Section 2 below.)

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. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

Logical Operators

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

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as relational operators can 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 (see IF in Section 2). 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 -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as zero or -1, logical operators return zero 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 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, 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 in an expression calls 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 BASIC intrinsic functions are described in Section 3.

BASIC also allows "user-defined" functions, which are written by the programmer. See DEF FN in Section 2.

String Operations

Strings can be concatenated using +. For example:

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

Strings can be compared using the same relational operators that are used with numbers:

```
=
<>
<
>
<=
>=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, 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.

Examples:

```
        "AA" < "AB"
  "FILENAME" = "FILENAME"
        "X&" > "X#"
      "CL " > "CL"
        "kg" > "KG"
     "SMYTH" < "SMYTHE"
          B$ < "9/12/78"   WHERE B$ = "8/12/78"
```

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

SECTION 2
BASIC COMMANDS

This section of the manual lists all BASIC commands alphabetically. Each command is presented with the following information:

- the general form of the command
- an explanation of how the command works
- an example of the command used in a BASIC program
- an explanation of the example describing the command's action

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

10 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.

AUTO

AUTO

Form of Statement

AUTO

AUTO *beginning line number*

AUTO *beginning line number*, *line number increment*

DESCRIPTION

The AUTO command is entered before you begin program input. It automatically numbers the lines of the program as they are entered. There are three forms of the AUTO command.

The first form of AUTO begins with line number 10 and increases by 10 for each new line entered.

The second form begins with the number you put in as the beginning line number and increases by 10 for each new line entered.

The third form of AUTO begins numbering lines with the number you put in as the beginning line number and increases by the value specified as the line number interval. The AUTO command does not appear in your program listing. Terminate AUTO by pressing CANCEL.

EXAMPLE	EXPLANATION
AUTO	This command numbers lines in increments of 10, beginning with 10 for the first line.
AUTO 100	This command numbers lines in increments of 10, beginning with 100 for the first line.
AUTO 100, 20	This command numbers lines in increments of 20, beginning with 100 for the first line.

CALL

Form of Statement

10 CALL procedure name

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

DESCRIPTION

The CALL command calls a nonvalue-returning, non-BASIC procedure. The command may call a procedure using only the procedure name, as in statement 10 above, or it may call a procedure and pass information to that procedure as is done in statement 20.

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

EXAMPLE	EXPLANATION
10 REM USING CALL COMMAND 20 CALL INITIALIZE (PTR(A#))	The program calls the procedure INTIALIZE, passing a value that is a pointer.

CHAIN

Form of Statement

```
10 CHAIN  "filename"

10 CHAIN  "filename",,ALL

10 CHAIN  "filename",
          line number

10 CHAIN  "filename", line
          number, ALL

10 CHAIN MERGE "filename"

10 CHAIN MERGE "filename",,
         DELETE line number
         range

10 CHAIN MERGE "filename",
   line number, DELETE line
   number range
```

DESCRIPTION

The CHAIN command calls a BASIC program and may pass variables to it from the current program. The filename in the statement format above is the name of the called program. Programs load faster if they are stored in binary instead of ASCII.

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

When the ALL option is used, every variable in the current program is passed to the called program. If the ALL option is not used, only those variables listed in a COMMON statement are passed to the called program.

When the ALL option is not used, a COMMON statement must be included in the current program before the second program is called if variables are to be passed to the called program.

When the MERGE option is used, the called program is overlayed on the existing program. The merged program must be an ASCII file.

The DELETE option deletes an overlay that was previously included with the MERGE option. For example, if the following statement is executed:

```
CHAIN MERGE "OVERLAY1", 1000
```

then it may be desirable to delete it when the program is done with it so that a new overlay can be brought in as in the following statement:

```
CHAIN MERGE "OVERLAY2", 1000, DELETE 1000-5000
```

NOTE

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program. Also, OPTION BASE is not preserved.

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 begins at line number 200 of MYPROG.*
180 CHAIN "MYPROG",,ALL	Statement number 180 calls the program MYPROG. Execution begins at the first line of MYPROG and all variables from the current program are passed to MYPROG.*
220 CHAIN MERGE "MYPROG"	Statement number 220 calls the program MYPROG and overlays it on the program currently in memory.
240 CHAIN MERGE "MYPROG", DELETE 10-50	Statement number 240 calls the program MYPROG and overlays is on the program currently in memory. When MYPROG is executed, it is deleted from memory.

* The program MYPROG replaces the program currently in memory.

CLEAR

CLEAR Form of Statement

20 CLEAR

DESCRIPTION

The CLEAR command clears all memory, sets all numeric variables to 0 and sets all string variables to null. It is recommended that you do not use the clear command in subroutines.

EXAMPLE	EXPLANATION
10 A$ = "GOOD" 20 B$ = "DAY" 30 PRINT A$ +B$ 40 CLEAR 50 C$ = "BYE" 60 PRINT A$ +B$ +C$	This program displays GOODDAY in statement 30 and then sets the variables A$ and B$ as having no value Statement 60, which displays the string values of A$ + B$ + C$, will only display BYE. Variables A$ and B$ have no value.

CLOSE

Form of Statement

150 CLOSE

180 CLOSE # first filenumber,
second filenumber,...

DESCRIPTION

The CLOSE command ends 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.

EXAMPLE	EXPLANATION
120 CLOSE	This command closes all open files.
240 CLOSE #1	This command closes the file opened as #1. All other open files remain open.

COMMON

COMMON Form of Statement

10 COMMON first variable,
second variable,...

DESCRIPTION

The COMMON statement indicates the variables to be passed to a program when that program is called by the CHAIN command.

Common statements may appear anywhere in a program, although it is recommended that they appear at the beginning.

The same variable cannot be in more than one COMMON statement.

If an array is to be passed to a called program, the array name followed by a ( ) should be included in the COMMON statement.

EXAMPLE	EXPLANATION
100 COMMON A, B, C, XYZ ( ), G$ 110 CHAIN "MYPROG"	The MYPROG program is called from the current program. The three numeric variables (A, B, and C), an array (XYZ), and a string variable (G$) are passed to program MYPROG.

CONT

Form of Statement

CONT

DESCRIPTION

The CONT command continues the execution of a program when execution is interrupted by pressing ACTION-CANCEL or when a STOP statement is encountered in the program.

The program execution continues from the point at which the interruption was encountered.

The CONT command does not work if the program is edited during the interruption.

EXAMPLE

```
10 A = 2
20 B = 3
30 SUM = A + B
40 STOP
50 Print SUM
```

EXPLANATION

This program will execute up to line 40. At Line 40 there is a break. To continue, the user enters a CONT command. The program execution continues at line 50 and displays the value of the variable SUM.

DATA

Form of Statement

100 DATA constant, constant, ...

DESCRIPTION

The DATA statement is a nonexecutable 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 line number.

A DATA statement may appear anywhere in a program before an END statement.

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	EXPLANATION
10 READ A, B, H$	When run, this program reads the values for the variables in statement 10 from the DATA statement (statement 90).
90 DATA 5, 10, TOTAL	A is assigned 5, B is assigned 10, and H$ is assigned the string constant TOTAL.

DEF FN

Form of Statement

10 DEF FN function name = function definition

or

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

DESCRIPTION

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 and is limited to one line.

The variables that appear in the function definition are used only to define the function. They do not affect program variables that 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, an error message occurs. DEF FN is illegal in direct mode.

EXAMPLE

```
40 DEF FNABC (X,Y) = X*Y
50 COST = FNABC  (RATE,TIME)
60 AREA = FNABC  (H,W)
```

EXPLANATION

Statement 40 defines a function, FNABC, to be the value of X multiplied by Y.

Statement 50 assigns to the variable COST the value of the function using the variables RATE and TIME. COST equals the value of RATE multiplied by TIME.

Statement 60 assigns AREA the value of H multiplied by W.

DEF

Form of Statement

10 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 to be single-precision variables.

EXAMPLE	EXPLANATION
20 DEFDBL A-G	Defines all variables beginning with the letters A through G as double precision.
40 DEFSTR X	Declares that all variables beginning with the letter X as string variables.
50 DEFINT A-C, W-Z	Declares all variables beginning with the letters A, B, C, W, X, Y, or Z as integer variables.

DELETE

Form of Statement

DELETE line number

or

DELETE line number-line number

or

DELETE -line number

DESCRIPTION

The DELETE command deletes program lines and returns to BASIC command level. If the line number does not exist, an error message results.

EXAMPLE	EXPLANATION
DELETE 40	Deletes line 40 of program currently in memory.
DELETE 40 - 80	Deletes lines 40 through 80 of the program currently in memory.
DELETE - 100	Deletes all lines up to and including line 100 of the program currently in memory.

DIM

DIM

Form of Statement

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

DESCRIPTION

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

It sets all elements of the specified numeric arrays to a value of 0 and all elements of string arrays as having no value.

EXAMPLE	EXPLANATION
10 DIM A [5] 20 FOR I = 1 to 5 30 READ A [I] 40 NEXT I	The DIM statement specifies that variable A has a maximum of 5 elements.
50 DATA 5, 6, 8, 10, 100	The remainder of the program reads the values for each of the elements.

EDIT Form of Statement

EDIT line number

DESCRIPTION

The EDIT command enters edit mode at the specified line number. In edit mode it is possible to edit portions of a line without retyping the entire line. Upon entering edit mode, BASIC types the line number of the line to be edited, then it types a space and waits for an edit mode subcommand.

Edit mode subcommands move the cursor or insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most edit mode subcommands can be preceded by an integer, which executes the command that number of times. The default preceding integer is 1.

The subcommands within edit mode allow the user to:

move the cursor
insert text
delete text
find text
replace text
end and restart edit mode

NOTE

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and GO represents the GO key.

The subcommands used in the BASIC edit mode are shown in the following table:

ACTION	SUBCOMMANDS	DESCRIPTION OF SUBCOMMAND
moving the cursor	<space>	Use the space bar to move the cursor to the right. [i]<space> moves the cursor i spaces to the right. Characters are printed as you move.
	BACKSPACE	In edit mode, [i] BACKSPACE moves the cursor i spaces to the left. Characters are printed as you move.

ACTION	SUBCOMMANDS	DESCRIPTION OF SUBCOMMAND
insert text	I	inserts text beginning at the cursor position. I<text> GO inserts <text> at the current cursor position. The inserted characters are displayed on the screen. To terminate insertion, press GO. If RETURN is pressed during an Insert command, the effect is the same as pressing GO and then RETURN. During an Insert command, BACKSPACE can be used to delete characters to the left of the cursor. If you try to insert a character that makes the line longer than the maximum permitted length, then the character is ignored.
	X	inserts text at the end of the specified line. The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command was given. When you are finished extending the line, press RETURN.
delete text	D	deletes individual characters. [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

ACTION	SUBCOMMANDS	DESCRIPTION OF SUBCOMMAND
delete text (cont.)	H	deletes all characters from the cursor position to the end of the line. H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.
find text	S	searches for a specified character. The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, then the cursor stops at the end of the line. All characters passed over during the search are printed.
	K	searches for a specified character and deletes all characters passed over. The subcommand [i]S<ch>, except all characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

ACTION	SUBCOMMANDS	DESCRIPTION OF SUBCOMMAND
replace text	C	changes next character(s) to specified character(s). The Subcommand C<ch> changes the next character to <ch>. To change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you return to edit mode.
end and restart edit mode	RETURN	Pressing RETURN prints the remainder of the line, saves the changes you made, and exits edit mode.
	E	The E subcommand has the same effect as RETURN, except the remainder of the line is not printed.
	Q	The Q subcommand returns to BASIC command level, without saving any of the changes that were made to the line during edit mode.
	L	The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in edit mode. L is usually used to list the line when you first enter edit mode.
	A	The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

If BASIC receives an unrecognizable command or illegal character while in edit mode, it sounds an alarm and the command or character is ignored.

Syntax Errors: When a syntax error is encountered during execution of a program, BASIC automatically enters edit mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type RETURN (or the E subcommand), BASIC reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first exit edit mode with the Q subcommand. BASIC returns to command level, and all variable values are preserved.

CODE-A: To enter edit mode on the line you are currently typing, type CODE-A. BASIC responds with a RETURN, an exclamation point (!), and a space. The cursor is positioned at the first character in the line. Proceed by typing an edit mode subcommand.

NOTE

Remember, if you just entered a line and wish to go back and edit it, the command "EDIT." enters edit mode at the current line. (the line number symbol "." always refers to the current line in BASIC.)

EXAMPLE	EXPLANATION
EDIT 40	Enters edit mode at line 40 of the program currently in memory.

END

END Form of Statement

900 END

DESCRIPTION

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

EXAMPLE

```
950 IF NUM = 999 THEN END
960 NUM = NUM + 1
        .
        .
        .
990 GOTO 950
```

EXPLANATION

Statement number 950 tests if the variable NUM is equal to 999. If it is, the program terminates.

ERASE

Form of Statement

100 ERASE array name, array name,...

DESCRIPTION

The ERASE statement eliminates specified arrays from a program.

Arrays may be redimensioned after they have been eliminated by an ERASE statement, or the previously allocated array space in memory can be used for other purposes.

If an attempt is made to redimension an array that was not erased, an error message results.

EXAMPLE	EXPLANATION
10 DIM A [20], B[100] 20 ERASE B 30 DIM B [5]	Arrays A and B are dimensioned with 20 and 100 elements, respectively. Array B is then eliminated by an ERASE statement. Array B is then redimensioned with 5 elements.

ERR/ERL

Form of Statement

100 ERR = errorcode

120 ERL = linenumber

DESCRIPTION

The ERR and ERL variables are used during an error-trapping routine. The variable ERR contains the error code for the error, and the variable ERL contains the number of the line in which the error was detected.

The ERR and ERL variables can not be assigned directly by the user. If you wish to assign a specific integer to a specific error condition, the ERROR command must be used. Once the integer is assigned, the ERR and ERL variables can be used to trap the condition.

The ERR and ERL variables are used in IF...THEN statements to direct program flow in the error-trap routine.

EXAMPLE

```
110 ON ERROR GOTO 140
120 INPUT "CURRENT COST"; C

130 IF C>BUDGET THEN ERROR 215
135 GOTO 160
140 IF ERR = 215 THEN PRINT
  "COST IS OVER BUDGET"
150 IF ERL = 130 THEN RESUME 120
160 END
```

EXPLANATION

The user defines an error in statement 130.
If this error code is encountered in line 140, the "cost over budget" message is displayed.
If error is encountered at line 130, program execution resumes at line 120.

ERROR

Form of Statement

20 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 greater than zero and less than 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 examples below.

EXAMPLE

```
10 A= 10
20 B = 5
30 ERROR A + B
40 END
```

EXPLANATION

This program results in the error message:

Fatal Error: String too long in line 30.

This is BASIC error message number 15.

```
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET?";B

130 IF B>5000 THEN ERROR 210
    .
    .
    .
400 IF ERR=210 THEN PRINT
    "HOUSE LIMIT IS $5000"
410 IF ERL=130 THEN RESUME 120
```

When the value of B is greater than 5000, the program will go to the error-trapping routine.

In line 400, the error message is printed, and then execution resumes at line 120.

FIELD

Form of Statement

240 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, or 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, the variable's pointer is moved to string space.

EXAMPLE	EXPLANATION
240 FIELD #1, 10 AS ID$, 20 AS PR$	Statement 240 allocates space in the random file buffer for data to be moved to/from file 1. The statement allocates the first 10 positions (bytes) in the random file buffer to the string variable ID$. The next 20 positions (bytes) are allocated to PR$.

FOR...NEXT

Form of Statement

10 FOR variable = initial value TO final value

20 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, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement.

The default increment is one. If step is negative,, 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 the 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, an error message results.

EXAMPLE	EXPLANATION
20 FOR INIT = 1 to 5 30 PRINT INIT; 40 NEXT INIT	The statements in this FOR...NEXT loop display the values 1 through 5
10 K=10 20 FOR I = 1 to K STEP 2 30 PRINT I; 40 NEXT I	The statements in this loop display the numbers 1, 3, 5, 7, 9.

GET

GET

Form of Statement

150 GET #file number

200 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. An invalid record number results in an error message.

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

EXAMPLE	EXPLANATION
60 GET #1, 24	Statement 60 reads record number 24 from random disk file 1.

GOSUB ... RETURN

Form of Statement

150 GOSUB line number

line number

410

420 SUBROUTINE

430

440 RETURN

DESCRIPTION

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

EXAMPLE

```
10 GOSUB 50
20 PRINT "THESE STATEMENTS":
30 PRINT "WILL BE DISPLAYED";
40 PRINT "ON REENTRY TO PROGRAM"
45 END
50 PRINT "YOU HAVE JUST";
60 PRINT "ENTERED A SUBROUTINE"
70 RETURN
```

EXPLANATION

This example branches to line 50 and displays:

YOU HAVE JUST ENTERED A SUBROUTINE

Control then transfers back to line 20 and displays the message:

THESE STATEMENTS WILL BE DISPLAYED ON REENTRY TO PROGRAM

GOTO

Form of Statement

10 GOTO line number

DESCRIPTION

The GOTO causes control to branch to the indicated line number. Program action then proceeds from that point in the program sequence.

EXAMPLE

```
10 SUM = 0
20 NUM = 1
30 SUM = NUM + SUM
40 PRINT SUM
50 IF SUM>10  GOTO 70
60 GOTO 20
70 END
```

EXPLANATION

This example calculates and displays the value of the variable SUM. If the value is greater than 10, the program terminates. Statement 60 transfers control back to statement 20, and the sequence of calculating and displaying SUM continues.

	Form of Statement
IF...THEN	20 IF expression THEN statement
	30 IF expression THEN line number
IF...THEN...ELSE	40 IF expression THEN statement ELSE statement
	50 IF expression THEN line number ELSE line number
IF...GOTO	60 IF expression GOTO line number
IF...GOTO...ELSE	70 IF expression GOTO line number ELSE statement
	80 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.

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 THAN" 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.

If an IF...THEN statement is followed by a line number in direct mode, an "Undefined line" error message results unless a statement with the specified line number was previously entered in the indirect mode.

NOTE

When using IF to test equality for a value that is the result of a floating-point computation, remember that 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	EXPLANATION
60 IF A>B THEN PRINT "A is GREATER THAN B" ELSE PRINT "A is LESS THAN OR EQUAL TO B"	This statement evaluates the expression A is greater than B (A>B). If true, the message, "A is GREATER THAN B" is displayed. If not true, the other portion of the statement displays: A is LESS THAN OR EQUAL TO B

INPUT

Form of Statement

10 INPUT variable1, variable2, . . .

or

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

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 statement 20 above, the message is displayed before the question mark.

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.

EXAMPLE	EXPLANATION
10 INPUT VAR1,VAR2 20 SUM=VAR1+VAR2 30 PRINT SUM	This program prompts the user with a question mark. When the user enters two numbers, the values are assigned to VAR1 and VAR2 respectively. The program then displays their sum.
10 INPUT "ENTER TWO VALUES"; VAR1,VAR2 20 SUM=VAR1+VAR2 30 PRINT "THE SUM IS";SUM	This program produces the same result as the program above but prompts the user ENTER TWO VALUES. The result is preceded by: THE SUM IS

INPUT#

Form of Statement

100 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. With numeric values, leading spaces, RETURN's, and BOUND's are ignored. 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, RETURNs, and BOUNDs 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, the item is terminated.

EXAMPLE	EXPLANATION
210 OPEN "I", #1, "DATA" 220 INPUT #1, A, B, C	Statement number 220 reads three values from file number 1 and assigns them to the program variables A, B, and C. The file number (#1) is the file opened in statement 210.

KILL

Form of Statement

50 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 not currently open.

An error message results if a KILL statement is given for a file currently open or for a nonexistent file.

EXAMPLE	EXPLANATION
400 KILL "MYDATA1"	This statement deletes the file MYDATA1 from the disk.

LET

Form of Statement

10 LET variable = expression

or

20 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 expression are sufficient for assignment statements.

EXAMPLE	EXPLANATION
10 LET A=10 20 LET B = 5 30 LET SUM=A+B	Each of these statements assigns a value to a variable. Statement 10 assigns the value 10 to the variable A.
100 A=10 120 B=5 130 SUM=A+B	Statement 20 assigns the value 5 to the variable B. Statement 30 assigns a value that results from the expression A + B to the variable SUM. In this example, SUM is assigned the value 15. Statements 100, 120, and 130 perform the same actions.

LINE INPUT

Form of Statement

110 LINE INPUT string variable

111 LINE INPUT "prompt string";
string variable

DESCRIPTION

The LINE INPUT statement inputs an entire line (up to 254 characters) and assigns it to a string variable. The second form of the LINE INPUT statement provides a message on the video display before the line to be entered.

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

```
10 REM USING THE LINE INPUT statement
20 LINE INPUT "ENTER CUSTOMER
   INFORMATION"; A$
30 PRINT A$
```

EXPLANATION

This program displays the following message at the video display:

ENTER CUSTOMER INFORMATION

The user can then enter the required customer information in the form:

ABC CORP ACCOUNT NO. 1234

All this information is assigned to the string variable A$.

Statement 30 displays:

ABC CORP ACCOUNT NO. 1234

LINE INPUT#

LINE INPUT #

Form of Statement

230 LINE INPUT # file number,
string variable

DESCRIPTION

The LINE INPUT # statement reads an entire line (up to 254 characters) 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.

EXAMPLE

```
10 OPEN "O",#1, "ACCTS"
20 LINE INPUT "ENTER
   CUSTOMER INFORMATION",A$
30 PRINT #1, A$


40 CLOSE #1
50 OPEN "I",#1,"ACCTS"
60 LINE INPUT #1,B$
70 PRINT B$
80 CLOSE #1
```

EXPLANATION

This example creates the sequential file ACCTS. The user then asked in statement 20 to ENTER CUSTOMER INFORMATION. The following is an example entry that is assigned to variable A$.

ABC CORP, ACCOUNT NO. 1234

This information is then written to sequential file number 1 under the file name ACCTS.

The file is closed in statement 40 and reopened in statement 50 so that it can be read.

Statement 60 reads the entire line of information and assigns it to the string variable B$. The following is displayed when line 70 is executed.

ABC CORP, ACCT NO 1234

Statement 80 closes the sequential file.

LIST

LIST	Form of Statement
	LIST
	LIST line number
	LIST line number - line number
	LIST line number-
	LIST -line number

DESCRIPTION

The LIST command lists all or part of a program currently in memory.

The first form lists the entire program.

The second form lists a specified line number.

The third form lists the program lines in the specified range. If only the first number of the range is specified, that line and all following program lines are listed. If only the second number of the range is specified, all lines from the beginning of the program up to and including the line specified are listed.

BASIC always returns to command level after a LIST is executed.

EXAMPLE	EXPLANATION
LIST	Lists the entire program.
LIST 100	Lists line 100 of the program.
LIST 100 - 500	Lists line 100 through 500 of the program
LIST 100 -	Lists all lines from statement 100 to the end of the program.
LIST - 500	Lists all lines from the beginning statement up to and including line 500 of the program.

LLIST

LLIST	Form of Statement
	LLIST
	LLIST *line number*
	LLIST *line number* - *line number*
	LLIST *line number*-
	LLIST -*line number*

DESCRIPTION

The LLIST command lists all or part of a program currently in memory on the line printer.

The first form lists the entire program on the line printer. The second form lists only the specified line number on the line printer. The third form lists the specified range of program lines on the line printer.

If only the first number of the range is specified, that line and all following program lines are listed. If only the second number of the range is specified, all lines from the beginning of the program up to and including the line specified are listed.

LLIST assumes 132 character wide printer and always returns to command level after an LLIST is executed.

EXAMPLE	EXPLANATION
LLIST	Lists the entire program on the line printer.
LLIST 100	Lists line 100 of the program on the line printer.
LLIST 100 - 500	Lists lines 100 through 500 of the program on the line printer.
LLIST 100 -	Lists all lines from statement 100 to the end of the program on the line printer.
LLIST - 500	Lists all lines from the beginning statement up to and including line 500 of the program on the line printer.

LOAD

Form of Statement

LOAD "filename"

or

LOAD "filename", R

DESCRIPTION

The LOAD command brings a program from disk into memory. This option closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program.

If the second form of the command is used, the program is run immediately after it is brought into memory. This option also keeps all open data files open. LOAD, with option R, may be used to chain programs.

EXAMPLE	EXPLANATION
LOAD "MYPROG"	Brings the program MYPROG into memory.
LOAD "MYPROG", R	Brings the program MYPROG into memory and runs the program once it is loaded.

LPRINT

LPRINT

Form of Statement

150 LPRINT

150 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 but can be modified by the use of the WIDTH command.

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

EXAMPLE	EXPLANATION
10 A=5 20 B=10 30 LPRINT A,B,A+B	This displays the values of variables A and B and the value which results from the expression A + B
	The result of statement 30 would be 5 10 15

LPRINT USING Form of Statement

200 LPRINT USING "formatting expression"; list of expressions

DESCRIPTION

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

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

```
10 A=10
20 B=5.8
30 LPRINT USING "##.##";
   A,B,A+B
```

EXPLANATION

Statement 30 displays the values of variables A and B and also the results of the expression A+B on the line printer. These values are displayed in the specified format.

The result of statement 30 is in the format:

```
 10.00      5.80      15.80
```

LSET AND RSET

LSET and RSET

Form of Statement

150 LSET string variable = string expression

180 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, 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	EXPLANATION
100 N$ = "JANE DOE" 110 A$ = SPACE$(20) 120 rset A$=N$ 130 PRINT A$	This example right-justifies the string "JANE DOE" in a 20-character field.
10 N$ = "BASIC REFERENCE MANUAL" 20 A$ = SPACE$(20) 30 RSET A$=N$ 40 PRINT A$	This example right-justifies the string N$ and truncates it as follows: BASIC REFERENC MANUAL
150 LSET A$=MKS$(RECVBL)	This statement performs the required conversion of a numeric variable (RECVBL) to a string and then moves the resulting string variable A$ to the random buffer and left-justifies it.

MERGE

Form of Statement

MERGE "file name"

DESCRIPTION

The MERGE statement merges the specified disk file into the program currently in memory. If any of the line numbers of the disk file are the same as the line numbers of the program in memory, the lines from the disk file replace the program lines. Lines different from the program statement numbers are merged according to line number order.

The "file name" is the name used when the file was SAVEd. The file must have been SAVEd in ASCII format, or an error message results.

BASIC always returns to command level after executing a MERGE command.

EXAMPLE	EXPLANATION
MERGE "ADDON"	This statement merges the disk file ADDON with the program currently in memory.

MID$

Form of Function

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

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

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

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

MID$

Form of Statement

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

20 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	EXPLANATION
10 A$ = "SALEM, OR" 20 MID$ (A$,8)="MA" 30 PRINT A$	Statement 20 replaces the portion of the original string beginning in the eighth position with the entire new string. Statement 30 now displays: SALEM, MA
100 A$="SOUTH DAKOTA" 110 MID$(A$,1,3)="NORTH" 120 PRINT A$	Statement 110 replaces the portion of the original string beginning in the first position with the new string for 3 characters. Statement 120 now displays NORTH DAKOTA.

NAME

NAME — Form of Statement

NAME "old file name" A$
"new file name"

DESCRIPTION

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

EXAMPLE	EXPLANATION
NAME "MYFILE"AS"LEDGER"	This command renames the file called MYFILE to a new file called LEDGER. All file characteristics other than the name remain the same.

NEW Form of Statement

NEW

DESCRIPTION

The NEW command deletes a program from memory and returns BASIC to command level.

EXAMPLE	EXPLANATION
LOAD "MYPROG" OK AUTO 10* <CANCEL> NEW AUTO 10	The program MYPROG is loaded in memory. When the AUTO command is entered, 10* indicates that this line number was already in memory. AUTO is then cancelled and memory is cleared with the NEW command. The AUTO command now begins numbering program lines with 10.

ON ERROR GOTO

ON ERROR GOTO

Form of Statement

10 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 shift program action 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. An ON ERROR GOTO 0 statement that appears in an error-trapping subroutines causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error-trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE

If an error occurs during execution of an error-handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error-handling subroutine.

EXAMPLE	EXPLANATION
50 ON ERROR GOTO 1500	When an error is encountered, control shifts to line 1500, the first line of an error-handling routine.

ON...GOSUB

Form of Statement

100 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 expression is 0 or greater than the number of entries in the line number list, control passes to the next executable statement. If the value of the expression is negative, or greater than 255, an error message results.

EXAMPLE	EXPLANATION
40 ON A GOSUB 100,300,550,1400	If A equals 1, control transfers to line 100 If A equals 2, control transfers to line 300 If A equals 3, control transfers to line 550 If A equals 4, control transfers to line 1400 When the subroutine has been executed, control returns to the statement immediately following the ON GOSUB statement.

ON...GOTO

Form of Statement

10 ON expression GOTO
line number list

DESCRIPTION

The ON...GOTO command branches the program to one of the specified line numbers.

The expression controls which line number the program will branch to. 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 or greater than 255, an error message results.

EXAMPLE	EXPLANATION
50 M=4 55 N=2 60 ON M-N GOTO 100,200,300	Statement 60 evaluates the expression M-N. The resulting integer 2 transfers control to statement number 200.

OPEN

Form of Statement

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

200 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 15. The number is then associated with the file for as long as it is open.

"filename" is a string variable having a maximum of 50 characters.

An optional record length may be specified for random access files. The default record length is 128 characters. To change the default record length, refer to "Invoking BASIC" in Section 1.

EXAMPLE

```
100 OPEN "A",#2,"LEDGER"
200 OPEN "O",#1,"PAYABLE"
300 OPEN "I",#5,"RECEIVABLE"
400 OPEN "R",#3,"CUSTOMERS"
500 OPEN "S",#4,"MAIL LIST",248
```

EXPLANATION

Statement 100 opens a sequential disk file LEDGER with file number 2, so that data may be appended to its existing contents. Statement 200 opens a sequential disk file PAYABLE with file number one, so that data may be written to it. If the file PAYABLE contains data, it will be lost. Statement 300 opens a sequential disk file RECEIVABLE with file number 5. So that data may be read from it.

Statement 400 opens random file #3 named CUSTOMERS, for input or output. The maximum record length by default is 128 characters (bytes).

Statement 500 opens random file #4, named MAIL LIST, for input. The maximum record length is set at 248 character (bytes).

OPTION BASE Form of Statement

```
10 OPTION BASE 1

20 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 reserves space for these locations:

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

OPTION BASE 0 reserves space for these locations:

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

EXAMPLE	EXPLANATION
10 OPTION BASE 1 20 DIM A[20],B[5]	Statement 10 sets the minimum value for the subscripts in arrays A and B at 1.

NOTE

Refer to the section that discusses the CHAIN command for the effect chaining has upon OPTION BASE.

OPTION COMMA

OPTION COMMA Form of Statement

10 OPTION COMMA [,S$]

DESCRIPTION

The input and output of numeric data lists during program execution, OPTION COMMA changes the physical representation of the following items:

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. Logcal 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 parameter 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 not a null string, OPTION COMMA resets the above items based on the first four characters in the S$. If S$ contains fewer than four characters, the new default values are used for the unspecified items.

EXAMPLE

```
10 OPTION COMMA
20 INPUT X,Y
30 PRINT USING "$$##,##.##";X,Y
40 PRINT X,Y
50 WRITE X,Y
```

EXPLANATION

```
If the input was
12345,67/76543,21 then the
output will be $12.345,67
$76.543,21
12345,67    76543,21
12345,67 / 76543,21
```

OUT

Form of Statement

100 OUT"B", port number,integer expression

or

200 OUT"W", port number,integer expression

DESCRIPTION

The OUT command has two forms. OUT"B" 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. Both the port number and the integer expression must be in the range of 0 to 65535, and the port number must be even.

EXAMPLE	EXPLANATION
100 OUT "B",32,100 200 OUT "W",64,&HFFFF	Statement 100 sends the byte information to byte port 32. Statement 200 sends the word information concerning the hexadecimal integer to port number 64.

POKE

POKE

Form of Statement

100 POKE"B", single precision pointer, integer expression

200 POKE"W", single precision pointer, integer expression

DESCRIPTION

The two 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 to be poked into the specified memory location.

EXAMPLE

```
100 POKE"B",PTR(M%),&HFF
200 POKE"W",PTR(N%),&HFFFF
```

EXPLANATION

Statement 100 "pokes" the byte representation of the hexadecimal variable into the memory location specified by the pointer.

Statement 200 "pokes" the word representation of the hexadecimal variable into the memory location specified by the pointer.

PRINT

Form of Statement

50 PRINT list of expressions

DESCRIPTION

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

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

Separating by semicolons displays each value immediately after the last value. Separating by commas displays each value to be displayed in the print zones. BASIC divides the line into print zones of 14 spaces each. If the list of expressions is omitted, a blank line is printed.

A question mark (?) can be used in place of the word "print" in a Print statement.

EXAMPLE

```
10 A=5
20 B=10
30 PRINT "THE SUM IS";A+B
```

EXPLANATION

This example displays the following:

THE SUM IS 15

```
10 X = 5
20 PRINT X+5,X-5,X>(-5),XX5
30 END
```

In this example, the commas print each value at the beginning of the next print zone

```
 10      0      -1      0
```

```
10 FOR X=1 TO 3
20 J=J+5
30 K=K+10
40 PRINT J;K;
50 NEXT X
```

In this example, the semicolons print each value immediately after the preceding value.

```
 5    10    10    20    15    30
```

PRINT USING

PRINT USING Form of Statement

20 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, two characters are printed; with one space, three characters are printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, 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:

#	Represents each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.
.	Can be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit always is printed (as 0 if necessary). Numbers are rounded as necessary.

PRINT USING (Cont.)

+	At the beginning or end of the format string, prints the sign of the number (plus or minus) before or after the number.
-	At the end of the format field, prints negative numbers with a trailing minus sign.
**	At the beginning of the format string, fills leading spaces in the numeric field with asterisks. The ** also specifies positions for two more digits.
$$	Prints a dollar sign to the immediate left of the formatted number. The $$ 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.
**$	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.
,	To the left of the decimal point in a formatting string, prints a comma 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.
^^^^	Can be placed after the digit position characters to specify exponential format. The four carats (or up-arrows) 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.)

_ 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, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign is printed in front of the rounded number.

```
EXAMPLE                                  EXPLANATION

120 PRINT USING "##.##"; 10.5            Video Display Shows:   10.50
150 PRINT USING "$$###.##"; 210.15       Video Display Shows: $210.15
180 PRINT USING "####,.##"; 1234.5       Video Display Shows: 1,234.50
210 PRINT USING "##.#-";-10.5            Video Display Shows: 10.5-
260 PRINT USING "##.##^^^^";234.56       Video Display Shows: 2.35E+02
280 PRINT USING "+##.#"; 10.5            Video Display Shows: +10.5
310 PRINT USING "+##.## ";-68.95         Video Display Shows: -68.95
320 PRINT USING "**#.#";12.39            Video Display Shows: *12.4
330 PRINT USING "**$##.##";2.34          Video Display Shows:***$2.34
340 PRINT USING "####,.##";1234.5        Video Display Shows: 1,234.50
350 PRINT USING "!##.##!";"12.34"        Video Display Shows: -1
360 PRINT USING "##.##";111.22           Video Display Shows: %111.22

10 A$="LOOK":B$="OUT"                    Video Display Shows:
30 PRINT USING "!";A$;B$                        LO
40 PRINT USING "\    \";A$;B$                   LOOK  OUT
50 PRINT USING "\    \";A$;B$;"!!"              LOOK  OUT !!

10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;                   Video Display Shows:  LOUT
30 PRINT USING "&";B$
```

PRINT#

Form of Statement

180 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.

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, take care 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, 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.)

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, 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$;B$
```

writes the following image to disk:

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

The statement

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

inputs "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"
```

The statement

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

inputs "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
```

EXAMPLE	EXPLANATION
10 A = 2 20 B = 3 30 PRINT #1, A; B; A+B	This program segment outputs the following information to sequential file number 1: 235 Notice that a later INPUT# statement cannot separate the three values.
10 A$ = ACCOUNTS 20 B$ = PAYABLE 30 PRINT #2, A$; ","; B$	This program writes the following image to sequential file number 2 ACCOUNTS,PAYABLE If the comma enclosed in quotation marks had been omitted, the image ACCOUNTSPAYABLE would have been written to the disk. This could not be recovered as the two variables A$ and B$.

PRINT# USING

PRINT# USING

Form of Statement

200 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

```
230 OPEN "O", #2, "MYFILE"
240 A = 240
250 B = 860
260 C = 10.5
270 PRINT #2, USING "$$###.##"; A;B;C
```

EXPLANATION

This program writes the following image to sequential file number 2:

```
 $240.00 $860.00 $10.50
```

PUT

Form of Statement

250 PUT #file number

260 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	EXPLANATION
260 PUT #1, CODE	Writes a record to random disk file number 1. The record number for this record is the value of CODE.

RANDOMIZE

RANDOMIZE

Form of Statement

10 RANDOMIZE

20 RANDOMIZE seed value expression

DESCRIPTION

The RANDOMIZE command is used to establish a beginning point (seed) for the random number generator. RANDOMIZE is used with the RND function. Changing the seed value is necessary if a random sequence of numbers is to be obtained.

If the seed value is not changed, the RND function returns the same sequence of random numbers each time the program is run.

When the RANDOMIZE command is used without a seed value expression, as in statement 10 above, BASIC asks for a seed value by displaying the message:

```
RANDOM NUMBER SEED (-32768 to 32767)?
```

The use of the seed value expression allows the user to change the sequence of random numbers generated during program execution.

EXAMPLE	EXPLANATION OF EXAMPLES
`10 RANDOMIZE` `20 PRINT RND`	This program first displays the message RANDOMIZE NUMBER (-32768 to 32767)?

`10 FOR I = 1 to 5`
`20 RANDOMIZE I`
`30 PRINT RND;`
`40 NEXT I`
	This program varies the seed value from 1 to 5 and produces a random number sequence.

READ

Form of Statement

10 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, an error message results.

EXAMPLE	EXPLANATION
10 READ A, AB$, B 90 DATA 3.5, "CUSTOMER ACCT. NO.", 1000	This program assigns the value 3.5 to variable A, the value CUSTOMER ACCT. NO. to the variable AB$, and the value 1000 to variable B.

REM

REM Form of Statement

10 REM programmer-supplied remarks

20 REM '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

```
10 REM THIS PROGRAM CALCULATES
     THE MONTHLY
20 REM PAYMENTS ON LOANS
     USING DIFFEREN
30 REM INTEREST RATES AND DIFFERENT
40 REM LOAN TERM
50 INPUT "ENTER THE INTEREST RATE";A
     'A = Interest rate
```

EXPLANATION

These four REM statements appear only in the program listing. When this program is run, the first message the user sees is:

```
ENTER THE INTEREST RATE
```

RENUM

Form of Statement

RENUM

RENUM first line number to be used in new sequence, line number where renumbering is to begin, increment

DESCRIPTION

The RENUM command is used to renumber program lines. When program line numbers are changed, all references to them are also changed. For example, any line number references in GOSUB, GOTO, ON...GOSUB, ON...GOTO, or ERL statements are changed.

When RENUM is used without declaring the first line number to be used, renumbering begins with 10.

When RENUM is used without stating the line number where renumbering is to begin, the entire program is renumbered.

When the interval of the line numbers is not specified, each line number increases by 10.

RENUM cannot change the order of program lines, nor can it create line numbers greater than 65529. An error message results in both cases.

EXAMPLE	EXPLANATION
RENUM	Renumbers the entire program. The first line is statement number 10, and each following statement number is incremented by 10.
RENUM 100, 10, 50	This command renumbers the existing program beginning at line number 10. This line is renumbered as line number 100. Each following line number is incremented by 50.
RENUM 300,20	This command renumbers the existing program. The first new line number is 300 and increments each following line by 20.

RESTORE

RESTORE

Form of Statement

210 RESTORE

220 RESTORE line number

DESCRIPTION

The RESTORE command moves the data stack pointer to a specified line. When no line number is used, the data stack pointer is moved 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 RESTORE statements, the data stack pointer moves to the first value in the line specified.

EXAMPLE	EXPLANATION
10 READ A, B, C, D, E, F 20 RESTORE 30 READ G, H, I, J, K, L 40 DATA 2, 4, 6 50 DATA 8, 10, 12	This program assigns the values 2, 4, 6, 8, 10, 12 to A, B, C, D, E, F respectively. The RESTORE command moves the data stack pointer back to statement 40, and the same sequence of values is assigned to the variables G, H, I, J, K, L.

RESUME

Form of Statement

10 RESUME

20 RESUME 0

30 RESUME NEXT

40 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 not in an error-handling routine displays the error message RESUME without error.

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 that caused the error.
2. When the form is RESUME NEXT, program execution resumes at the statement that immediately follows the one that caused the error.
3. When the form is RESUME line number, program execution resumes at the specified line number.

EXAMPLE	EXPLANATION
10 ON ERROR GOTO 500	On encountering an error, this program segment transfers control to line number 500.
500 IF ERR = 230 THEN PRINT "Your ENTRY is TOO LARGE"	If the error is error 230, the message "Your ENTRY is TOO LARGE" is displayed.
510 RESUME 80	Control then transfers back to line 80 as directed in the RESUME statement.

RUN

RUN

Form of Statement

RUN

RUN line number

RUN "filename"

RUN "filename", R

DESCRIPTION

The RUN command may be used to execute a program currently in memory or to load a program from a disk into memory and execute it.

Used alone, the RUN command executes the program currently in memory beginning at the lowest statement number.

The RUN line number statement executes the program currently in memory beginning at the line number specified.

RUN "filename" loads a program from disk into memory and executes it. Before doing this, memory is cleared and all open files are closed.

The RUN "filename", R form has the same action as RUN "filename" but all data files remain open.

EXAMPLE	EXPLANATION
RUN	This command runs the program currently in memory from the beginning.
RUN 1001	This statement runs the program currently in memory beginning executing at line 1001.
RUN "MYPROG"	This statement deletes the contents of memory, closes all files, and executes the program MYPROG.

SAVE	Form of Statement
	SAVE "filename"
	SAVE "filename", A
	SAVE "filename", P

DESCRIPTION

The SAVE command saves a program on disk in one of three forms:

1. SAVE "filename" saves the file in a compressed binary format. This format makes efficient use of disk space, but certain commands require ASCII format. The MERGE command in BASIC requires ASCII format as does the B 20 executive command TYPE. If a file is to be edited using the B 20 Editor, it must be saved in ASCII format.

2. SAVE "filename", A saves the program in ASCII format.

3. SAVE "filename", P saves the program in an encoded binary format. This format allows the program to be used with the RUN or LOAD commands, but LIST or EDIT commands will fail.

EXAMPLE	EXPLANATION
SAVE "MYPROG"	Saves the program MYPROG in compressed binary format.
SAVE "MYPROG", A	Saves the program MYPROG in ASCII format.
SAVE "MYPROG",P	Saves the program MYPROG in encoded binary format.

STOP

STOP

Form of Statement

40 STOP

DESCRIPTION

The STOP command halts program execution. STOP may be used anywhere in a program. When STOP is encountered, the message "BREAK IN LINE line number" is printed. Program execution may be resumed by using the CONT command.

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

EXAMPLE

```
10 DIM A [5]
20 FOR I = 1 to 5
30 READ A [I]
40 NEXT I
50 STOP
   .
   .
   .
900 DATA 2, 4, 6, 8, 10
RUN
BREAK IN 50
PRINT A [1]
```

EXPLANATION

This section of a program reads the five elements of any array A. When STOP is encountered, the message BREAK IN LINE 50 is printed at the video and program execution stops. At this point, the user may wish to check an array value with a direct mode statement such as PRINT A [1].

If the user checks this value successfully, program execution may be resumed with a CONT command.

SWAP

Form of Statement

20 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

```
10 A$ = "OVER"
20 B$ = "LOOK"
30 C$ = "THIS REPORT"
40 PRINT B$ A$ C$
50 SWAP A$, B$
60 PRINT B$ A$ C$
```

EXPLANATION

When this program is executed, it first displays:

LOOK OVER THIS REPORT

At Line 50 the values of variables A$ and B$ are exchanged and the same display statement now causes OVER LOOK THIS REPORT to be displayed.

SYSTEM

Form of Statement

```
10 SYSTEM
```

DESCRIPTION

The SYSTEM command exits from the program. During input the same action may be accomplished by pressing the FINISH key twice.

EXAMPLE

```
10 DIM B$ [10]
20 INPUT A$

30 IF A$ = "END" THEN SYSTEM
40 B$ [1] = LEFT$ (A$, 1)
```

EXPLANATION

This section of a program prompts for the input of a string variable. If the user enters the character string END, the SYSTEM command causes BASIC to be exited. Otherwise, the program continues.

TRON/TROFF

Form of Statement

TRON

TROFF

DESCRIPTION

The TRON command traces the execution of program statements by printing the program statement line numbers as they are executed. The line numbers appear enclosed in square brackets. Variables output with print statements are printed at the point of execution.

The tracing action of the TRON command may be stopped by entering a TROFF command.

The TRON and TROFF commands are used from the BASIC command level.

EXAMPLE

```
OK
LIST
10 VAR1 = 2
20 FOR I = 1 to 2
30 VAR2 = VAR1+2
40 PRINT I; VAR1; VAR2
50 VAR1 = VAR1 +10
60 NEXT I
70 END

OK
TRON
RUN
[10] [20] [30] [40] 1  2  4
[50] [60] [30] [40] 2  12 14
[50] [60] [70]

OK
TROFF
RUN
1  2  4
2  12 14
```

EXPLANATION

The example first lists a short program. The TRON command is entered and the program is executed with the RUN command.

The video output is shown. The bracket numbers are the statement numbers presented in the order or their execution.

Stopped the trace by entering a TROFF command, and the program is rerun.

WAIT

Form of Statement

WAIT"B", port number, integer expression1, integer expression2

WAIT"W", port number, integer expression1, integer expression2

DESCRIPTION

The WAIT command has two forms. WAIT "B" is used to suspend program execution while monitoring the status of an 8086 byte port. The port number must be in the range of 0 to 65535, and integer expressions 1 and 2 must be in the range of 0 to 255.

WAIT "W" is used to suspend program execution while monitoring the status of an 8086 word port. The port number as well as the integer expressions 1 and 2 must be in the range 0 to 65535, and the port number must be even.

The WAIT statement suspends execution until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive ORed with the integer expression2 and then ANDed with integer expression1. If the result is 0, then BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If integer expression2 is omitted, it is assumed to be 0.

In the second format, an error message ocurs if the port number is odd.

CAUTION

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

EXAMPLE	EXPLANATION
100 WAIT "B" ,32,2 200 WAIT "W", 3033,&HFFFF,14	Program execution is suspended while the specified 8086 ports are monitored. In statement 100 a byte port, 32, is monitored while in statement 200 a word port, 3033, is monitored.

WHILE...WEND

Form of Statement

```
10  WHILE expression
      .
      .
      .
 STATEMENTS
      .
      .
      .
 50 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. WHILE...WEND loops can be nested to any level.

EXAMPLE

```
10 NUM = 10
20 WHILE NUM<100
30 FOR I = 1 to 10
40 PRINT "NUMBER"; NUM,
50 NUM = NUM + 20
60 NEXT I
70 WEND
```

EXPLANATION

This program executes the FOR...NEXT loop 10 times. The value of NUM is printed only when its value is less than 100.

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, however, the number of characters per line depends on the characteristics of the printer. If the value is not within the specified range (15-132), no action occurs. If your video display has a width of 80 characters, which is the width for a B 21, a width greater than 80 is ignored.

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

EXAMPLE	EXPLANTION
10 PRINT "HOW WIDE IS MY SCREEN?" RUN HOW WIDE IS MY SCREEN? OK WIDTH 15	Statement 10 prints the message HOW WIDE IS MY SCREEN? This message is 22 characters long (counting spaces and the question mark).
OK RUN Ok HOW WIDE IS MY SCREEN? OK	When the WIDTH command specifies that only 15 characters may be printed on a line, the message is changed as shown.

WRITE

Form of Statement

```
30 WRITE
40 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

```
10 A = 10
20 B = 15
30 C$ = "CUSTOMER"
40 PRINT A,B,C$
50 WRITE A,B,C$
```

EXPLANTION

This program assigns values to variables A, B, and C$. It then outputs results using a PRINT statement and a WRITE statement.

Statement 40 outputs the following:

```
 10   15   CUSTOMER
```

Statement 50 outputs as:

```
10,15,"CUSTOMER"
```

Notice the difference in the output format.

WRITE# Form of Statement

250 WRITE# filenumber, list of expressions separated by commas

DESCRIPTION

The WRITE# command writes data to the sequential file specified by the filenumber. The filenumber 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. WRITE# offers the advantage of not requiring the user to insert specific delimiters when writing to sequential files.

EXAMPLE

```
5 OPEN,"O",#1,"DATA"
  .
  .
  .
10 A$ = "ACCOUNTS"
20 B$ = "PAYABLE"
30 PRINT #1, A$; B$
40 WRITE #2, A$, B$
  .
  .
  .
```

EXPLANTION

Statement 30 outputs the following to file 1:

ACCOUNTSPAYABLE.

This cannot be read as two variables with an INPUT# statement.

Statement 40 outputs the following to file 2:

ACCOUNTS, PAYABLE

SECTION 3
BASIC FUNCTIONS

This section defines all the BASIC functions that are supported by the BASIC Interpreter. Each function is listed alphabetically and is presented with the following information:

- an example of the general form in which the function is used
- a description that explains how each function works
- an example of the function in a program
- an explanation of the example describing the action caused by the function

ABS

ABS

Form of Function

10 ABS (numeric expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 A = 10 20 B = 15 30 PRINT A-B, ABS(A-B)	Statement 30 prints - 5 5

ASC

Form of Function

10 ASC (string expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 A$= "ARITHMETIC" 20 PRINT ASC(A$)	Statement 20 prints 65, the ASCII value of A.

ATN

ATN

Form of Function

20 ATN (numeric expression in radians)

DESCRIPTION

The ATN function returns the arctangent of the numeric expression. The numeric expression must 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	EXPLANATION
10 A = 3 20 PRINT ATN(A)	Statement 20 prints 1.249046, the arctangent of 3 radians.

CDBL

Form of Function

10 CDBL (numeric expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 A = 454.67 20 PRINT CDBL(A)	This program displays the value 454.6699829101563.

CHR$

CHR$

Form of Function

10 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 VDU screen.

EXAMPLE	EXPLANATION
10 PRINT CHR$(65)	This statement displays A, the equivalent of ASCII code 65.

CINT

Form of Function

10 CINT (numeric expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 PRINT CINT(48.33)	The integer 48 is displayed.
20 PRINT CINT(48.67)	The integer 49 is displayed.

COS

COS

Form of Function

10 COS (numeric expression in radians)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 X = 3 20 PRINT COS(X)	Statement 20 displays the value -.9899926, which is the cosine equivalent of 3 radians.

CSNG

Form of Function

10 CSNG (numeric expression)

DESCRIPTION

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

EXAMPLE

```
10 A# = 987.654321
20 B% = 2
30 PRINT A#;CSNG(A#);
   B%;CSNG(B%)
```

EXPLANATION

Statement 30 displays the double-precision value and its single-precision equivalent and then displays the integer and its single-precision equivalent. The screen displays:

```
987.654321    987.6543      2       2
```

CVI, CVS, CVD

CVI, CVS, CVD

Form of Function

10 CVI (2 byte string)

20 CVS (4 byte string)

30 CVD (8 byte string)

DESCRIPTION

CVI, CVS, and CVD 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.

Also see the subsections on MKI$, MKS$, and MKD$.

EXAMPLE	EXPLANATION
10 OPEN "R", #1, "MYFILE" 20 FIELD #1, 4 AS A$ 30 GET #1, 32 40 PRINT CVS(A$)	This program opens random file 1 and gets record number 32 from the random buffer. The print statement displays the reconverted number by using the CVS function. CVS is used because the variable A$ is a 4-byte, string as can be seen in the FIELD statement.

EOF

Form of Function

10 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	EXPLANATION
5 DIM VAR[1000] 10 OPEN "I",#1,"MYDATA" 20 FOR I = 1 to 1000 30 IF EOF(1)GOTO 60 40 INPUT #1,VAR(I) 50 NEXT I 60 STOP	This program segment reads values for the array VAR. The loop allows for 1000 input repetitions. If the end of the file is encountered before the 1000th record is read, control transfers to line 60.

EXP

EXP

Form of Function

10 EXP (numeric expression)

DESCRIPTION

The EXP function returns the value of e (e=2.71828) to the power of the numeric expression. The numeric expression must be less than or equal to 88.02969.

EXAMPLE	EXPLANATION
10 X = 4 20 PRINT EXP(X)	Statement 20 displays 54.59815.

FIX

Form of Function

10 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.

EXAMPLE	EXPLANATION
10 X = 54.38 20 PRINT FIX(X)	The print statement displays 54, the integer portion of the number 54.38.

FRE

FRE	Form of Function
	10 FRE (0)
	20 FRE (" ")

DESCRIPTION

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

The second format forces a garbage collection before returning the number of free bytes. This can take 1 to 1 1/2 minutes because BASIC does not initiate garbage collection until all free memory is used.

EXAMPLE	EXPLANATION
150 PRINT FRE(0)	This statement returns the amount of free memory (in bytes).

GETRA

Form of Function

10 GETRA (pointer)

DESCRIPTION

The GETRA function returns the relative address of a pointer. The pointer is a function that returns the address of the first byte of data of a variable.

EXAMPLE	EXPLANATION
10 I = 1 20 PRINT GETRA(PTR (I))	Statement 20 displays the relative address of variable I.

GETSA

GETSA

Form of Function

10 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	EXPLANATION
10 I = 1 20 PRINT GETSA(PTR(I))	Statement 20 displays the segment address of variable I.

HEX$

Form of Function

10 HEX$ (numeric expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 A = 32 20 PRINT A;HEX$(A)	Statement 20 displays 32 and 20, the decimal value and its hexadecimal equivalent.

INP

INP

Form of Function

10 INP ("B", port number)

20 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.

EXAMPLE	EXPLANATION
10 I = INP("B",255) 20 PRINT I	This program segment displays the byte read from port 255.
10 I = INP("W",3000) 50 PRINT I	Statement 50 displays the word read from port 3000.

INPUT$

Form of Function

10 INPUT$ (number of string characters)

20 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 (form 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

```
100 PRINT "TYPE CONTINUE
    TO GO TO NEXT LESSON"

110 X$= INPUT$(8)
120 IF X$<>"CONTINUE"THEN 999
   .
   .
   .
999 END
```

EXPLANATION

The user responds to the prompt displayed at the video by statement 100. If the user does not type CONTINUE, the program terminates.

```
10 OPEN "I",#1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC (INPUT$(1,#1)));
40 GOTO 20
50 PRINT "END OF FILE"
60 END
```

This example lists the contents of a sequential file, "DATA", in hexadecimal one character at a time.

INSTR

INSTR

Form of Function

10 INSTR (stringl, string2)

20 INSTR (position for starting search, stringl, 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 stringl, or if stringl is null, or if string2 cannot be found, then INSTR returns 0. Stringl and string2 can be string variables, string expressions or string literals.

EXAMPLE	EXPLANATION
10 A$= "CZECHOSLOVAKIA" 20 B$= "OSLO" 30 PRINT INSTR(A$,B$) 40 PRINT INSTR (8,A$,B$)	The function searches for string OSLO. The print statement in line 30 displays 6, the position at which OSLO begins. The print statement in line 40 displays 0 because "OSLO" was not found.

INT

Form of Function

10 INT (numeric expression)

DESCRIPTION

The INT function returns the largest integer less than or equal to the numeric expression.

EXAMPLE	EXPLANATION
10 PRINT INT(85.98)	This statement displays 85.
20 PRINT INT(-85.98)	This statement displays -86.

LEFT$

LEFT$

Form of Function

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

DESCRIPTION

The LEFT$ function returns the specified number of characters beginning at the left. The number of characters must be in the range of 0 to 255.

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, the null string is returned.

EXAMPLE	EXPLANATION
10 A$= "MASSACHUSETTS" 20 B$= LEFT$(A$,2) 30 PRINT B$	Statement 30 displays MA, the 2 leftmost characters of MASSACHUSETTS.

LEN

Form of Function

10 LEN (string expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 A$= "JOHN DOE" 20 PRINT LEN(A$)	This program displays 8, the number of letters including the space in JOHN DOE.

LOC

LOC

Form of Function

10 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

```
1200 IF LOC(1)>150 THEN
     PRINT "LAST ENTRY ON
     THIS DISK"
```

EXPLANATION

This statement can be used when writing to a random file. It displays the message:

LAST ENTRY ON THIS DISK

when the 150th record is read or written.

LOF

Form of Function

10 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	EXPLANATION
10 OPEN "S",#1,"MYFILE" 20 FOR I = 1 TO LOF(1) 30 GET #1 40 NEXT I	These statements open a random file named MYFILE and read each record of the file from beginning to end. Notice that this is done by using the LOF function as the final value in the FOR statement.

LOG

LOG

Form of Function

10 LOG (numeric expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 PRINT LOG(5 + 8/2)	This statement displays 2.197225, the natural logarithmic equivalent of (8/2 plus 5).

LPOS

Form of Function

10 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	EXPLANATION
240 IF LPOS(X)>120 THEN PRINT CHR$(7)	This statement causes the bell to sound when the print head is beyond the 120th position in the buffer.

MAKEPOINTER

MAKEPOINTER

Form of Function

10 MAKEPOINTER (relative address integer, segment address integer)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 PRINT MAKEPOINTER(0,0)	This statement displays the pointer corresponding to relative address 0, segment address 0.

MID$ Form of Function

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

DESCRIPTION

The MID$ function returns a character string that is a portion of the string expression specified in the function. The beginning position and length of character string values are integers in the range of 0 to 255.

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

The length of the character string to be returned is not specified; the function returns the portion of the string expression starting at the beginning position to the end.

EXAMPLE	EXPLANATION
10 A$= "ACCOUNTS " 20 B$= "PAYABLE RECEIVABLE" 30 PRINT A$;MID$(B$,1,8) 40 PRINT A$;MID$(B$,9,10)	Statement 30 displays ACCOUNTS PAYABLE. Statement 40 displays ACCOUNTS RECEIVABLE.

MKI$ MKS$ MKD$

MKI$ MKS$ MKD$

Form of Function

10 MKI$ (integer expression)

20 MKS$ (single-precision expression)

30 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 also subsections on CVI, CVS, and CVD.

EXAMPLE	EXPLANATION
10 OPEN "R",#1,"MYFILE" 20 FIELD #1,4 AS A$ 30 LSET A$= MKS$(AMOUNT) 40 PUT #1	This program opens random file number 1 and places in the buffer a single-precision variable AMOUNT. AMOUNT is first converted to a four-byte string variable. The four-byte string variable A$ is fielded in statement 20.

OCT$

Form of Function

10 OCT$ (numeric expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 A = 10 20 B = 9.9 30 PRINT A,OCT$(A),B,OCT$(B)	Statement 30 prints: 10 12 9.899999 12 Notice that the octal values are equivalent for the two variables. This is because variable B is rounded before its octal value is returned.

PEEK

PEEK

Form of Function

10 PEEK ("B", single-precision pointer)

or

20 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

```
 5 J% = 10858
10 A = PEEK("B",PTR(J%))
20 B = PEEK("W",PTR(J%))
30 PRINT A;B
```

EXPLANATION

Statement 30 displays the byte and the word read from the memory location specified by the pointer of the variable J:

```
 106     10858
```

POS

Form of Function

```
10 POS("X")
20 POS("Y")
```

DESCRIPTION

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

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

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

EXAMPLE	EXPLANATION
`10 IF POS("X")>60 THEN PRINT CHR$(35).`	If the cursor position is beyond column 60, # is displayed.
`20 IF POS("Y")>28 THEN PRINT CHR$(36).`	If the cursor position is beyond line 28, $ is displayed.

PTR

PTR

Form of Function

10 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 before 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	EXPLANATION
10 A% = 2 20 PRINT PTR(A%)	Statement 20 displays the address of the variable A%.

PWA

Form of Function

10 PWA (single-precision pointer)

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	EXPLANATION
10 DIM BUFFER%[512] 20 . 30 . 40 ERC%=OPENDAFILE ..., PWA(PTR(BUFFER%[0]))....)	Statement 10 allocates a 1026-byte buffer where only 1024 bytes are really needed. Statement 40 ensures that BUFFER%[0] starts on a word-aligned boundary, as required by the OPENDAFILE call.

RIGHT$

RIGHT$

Form of Function

10 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, the null string is returned.

EXAMPLE	EXPLANATION
10 A$= "BASIC LANGUAGE TRAINING" 20 PRINT RIGHT$(A$,8)	Statement 20 displays TRAINING

RND

Form of Function

10 RND

20 RND (numeric expression)

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	EXPLANATION
10 FOR I = 1 TO 5 20 PRINT RND 30 NEXT I	When this program is run it displays a sequence of random numbers. The sequence will be the same for each run.
5 RANDOMIZE 15 FOR I = 1 TO 5 25 PRINT RND 35 NEXT I	When this program is run it displays a sequence of random numbers that will be different for each run.

SGN

SGN

Form of Function

10 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

```
 10 A = 2
 20 ON SGN(A)+2 GOTO 100,200,300
    .
    .
    .
100 PRINT "THIS IS A NEGATIVE VALUE"
    .
    .
    .
200 PRINT "THIS VALUE IS ZERO"
    .
    .
    .
300 PRINT "THIS IS A POSITIVE VALUE"
    .
    .
    .
```

EXPLANATION

The SGN function in this program returns a value of 1. This value +2 transfers control to the third statement number in the GOTO portion of line 20. The displayed message is:

"THIS IS A POSITIVE VALUE"

SIN

Form of Function

10 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	EXPLANATION
10 PRINT SIN(1.7)	The value displayed (.9916648) is the sine of 1.7 radians.

SPACE$

SPACE$ — Form of Function

10 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	EXPLANATION
10 WIDTH 30 20 A$ = SPACE$(12) 30 PRINT A$;"CENTER"	This PRINT statement places the word CENTER in the middle of the declared WIDTH.

SPC

Form of Function

10 SPC(numeric expression)

DESCRIPTION

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

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

EXAMPLE	EXPLANATION
5 WIDTH 30 10 PRINT SPC(12);"CENTER"	This PRINT statement places the word CENTER in the middle of the declared screen width.

SQR

SQR

Form of Function

10 SQR(numeric expression)

DESCRIPTION

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

EXAMPLE

```
10 FOR I = 4 TO 8
20 PRINT I;SQR(I)
30 NEXT I
```

EXPLANATION

This program displays:

```
 4    2
 5    2.236068
 6    2.44949
 7    2.645751
 8    2.828427
```

This is the number and its square root.

STR$ Form of Function

10 STR$ (numeric expression)

DESCRIPTION

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

EXAMPLE	EXPLANATION
10 INPUT "ENTER ANY NUMBER"; ANYVAL 20 ANYVAL$= STR$(ANYVAL) 30 X = LEN(ANYVAL$) -1	These statements tell the user to enter any number. The number's string representation is assigned to ANYVAL$ in line 20. X would be equal to the length of the number.

STRING$

STRING$

Form of Function

10 STRING$ (character string length, ASCII code)

or

20 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

```
10 X$= "-"
20 A$= STRING$(10,45)
30 B$= STRING$(10,X$)
40 PRINT A$;"MONTHLY REPORT";B$
```

EXPLANATION

This program prints:

```
----------MONTHLY  REPORT----------
```

Notice that A$ and B$ have the same effect. A$ uses the ASCII code form.

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 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	EXPLANATION
10 ON ERROR GOTO 120	10 Set up error trap
20 KILL "TIMEFILE"	20 Remove (possibly nonexistant) old file
30 OPEN "A",#1,"TIMEFILE"	30 Open new file
.	
.	
.	
100 CLOSE #1	100 Close file opened in statement 30
110 END	110 End execution
120 PRINT "SYSTEM ERROR IS";SYSERC()	120 Display system status code
130 IF ERR = 53 THEN RESUME NEXT	130 If BASIC error is "file not found", then continue from line following error

TAB

TAB

Form of Function

10 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 before the current position, 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.

TAB is useful in formatting tables and reports.

EXAMPLE	EXPLANATION
10 PRINT TAB(1);"CUSTOMER"; TAB(12);"ACCOUNT NO."	This PRINT statement displays: CUSTOMER ACCOUNT NO.

TAN

Form of Function

10 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	EXPLANATION
10 PRINT TAN(1.7)	The value displayed is -7.696595, the tangent of 1.7 radians.

VAL

VAL

Form of Function

10 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 plus or minus sign, or a digit, the function returns the value 0.

NOTE

The percent sign (%) cannot be used within a VAL statement. An error message is displayed when it occurs within the string expression.

EXAMPLE	EXPLANATION
10 X$= "10" 20 Y = 5 30 X = VAL(X$) 40 PRINT X+Y	Line 40 displays 15. The string representation of 10 must be converted to a numeric variable before the calculation.

SECTION 4
BASIC ERROR MESSAGES

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

Cross Reference to Run-Time Errors (Cont.)

Message	Number
Internal error	51
Invalid parameter	31
Line buffer overflow	23
Malformed record	72
Missing operand	22
Pointer required	38
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

Message	Number
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

RUN TIME ERRORS

Number	Message
1	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.
- that the index variable of the FOR statement matches the variable in the NEXT statement.
- that the loop was not entered after the FOR statement.

Number	Message
2	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.
- that the punctuation used with the BASIC statement is correct.
- the use of spaces in the BASIC statement.
- that the number of open parentheses "(" is equal to the number of closed parentheses ")".
- that reserved words have not been used as variable names.

Number	Message
3	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.
- that the line number specified in the GOSUB statement is correct.
- that the subroutine has not been entered accidently (trace the program).

Number	Message
4	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	Message
5	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	Message
6	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.
- that any variable used in the calculation has the proper intermediate value. Print these variables before the calculation.

Number	Message
7	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	Message
8	Undefined line

A nonexistent line has been referenced in a BASIC statement.

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

Number	Message
9	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	Message
10	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.
- the DIM statements in each subprogram if a CHAIN command has been used.

Number	Message
11	Division by zero

A division by zero has been encountered in an expression.

Check:

- intermediate values in calculations.
- the order of the arithmetic operators used.

Number	Message
12	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	Message
13	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.

- When using LOAD or SAVE commands, make sure the file name is enclosed in quotation marks.

Number	Message
14	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	Message
15	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	Message
16	String formula too complex

A string expression is too long or too complex.

- Break the string expression into smaller expressions.

Number	Message
17	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	Message
19	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	Message
20	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	Message
21	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.
- 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	Message
26	FOR without NEXT

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

Check:

- that each FOR statement is paired with a NEXT statement.
- that the index of the FOR statement matches the variable in the NEXT statement.

Number	Message
29	WHILE without WEND

A WHILE statement is encountered without a matching WEND.

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

Number	Message
30	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	Message
31	Invalid parameter

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

Check:

- that the numbers you are using were input as numerics.
- that numeric values have not been converted to their string equivalents.

Number	Message
32	Unknown procedure

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

Check:

- the spelling of the called procedure.
- punctuation of the CALL statement.

Number	Message
33	Too much parameter data

BASIC passed more parameter data than the called procedure requires.

Check:

- the arguments of the CALL statement.
- the punctuation in the CALL statement.

Number	Message
34	Not enough parameter data

BASIC passed less parameter data than the called procedure requires.

Check:

- the arguments of the CALL statement.
- the punctuation in the CALL statement.

Number	Message
35	Procedure called as function

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

- Use the CALL command to invoke the procedure.

Number	Message
36	Function called as procedure

A value-returning procedure was invoked with a CALL statement.

- Invoke the procedure in an expression.

Number	Message
37	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	Message
38	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	Message
39	System error

The system has detected an error.

- Check system status code.

Number	Message
50	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	Message
51	Internal error

An internal malfunction has occurred on the specified disk.

- If this error occurs frequently, change the disk.

Number	Message
52	Bad file number

A statement references a file with an unopened file number.

- Check that an OPEN statement has been executed.

Number	Message
53	File not found

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

Check:

- the disk file contents.
- that the file in question is not a restricted-access file.

Number	Message
54	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	Message
55	File already open

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

Check:

- that sequential files in I or A mode have been closed before being reopened in O mode.
- that a KILL command was not used with an open file.

Number	Message
57	Disk I/O error

An input/output error occurred on a disk.

Check:

- to see if the floppy disk you are using is damaged.
- that the disk is write-enabled if you are writing to a floppy disk.

Number	Message
58	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	Message
61	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.

Number	Message
62	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	Message
63	Bad record number

In a PUT or 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 before the PUT or GET statement.

Number	Message
64	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	Message
66	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.
- that the file being loaded is not a data file.

Number	Message
67	Too many files

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

- Reexecute program with a new disk mounted.

Number	Message
68	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	Message
69	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	Message
70	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.
- the number of records in the random file using the LOF function.

Number	Message
71	Not a random file

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

- Check that the disk file is a random access file.

Number	Message
72	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	Message4
73	Read/write beyond EOF

An attempt to open a random file has failed 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.

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

	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"
```

Example

The following BASIC statements assign character attribute controls to string variables. When each of these string variables is used in a PRINT statement, the character attribute is activated.

```
 10 REVERSE$=CHR$(255)+"AE"
 20 UNDERLINE$=CHR$(255)+"AC"
 30 FLASH$=CHR$(255)+"AI"
 40 HALFBRIGHT$=CHR$(255)+"AB"
 50 NORMAL$=CHR$(255)+"AA"
 60 PRINT REVERSE$;"This is displayed in reverse video."
 70 PRINT UNDERLINE$;"This information is underlined."
 80 PRINT FLASH$;"This information is flashing."
 90 PRINT HALFBRIGHT$;"This information is displayed in
                       half-brightness."
100 PRINT NORMAL$;"The screen is returned to normal."
```

The information in each of the PRINT statements is displayed using the specified character attribute. Statement 70, for example, underlines the sentence that follows it.

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 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.

EXAMPLE

The following BASIC statements assign screen attribute controls to the string variables, A$, B$, C$ and D$

```
10 A$=CHR$(255)+"RN"
20 B$=CHR$(255)+"RF"
30 C$=CHR$(255)+"HN"
40 D$=CHR$(255)+"HF"
50 PRINT"This is a sample"
60 PRINT"     of "
70 PRINT"B 20 screen control"
```

When this program is executed, it displays:

```
This is a sample
     of
B 20 screen control
```

Direct mode PRINT statements are now used to change the screen attributes.

Statement	Action
PRINT A$	The entire screen reverses.
PRINT C$	The reversed screen is reduced to half-brightness.
PRINT B$	The screen returns to a positive image but remains at half-brightness.
PRINT D$	The screen returns to full brightness.

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 use the following statement:

```
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, use the following statement:

```
CHR$(255)+"VF"
```

To make the cursor visible, use the statement:

```
CHR$(255)+"VN"
```

EXAMPLE

```
10 PRINT CHR$(12)
20 PRINT CHR$(255)+"C"+CHR$(30)+CHR$(15);
30 PRINT CHR$(255)+"VF";
40 INPUT "Enter a number here"; A
```

When these statements are executed, the screen clears, the cursor is moved to column 30, row 15, and the following is displayed:

```
Enter a number here
```

Notice that the cursor is not visible. To display the cursor in future statements you must use the command:

```
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, use the following command:

```
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 use the following command:

```
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"
```

EXAMPLE

```
10 PRINT CHR$(255)+"PN"
20 FOR  I = 48 TO 125
30 PRINT CHR$(I)
40 NEXT  I
```

When these statements are executed, a column of letters, numbers, and symbols is displayed. As the screen fills with information, the last line displayed is:

Press NEXT PAGE to continue.

When the NEXT PAGE key is pressed, the display of information continues.

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.

DIRECTING VIDEO DISPLAY OUTPUT

B 20 BASIC allows you to direct video output to a particular frame of the video display. To do this, use the following statement:

```
CHR$ (255) + "X" =CHR$(<frame>)
```

<frame> indicates which frame to direct the output to and should be a value between 0 and 7.

Note that when BASIC is invoked, frame 0 is defined as the entire video display and no other frame exists. This command is meaningful only after a call to the Video Display Manager InitVidFrame operation has been made to define new frames. See the BTOS Operating System Manual for details.

EXAMPLE

```
10 PRINT CHR$ (255) + "X" + CHR$ (0)
```

This statement directs the video output to frame 0.

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, use the following statement:

```
CHR$(255)+"EF"
```

Erasing turns off all previously specified character attributes.

EXAMPLE

```
10 PRINT CHR$(255)+"C"+CHR$(1)  + CHR$(1)
20 PRINT CHR$(255)+"EF"
```

These statements move the cursor to row 1, column 1, and then erase from that point to the end of the frame.

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 character, use the following statement:

```
CHR$ + "LF"
```

EXAMPLE

```
10 PRINT CHR$ (255) + "LN" + CHR$(7)
20 PRINT CHR$ (255) + "LF" + CHR$(7)
```

Statement 10 enables the user to display the special character for the hex value 7, which is a bell. Statement 20 disables the display special characters, and thus sounds a bell. For a list of the video display characters, refer to the B 20 Software Operations Guide.

APPENDIX B
PRINTING WITH THE PRINTER SPOOLER

When writing BASIC programs on a B 20 system that uses the Spooler utility to print, a special defining function (DEFLPRINT) must be used. Before an LPRINT or LPRINT USING command is executed, a DEFLPRINT function must be defined. After the printing command has been executed, a second DEFLPRINT function must be defined to print the file from the print queue.

The DEFLPRINT function is defined with the print queue name used by your B 20 system. There are three forms of DEFLPRINT function.

```
ERC% = DEFLPRINT("[SPL]")
ERC% = DEFLPRINT("[SPLB]")
ERC% = DEFLPRINT("[NUL]")
```

DEFLPRINT("[SPL]") is used for B 20 systems that use parallel printers.

DEFLPRINT("[SPLB]") is used for B 20 systems that use serial printers.

DEFLPRINT("[NUL]") is used to print a BASIC program result that has been placed in the print queue by an LPRINT or LPRINT USING command.

LPRINT AND LPRINT USING

The following statements print the upper-case letters of the alphabet. Notice the placement of the two DEFLPRINT functions. The first DEFLPRINT specifies that the system printer is a parallel printer..

```
10 ERC% = DEFLPRINT("[SPL]")
20 FOR I = 65 TO 90
30 LPRINT CHR$(I)
40 NEXT I
50 ERC% = DEFLPRINT("[NUL]")
```

If the system printer is serial, the first DEFLPRINT function (statement 10) is:

```
10 ERC% = DEFLPRINT("[SPLB]")
```

The remaining statements are the same.

LLIST

You need the following statements to list a program on the printer:

```
ERC% = DEFLPRINT("[SPL]")
LLIST
ERC% = DEFLPRINT("[NUL]")
```

Again, if the system printer is a serial printer, the first DEFLPRINT function reads:

```
ERC% = DEFLPRINT("[SPLB]")
```

APPENDIX C
DISK INPUT/OUTPUT

Disk input/output procedures for the beginning BASIC user are examined in this appendix. If you are getting disk-related errors, 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.

SAVE "filename"[,A]

Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

LOAD "filename"[,R]

Loads the program from disk into memory. Optional R runs the program immediately. Load always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections and access the same data files.

RUN "filename"[,R]

RUN "filename" loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.

MERGE "filename"

Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory and BASIC returns to command level.

```
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.

PROTECTED FILES

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way can not be listed or edited.

There are two types of disk data files that can be created and accessed by a basic program: sequential and random access.

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 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 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 C-1 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

NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

Figure C-1. Creating a Sequential Data File.

Now look at Figure C-2. It accesses the file "DATA" that was created in the program of Figure C-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
EBENEEZER SCROOGE
SUPER MANN
Fatal Error:  Input past end in line 20
```

Figure C-2. Accessing a Sequential File

The program in Figure C-2 sequentially reads 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 wirte 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.

When used with a sequential file, the LOC function 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 simple 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 used with random files are:

```
OPEN
FIELD
LSET/RSET
GET
PUT
CLOSE
LOC
MKI$
MKS$
MKD$
CVI
CVS
CVD
LOF
```

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. The default record length 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 C-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 input in line 30 becomes the record number.

Note: Do not use a FIELD 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 C-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 C-4 below accesses the random file "FILE" that was created in Figure C-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 C-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 C-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.

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6, "DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
    "BAD FUNCTION NUMBER":GOTO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY": RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
```

Figure C-5. Inventory (Sheet 1)

```
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$###.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY": RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY": RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%; " IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=>CVI(R$) THEN PRINT "QUANTITY NOW"; Q%;" REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;
    " QUANTITY";CVI(Q$);TAB(50);"REORDER LEVEL"; CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
    GOTO 840 ELSE GET#1, PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

Figure C-5. Inventory (Sheet 2)

APPENDIX D
MATHEMATICAL FUNCTIONS

Mathematical functions 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
CALLING NON-BASIC PROCEDURES

OVERVIEW

BASIC can directly call procedures 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 must specially configure a BASIC program that calls 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.

To call non-BASIC procedures, create a run file (Basic.run) that contains the BASIC Interpreter, a lookup table (rg Procedures), 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 procedure 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 and third parameters are string constants. BASIC accepts string variables and string constants as parameters to non-BASIC procedures. When a string parameter is used, BASIC passes a pointer to the string (pb) followed by the length of the string (cb). 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 zero to 255. The type of value that the procedure returns is specified by rgProcedures.

Procedure invocations can appear in any numeric expressions including parameter lists 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 zero 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

A parameter list is a sequence of parameters separated by commas. Valid parameters are strings, pointers, and numeric expressions. You cannot pass arrays and double-precision numbers directly; however, you can pass pointers to these variables; pointers are stored as single-precision numbers.

BASIC passes parameters by value. If the parameter evaluates to an integer, a word is pushed onto the stack. If the parameter evaluates to 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 following example shows how a BASIC variable to be modified is passed to the OpenFile operation.

```
100 fh% = 0
110 erc% = OpenFile ( PTR(fh%),... )
```

Array Parameters

You pass an array by passing a pointer to its first element. Arrays typically serve as working areas or buffers. The following example 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 following example 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 distinguishing between arrays and value-returning procedures.

CONFIGURING BASIC

Creating Basic.run

To reconfigure BASIC in which non-BASIC procedures can be called, create a run file that contains the BASIC Interpreter, a lookup table (rgProcedures) defining the non-BASIC procedures, and the actual non-BASIC procedures. The process of creating a new Basic.run is described below.

To create Basic.run:

1. Copy the files contained on the distribution diskette within directory BAIBLD to your desired directory.

```
Copy
     File from              [FO]<BAIBLD>*
     File to                [SYS]<directory name>*
     [Overwrite OK?]        y
     [confirm each]
          Press GO
```

2. If you are simply configuring in Burroughs software, skip this step.

If you are configuring in your own non-BASIC procedures, invoke the Editor or Word Processor to modify BasGen.asm, the assembly language module that defines rgProcedures.

Add an entry to rgProcedures for each of your non-BASIC procedures, using an existing entry as a template. The format of rgProcedures is described in detail below.

3. Assemble BasGen.asm to produce BasGen.obj. (See the B 20 Assembly Language Manual for details on invoking the assembler.) During assembly, the assembler asks questions of this type:

Are you calling BTOS File Management (press y or n)?

Are you calling BTOS Memory Management (press y or n)?

Are you calling Forms (press y or n)?

Are you calling Sort/Merge (press y or n)?

Are you calling ISAM (press y or n)?

If you answer y (for yes) to a question, the assembler creates an entry in rgProcedures for each procedures in the corresponding Burroughs package. This enables BASIC to call the non-BASIC procedures.

To create the packaged Basic.run, answer yes to the following options:

```
Forms
ISAM
Video Access Method
Video Display Manager
Memory Management
Task Management/Contingency Management
Openfile/Closefile
File Management/Disk Management
Keyboard Management
Timer Management
Queue Management
Parameter Management
```

4. Submit the file StartBasicLink.sub.

```
Submit
  StartBasicLink.sub
```

To create a swapping Basic.run, press GO. To create a resident Basic.run, type "RES" on the [Parameter] line, then press GO.

In general, the swapping version of the Basic.run is preferable to the resident version, because it requires less memory, with only a small performance degradation.

After pressing GO, the following link command will be displayed with the following information filled in. Since the packaged version is a swapping one, the example's fields contain information necessary to build a swapping version of Basic.run.

```
Link
  Object Module                 Basgen.obj Basic.lib(...
  Run file                      Basic.run
  [List file]                   Basic.map
  [Public?]
  [Line numbers?]
  [Stack Size]
  [Max memory array size]       63600
  [Min memory array size]       6000
  System build?]
  [Version]                     'Basic 4.0'
  [Libraries]                   <Sys>ISam.lib
  <Sys>Forms.lib
  [Disk allocation?]
  [Symbol file]                 Basic.sym
```

The cursor will be positioned in the [Libraries] field. If you choose other options when assembling Basgen.asm, add the appropriate library files to this line. Remember that these libraries should be taken from the 4.0 Operating System language disk. CTOS.lib is automatically called; Therefore, do not type it on this line. Press GO.

5. Copy Basic.run to [Sys]<Sys>Basic.run (see the Executive Manual for details).

You can now invoke BASIC in the normal manner and call non-BASIC 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 procedures 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 4 fully describes these errors.

Sample Program

The following example program ties together the concepts and procedures given in this section. Follow these directions to run this program:

1. Create a directory Sort.

Copy the sort/merge program (on pages E-9 and E-10) and file Files.lst on page E-11 onto this directory.

2. Reconfigure Basic to include packages Sort/Merge and SAM (follow the directions on E-6 and E-7).

(Note: Answer yes to SAM and Sort/Merge during assembly of Basgen.asm.) One must copy this Basic.run to [Sys]<Sys>Basic.run.

3. Invoke the BASIC Command and fill its subform as follows:

Basic

```
  [Initial program]                 Sort.bas
  [Maximum number of open files]
  [Maximum random record length]
```

Press GO.

4. This program prints file Files.lst, then file Files.out. Files.out is the sorted version of Files.lst.

For details see the B 20 Sort/Merge manual.

```
1 '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
2 '
3 ' This program illustrates how to call the Sort/Merge object
4 ' modules from BASIC.  The  sorted records are assumed to be
5 ' 65 bytes long and are read from the  SAM file 'files.lst'.
6 ' The  sorted  records  are output  to the file 'files.out'.
7 ' This file will also be a SAM file.
8 '
10 DIM RGBBUFFERIN%[512]
20 DIM RGBBUFFEROUT%[512]
30 DIM BSIN%[65]
40 DIM BSOUT%[65]
50 DIM RGBWORKAREA%[4096]
60 DIM PREPARESORTBLOCK%[112]
70 DIM KEYDESCRIPTOR%[8]
80 DIM RGWSORTSTATUSBLOCK%[2]
90 RECORD$ = SPACE$(65)
100 DEF FNQUAD(WHIGH%,WLOW%)=CVS(MKI$(WLOW%)+MKI$(WHIGH%))
105 WF1$ = "WF1"
110 WF2$ = "WF2"
115 PW$ = ""
120 IN$ = "FILES.LST"
130 OUTPUT$ = "FILES.OUT"
140 SSORTWORKFILECREATE% = 1024
150 SWORKFILEINCREMENT% = 512
160 SRECORDMAX% = 65
170 SRECORD% = 65
180 MODEREAD% = ASC("m")*&h100 + ASC("r")
190 MODEWRITE% = ASC("m")*&h100 + ASC("w")
200 ERC% = 0
210 W% = 0
220 SSORTWORKAREA% = 8192
221 ' Initialize key descriptor.
222 ' 2 keys are used.
223 ' The first key is of type character, is 6 bytes long, has
224 ' an offset of  6 bytes  from the beginning of the record,
225 ' and is sorted in ascending order.
```

```
230 KEYDESCRIPTOR%[0] = 2
240 KEYDESCRIPTOR%[1] = 6
250 KEYDESCRIPTOR%[2] = 6
260 KEYDESCRIPTOR%[3] = 2
270 KEYDESCRIPTOR%[4] = &HFFFF
271 ' The second  key is of type  character, is  6 bytes long,
272 ' has  an offset  of 26 bytes from  the  beginning  of the
273 ' record, and is sorted in ascending order.
280 KEYDESCRIPTOR%[5] = 26
290 KEYDESCRIPTOR%[6] = 6
300 KEYDESCRIPTOR%[7] = 2
310 KEYDESCRIPTOR%[8] = &HFFFF
320 ERC% = OPENBYTESTREAM(PTR(BSIN%[0]),IN$,PW$,
                    MODEREAD%,PWA(PTR(RGBBUFFERIN%[0])),1024)
330 IF ERC% <> 0 THEN GOTO 640
340 ERC% = OPENBYTESTREAM(PTR(BSOUT%[0]),OUTPUT$,PW$,
                    MODEWRITE%,PWA(PTR(RGBBUFFEROUT%[0])),1024)
350 IF ERC% <> 0 THEN GOTO 640
360 ERC% =
BASICPREPAREKEYSORT(PTR(PREPARESORTBLOCK%[0]),WF1$,PW$,
      WF2$,PW$,FNQUAD(0,SSORTWORKFILECREATE%),SWORKFILEINCREMEN
T%,
      FNQUAD(0,SSORTWORKAREA%),SRECORDMAX%,0,PTR(KEYDESCRIPTOR%
[0]),
      PWA(PTR(RGBWORKAREA%[0])),PTR(RGWSORTSTATUSBLOCK%[0]))
370 IF ERC% <> 0 THEN GOTO 630
380 PRINT "RELEASE RECORDS"
390 ERC% = READBSRECORD(PTR(BSIN%[0]),PTR(RECORD$),
                SRECORD%,PTR(W%))
400 IF ERC% <> 0 THEN GOTO 440
410 ERC% = RELEASERECORD(PTR(SRECORD%),PTR(RECORD$),
                PTR(RGWSORTSTATUSBLOCK%[0]))
420 IF ERC% <> 0 THEN GOTO 630
425 PRINT LEFT$(RECORD$,64)
430 GOTO 390
440 IF ERC% <> 1 THEN GOTO 630
450 PRINT "DO SORT"
460 ERC% = DOSORT(PTR(RGWSORTSTATUSBLOCK%[0]))
470 IF ERC% <> 0 THEN GOTO 630
475 PRINT "RETURN RECORDS"
480 ERC% = RETURNRECORD(PTR(SRECORD%),PTR(RECORD$),
                PTR(RGWSORTSTATUSBLOCK%[0]))
490 IF ERC% <> 0 THEN GOTO 540
500 ERC% = WRITEBSRECORD(PTR(BSOUT%[0]),PTR(RECORD$),
                SRECORD%,PTR(W%))
520 IF ERC% <> 0 THEN GOTO 630
525 PRINT LEFT$(RECORD$,64)
530 GOTO 480
540 IF ERC% <> 3404 THEN GOTO 630
550 ERC% = CONCLUDESORT(PTR(RGWSORTSTATUSBLOCK%[0]))
560 IF ERC% <> 0 THEN GOTO 630
570 ERC% = CLOSEBYTESTREAM(PTR(BSIN%[0]))
```

```
580 IF ERC% <> 0 THEN GOTO 630
590 ERC% = CLOSEBYTESTREAM(PTR(BSOUT%[0]))
600 IF ERC% <> 0 THEN GOTO 630
610 PRINT "SORT CONCLUDED SUCCESSFULLY"
612 PRINT
615 INPUT "END OF PROGRAM - PRESS 'RETURN' TO END", A$
616 IF A$<>"" THEN GOTO 615
620 END
630 W% = TERMINATESORT(PTR(RGWSORTSTATUSBLOCK%[0]))
640 PRINT "ERROR"
650 PRINT "ERROR CODE = ";ERC%
660 STOP
```

```
Emp#  456789    6 Dep     Mares , May      Marysville,  CA     M
Emp#  123456    3 Dep     Black , Jack     Hackensack,  NJ     M
Emp#  246802    0 Dep     Jensen, Jon      Jackson,     MI     S
Emp#  012345    7 Dep     Calem , Carl     Salem,       OR     M
Emp#  246802    8 Dep     Sacks , Jack     Sacramento,  CA     M
Emp#  375573    5 Dep     Pitts , Paul     Pittsburgh,  PA     H
Emp#  012345    4 Dep     Looney, Larry    Looneyville, LA     H
Emp#  456789    2 Dep     Keyes , Keith    Key West,    FL     M
Emp#  123456    9 Dep     Johns , John     Johnstown,   NC     M
Emp#  375573    1 Dep     Alton , Allen    Altoona,     PA     S
```

APPENDIX F
PROGRAMMING HINTS

This appendix presents programming hints which will assist in using BASIC. The hints cover four areas:

1. Using SamGenAll with BASIC
2. Single- and Double-Precision Numbers in BASIC
3. Accessing the System Date and Time using BASIC
4. Miscellaneous

USING SAMGENALL WITH BASIC

The Basic.run file is configured with the following byte streams: Spooler [SPL], Video [VID], Keyboard [KBD], Null [NUL], Serial Line Printer [PTR], and Parallel Printer [LPT]. In order to use the Communication [COMM] byte stream, Basic.run must be reconfigured. The following is a step-by-step procedure:

1. Create the directory BTOS on a hard disk and copy the following files to the directory BTOS:

```
SamGenAll.obj       ISAM.Lib
Basic.Lib           StarBasicLink.sub
Forms.Lib           BasGen.asm
```

2. Rename the file SamGenAll.obj to Bas-SamGen.obj.

3. Edit BasGen.asm. Change the entry "sSamBuffer dw 512" to read "sSamBuffer dw 1024".

4. Invoke the LIBRARIAN to add the object module Bas-SamGen.obj into the Basic.Lib (see example).

```
Command LIBRARIAN
Librarian
  Library file                  Basic.Lib
  [File to add]                 Bas-SamGen.obj
  [Modules to delete]
  [Modules to extract]
  [Cross-reference file]
  [Suppress confirmation?]
```

When the message "Bas-SamGen already exists. Replace? (Press GO to confirm, CANCEL to deny)" is displayed, press GO.

5. Assemble BasGen.asm (see example).

```
Command ASSEMBLE
Assemble
  Source Files                  BasGen.asm
  [Errors only?]
  [GenOnly, NoGen, or Gen]
  [Object file]
  [List file]
  [Error file]
  [List on pass 1?]
```

Answer y (yes)to the following questions:

```
Are you calling FORMS (y or n)?  Y
Are you calling ISAM (y or n)?  Y
Are you calling the Sequential Access Method (y or n)?  Y
Are you calling the Video Access Method (y or n)?  Y
Are you calling CTOS Memory Management (y or n)?  Y
Are you calling CTOS Task Management (y or n)?  Y
Are you calling CTOS OpenFile or CloseFile Management (y or n)?
Y
Are you calling CTOS File Management (y or n)?  Y
Are you calling CTOS Keyboard Management (y or n)?  Y
Are you calling CTOS Timer Management (y or n)?  Y
```

Note: Answer Y to Queue Management and Parameter Management for 4.0 Operating system.

6. Submit StartBasicLink.sub to link BasGen.obj. Change the version to reflect the addition of the COMM byte streams and add the following libraries: ISAM.Lib, and Forms.Lib (see example).

```
Command LINK
Link
  Object modules                BasGen.obj Basic.Lib(bas-
  samgen)
  Run file                      Basic.run
  [List file]                   Basic.map
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max memory array size]       64360
  [Min memory array size]       6000
  [System build?]
  [Version]                     'Basic with COMM'
  [Libraries]                   ISAM.Lib Forms.Lib
  [DS allocation?]
  [Symbol file]                 Basic.Sym
```

The following program transmits a COMM byte stream out of the COMM B port.

```
10 ERC% = DEFLPRINT("[COMM]B")
20 IF ERC% <> 0 THEN PRINT "ERROR.  ERC ="; ERC%: GOTO 70
30 FOR I = 16 TO 254
40 PRINT CHR$(I);
50 LPRINT CHR$(I);
60 NEXT I
70 END
```

To test the new Basic.run with the above source:

Create the configuration file "COMMBConfig.Sys" using the default parameters. Perform a COPY from [COMM]B to [VID] on the receiving B20. Run this program on the transmitting B20. The receiving B20 should display the complete ASCII character set.

SINGLE- AND DOUBLE-PRECISION NUMBERS IN BASIC

There are some real numbers in BASIC that 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. A way to prevent the incorrect printout of these values is to use the PRINT USING or LPRINT USING commands. These commands round the real numbers to the desired amount of decimal places and thus give a correct printout of the values. Some examples of these are shown below:

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 this BASIC program is run, the values printed are shown below. The first column printed uses the PRINT command and the second column used the PRINT USING command.

```
 8.649999648332596D-02          0.0865
 9.619999676942825D-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# = 8.649999648332596D-02
20 B# = 9.619999676942825D-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 0.1
30    LPRINT TAB(10); I; TAB(25);
40    LPRINT USING "#.#"; I
50 NEXT I
60 ERC% = DEFLPRINT("[NUL]")
70 END
```

The following is a printout of the above program using BASIC. The values printed range from one to five and are incremented by a value of 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.

```
          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
3.999999 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
```

ACCESSING THE SYSTEM DATE AND TIME USING BASIC

Most applications occasionally need to include the current date and/or time in their processing. There are several procedural calls available in BTOS to allow the user to retrieve the date and time field from the system and expand it into a readable day, date, and time.

There are two basic structures involved with date and time manipulation in BTOS. The date and time is kept internally in system memory as a three-word field containing the count of 50 or 50Hz clock ticks, the count of 100ms periods elapsed since the last second, the count of seconds since midnight or noon, and the count of 12-hour periods since March 1, 1952. (See the B20 Operating System (BTOS) Reference Manual, the "Timer Management" section.) The last two words are returned to the program when the date/time is requested; the first word can be examined when precise timings are needed. The expanded date and time format is a four-word structure with the year, month, day of month, day of week, hour, minute, and second embedded in it.

The compact system format can be used to time-stamp records, for example, while occupying only a four-byte field. The format of the compacted date also makes it useful for date calculations. For example, the date of 30 days from now can be obtained by adding 60 (12-hour periods) to the count, which specifies days in the system format, then expand it from there. If two dates are subtracted, the result divided by two is the number of days apart the two are. The day-of-week field can also be examined in a program (it is returned initially as a number: 0=Sun to 6=Sat) to perhaps look for the next business day after thirty days from now.

The following calls are available in BTOS to access the system date/time structure and are documented in the BTOS Operating System Manual.

CompactDateTime	Converts the expanded date/time format to the system format.
ExpandDateTime	Expands the system format to the expanded date/time format.
GetDateTime	Returns the current date and time in the system format.
SetDateTime	Sets the date and time for the system.

Analyzing the expanded date/time format using these routines can be tricky in high-level language. The expanded date is returned to the program as a 64-bit data type, for which few of the languages have a built-in structure. However, the facilities are available for the information to be extracted.

On page F-7 is an example of a BASIC program that obtains the system date and time and expands it. The route displays the day of the week, the date, and the time obtained from the system.

Included in the release of BASIC is function "DATETIME$", which returns a string containing a date and/or time. "DATETIME$" accepts one parameter that is a string template specifying the format of the date and time to be returned. For example:

```
10 PRINT DATETIME$(!*w! !*n! !*d! !*y! at !*h!:!0m! !AA!")
```

generates "Thursday April 21, 1983 at 11:11 AM". In a BASIC program run via the BASIC Interpreter, the "DATETIME$" function call is not available, but the procedural calls, "GetDateTime", "ExpandDateTime", etc., are available to the interpretive program.

If "GetDateTime" and "ExpandDateTime" are to be used in a BASIC program, the 64-bit expanded date and time can be returned to an array of four integers, each one word in length. The year is then equal to the first element of the array (line 250). Next, the byte fields must be extracted from the field. To do this, make a pointer to the desired word and "PEEK" at a byte of it (line 270). Note that the month is stored in zero-to-11 format, so using "GETRA" and "GETSA" (lines 290 and 300), the offset portion may then be incremented. Finally, use "MAKEPOINTER" (line 320) remembering that in a program such as this with interpretive BASIC using pointers, all variable names should be declared at the beginning of the program.

```
10 DATETIME = 0
20 ERC% = 99
30 YEAR% = 1999
40 MONTH% = 9
50 DAY% = 99
60 DAYNBR% = 9
70 HOUR% = 99
80 MIN% = 99
90 SEC% = 99
100 DIM D%[4]
120 I% = 99
130 J% = 99
140 DIM DAYOFWK$[7]
150 DAYOFWK$[0] = "SUN"
160 DAYOFWK$[0] = "MON"
170 DAYOFWK$[0] = "TUE"
180 DAYOFWK$[0] = "WED"
190 DAYOFWK$[0] = "THU"
200 DAYOFWK$[0] = "FRI"
210 DAYOFWK$[0] = "SAT"
230 ERC% = GETDATETIME(PTR(DATETIME))
240 ERC% = EXPANDDATETIME(DATETIME,PTR(D%[0]))
250 YEAR% = D%[0]
260 P = PTR(D%[0])
270 MONTH% = PEEK("B",P)
280 MONTH% = MONTH% + 1
290 I% = GETRA(P)
300 J% = GETSA(P)
310 I% = I% + 1
320 P = MAKEPOINTER(I%,J%)
330 DAY% = PEEK("B",P)
340 P = PTR(D%[2])
350 DAYNBR% = PEEK("B",P)
360 P = PTR(D%[2])
370 I% = GETRA(P)
380 J% = GETSA(P)
390 I% = I% + 1
400 P = MAKEPOINTER(I%,J%)
410 HOUR% = PEEK("B",P)
420 P = PTR(D%[3])
430 MIN% = PEEK("B",P)
440 I% = GETRA(P)
450 J% = GETSA(P)
460 I% = I% + 1
470 P = MAKEPOINTER(I%,J%)
480 SEC% = PEEK("B",P)
490 PRINT "DAY OF WEEK IS "; DAYOFWK$[DAYNBR%]
500 PRINT "DATE IS "; MONTH% "/"; DAY%; "OF THE YEAR"; YEAR%
510 PRINT "TIME IS "; HOUR%; ":"; MIN% ":"; SEC%
```

Miscellaneous

Graphics:

Passing real parameters to graphics procedures must be of 8087 format. Use the procedure Convertto8087 to create these parameters.

To reconfigure BASIC to include graphics procedures, you must:

1. Edit file Basgen.asm.

2. Remove the characters "%'" from in front of each table entry that you want added to BASIC.

3. Answer yes to the question "Are you calling Graphics", within the assembly of file Basgen.asm.

   The BASIC Interpreter can be customized to operate with 3.0 Graphics on a 4.0 operating system. However, during the customization process, the following error statements may appear (depending on your hardware and system configuration). They can be ignored by the user.

   1. Multiply defined symbol QO in file [SYS]<SYS>graphics.lib (grfheap1)

   2. Multiply defined symbol LOADCOLORSTYLERAM in file [SYS]<SYS>CTOS.lib (COLOR)

   3. Multiply defined symbol SETSTYLERAM in file [SYS]<SYS>CTOS.lib (COLOR)

   4. Warning: Proc near 0000 in [SYS]<SYS>CTOS.lib (awsron) doesn't follow CALL/RET conventions

If the BASIC Interpreter is configured with Graphics, the EASY application is unsupported.

LINKING WITH NONSTANDARD SEGMENTS

BASIC depends on a particular ordering of segments in memory for correct operation. (The Linker Manual describes segment ordering.) If you are calling object modules created with FORTRAN or Pascal compilers, the required segment order is guaranteed.

However, if you are calling object modules produced by the Assembler, the required segment ordering is guaranteed only if you restrict segment class names to "data", "stack", "const", "heap", or "code".

APPENDIX G
INSTALLATION PROCEDURES

The following are installation procedures for invoking the Install.sub and XEInstall.sub for the BASIC Interpreter. (For Dual Floppy Standalone, see Appendix H.)

HARD DISK INSTALLATION

Boot the system from the master or cluster where the installation of BASIC Interpreter is desired. The software is installed in the system files of the system directory.

- Insert the BASIC Interpreter 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 Interpreter 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 H
DUAL FLOPPY STANDALONE INSTALLATION

To install the dual Floppy Standalone configuration, insert Disk One of the 4.0 Standalone Operating System into [F0] and power on the system. After signing on, you can load BASIC from drive [F0]. To do this, insert the distribution diskette into [F0], then enter BASIC and press GO.

If you wish to load BASIC from [F1], you must first modify your .user file on your system disk to access the Sys.Cmds file on this drive before loading BASIC. Instructions for this modification are given in the B 20 Systems Operating System (BTOS) Reference Manual (form 1171675).

HELPFUL HINTS FOR DUAL FLOPPY

STANDALONE SYSTEMS

After performing the installation, you can remove the disk and insert a source disk into [F1].
Type "Load [F1]<Sys>programname", then press <RETURN>.

Note that unless a volume name is specified, all actions are routed to [F0] unless you have changed this default before entering BASIC. This means that "unspecified" files are created on [F0]. For example: if you are calling a form, and the open call did not specify disk [F1], the system looks for the form on [F0].

It is suggested that you copy all the files needed for execution of a program (except the actual program) on one disk. When needed, insert that disk into [F0]. For example, when printing from BASIC to a parallel or serial printer, the system looks for Lptconfig.sys or Ptrbconfig.sys (depending on the printer) on [F0].

It is also suggested that you copy Exec.run from the first disk of the Operating System package to the disk described in the previous paragraph. This enables you to come back to the system after finishing a Basic session. Otherwise, you are prompted to "Please mount a system volume in [sys], Then Press GO". Copying Exec.run enables one to come back to the system. Doing this provides you with the commands such as COPY, DELETE, etc.

ISAM: If you are creating a data set (unspecified), the files will be created on the disk in [F0]. If you are reading or modifying a data set (unspecified), the files must be on [F0].

ISAM INSTALLATION 4.0

Dual Floppy Standalone:

Insert disk B26IF4-1 into [F0]. Type "ISAM Install", then press <RETURN>. The following subform will be displayed:

Isam Install
[No. of ISAM users (default from OS configuration)]
[Configuration file ([Sys]<Sys>Isam.Config)]

Fill in the parameters with the appropriate values, then press <RETURN>. See B 20 Systems Indexed Sequential Access Method (ISAM) Reference Manual (form 1168465) for more information.

Accessing ISAM from Basic is the same for 4.0 and 3.1, but ISAM 4.0 contains some new calls, as described in its manual.

INDEX

Documentation Evaluation Form

Title: B 20 Systems BASIC Interpreter Reference Manual Release 4.0

Form No: 1180155

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.